

Hard Fault
или
Фундаментальные Основы
Программирования Микроконтроллеров
(версия после цензуры)

Александр Барунин

18 марта 2025 г.

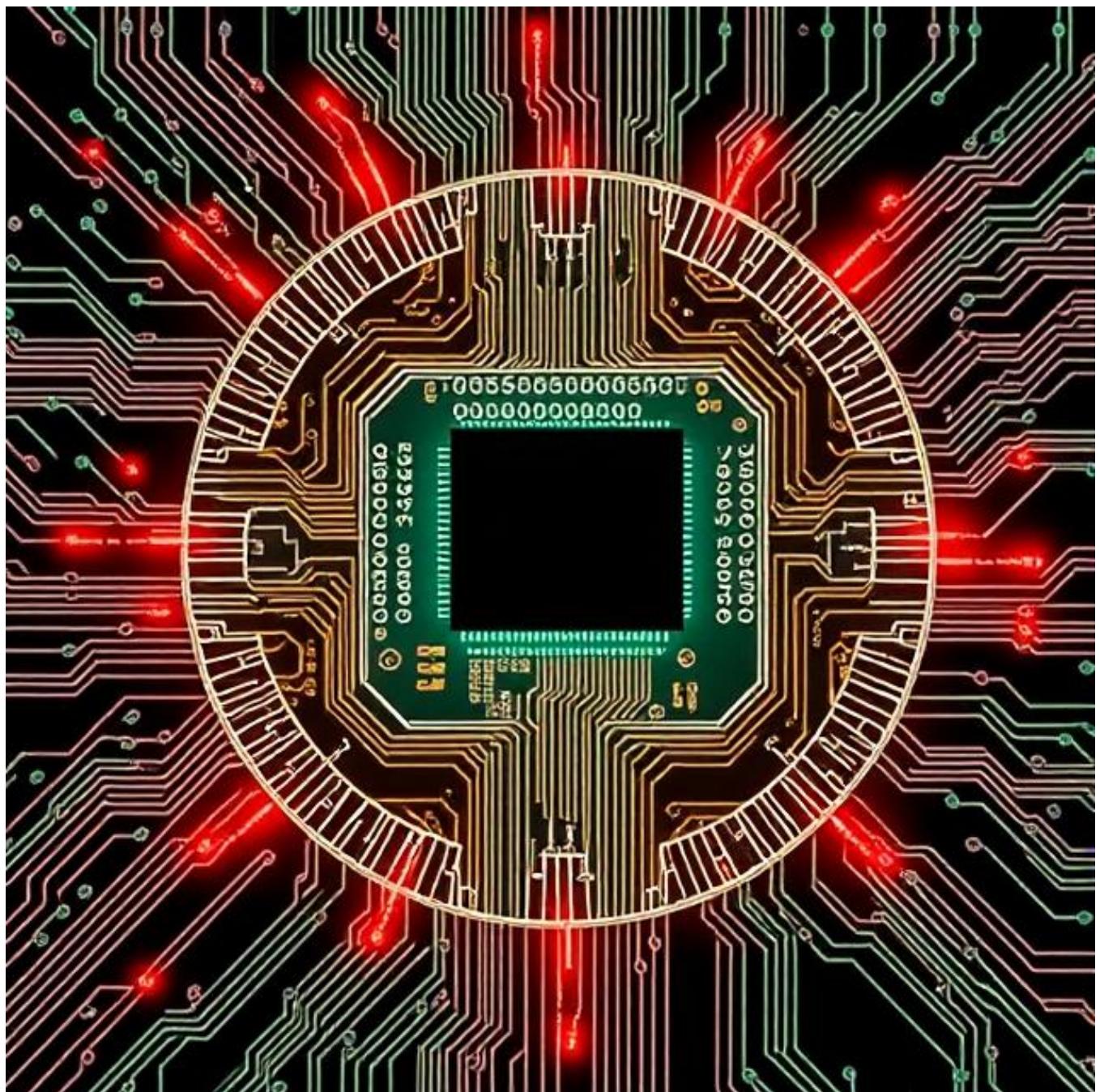


Рис. 1:

Оглавление

1	Об авторе	11
2	Предисловие	13
3	Отзывы благодарных читателей	15
4	Аббревиатуры	17
5	Определения	21
6	Настройка ToolChain-на для Разработки	27
6.1	Пролог	27
6.2	Почему микроконтроллеры программируют на Си?	27
6.3	Каков план?	28
6.4	Что надо из оборудования?	28
6.5	Что надо из документации?	29
6.6	Что надо из ПО?	29
6.7	Установка текстового редактора	30
6.8	Установка GCC Cross-компилятора для ARM	30
6.9	Установка Build Tools binaries	32
6.10	Скачивание SDK от вендора	33
6.11	Система сборки	34
6.12	Препроцессор	34
6.13	Компиляция	35
6.14	Компоновка	37
6.14.1	Какую выбрать libc?	37
6.14.2	Секции	37
6.14.3	Таблица векторов прерываний	39
6.15	Настройка пошаговой отладки.	40
6.15.1	Отладочный сервер	40
6.15.2	Отладочный клиент	40
6.16	Блок-схема ToolChain-а	40
6.17	Последовательность пуска микроконтроллера	40
6.18	Обработка прерываний на микроконтроллерах	43
6.19	UART-CLI Для отладки	45
6.20	Итог	46

6.21	Ссылки	46
7	Атрибуты Хорошой Прошивки (Firmware)	49
7.1	Микросекундный UpTime счетчик (камертон)	49
7.2	Сторожевой таймер (сторожевой пёс)	49
7.3	Загрузчик (BootLoader)	49
7.4	NVRAM (числохранилище)	50
7.5	Модульные тесты (скрепы)	50
7.6	Health Monitor (медбррат)	50
7.7	Command Line Interface (CLI-шка)	50
7.8	Диагностика	53
7.9	Black-Box Recorder (Чёрный ящик)	54
7.10	Переход в энергосбережение	54
7.11	Аутентификация бинаря (optional)	54
7.12	Итоги	55
7.13	Гиперссылки	56
8	Архитектура Хорошего Кода Прошивки (Массив - наше всё)	57
8.1	Пролог	57
8.2	Массивы конфигурационных структур	58
8.3	Массив функций инициализаций прошивки	59
8.4	Массив функций для суперцикла	62
8.5	Массив параметров в NVRAM	62
8.6	Массив отладочных токенов для диагностики	63
8.7	Массив команд для CLI	64
8.8	Массив функций-модульных тестов	65
8.9	Массив функций потоков для RTOS	66
8.10	Итоги	67
8.11	Гиперссылки	67
9	Модульное Тестирование в Embedded	69
9.1	Пролог	69
9.2	Достоинства модульных тестов	69
9.3	Недостатки модульных тестов	71
9.4	Общие принципы модульного тестирования	72
9.5	Автотесты	74
9.6	Итоги	75
9.7	Гиперссылки	76
10	Атрибуты Хорошего Загрузчика	77
10.1	Пролог	77
10.2	Терминология	77
10.3	Достоинства загрузчика	78
10.4	Недостатки загрузчика	78
10.5	Атрибуты хорошего загрузчика	79

10.6 Итоги	82
10.7 Гиперссылки	82
11 Атрибуты Хорошего С-кода	83
11.1 Пролог	83
11.2 Фундаментальные правила оформления Си-кода	83
11.3 Про функции	83
11.4 Оформление переменных и констант	85
11.5 Файлы	86
11.6 Оформление кода	87
11.7 Оформление процесса конфигурации сборки и тестирования	91
11.8 Аномалии оформления сорцов из реальной жизни (War Stories)	93
11.9 Финальная структура добротного программного компонента	94
11.10 Итог	95
11.11 Источники	95
12 Что Должно Быть в Каждом FirmWare Репозитории	97
12.1 Пролог	97
12.2 Компонент для поддержки каждого конкретного процессорного ядра (Core)	98
12.3 Компонент для поддержки каждого конкретного микроконтроллера (microcontroller)	98
12.4 Компонент для поддержки каждой конкретной платы (platform code,boards)	98
12.5 Отдельная папка для каждой конкретной сборки (projects)	98
12.6 Абстрактные структуры данных (ADT)	98
12.7 Загрузчик (BootLoader)	99
12.8 MicroController Abstraction Layer (MCAL)	100
12.9 Драйверы периферийных чипов (ASICs)	102
12.10 Папка со всем оставшимся (miscellaneous)	102
12.11 Чужой код (Third Party)	102
12.12 Sensitivity	102
12.13 Computing	103
12.14 Connectivity	105
12.15 Storage	106
12.16 Код для информационной безопасности (Security)	106
12.17 Модульные тесты	107
12.18 Сборка из Make	107
12.19 Скрипты сборки CMake	107
12.20 Скрипты автосборки	107
12.21 Скрипты авто прошивки	108
12.22 Скрипты отчистки и автоматического форматирования	108
12.23 Статические библиотеки	108
12.24 Кодогенераторы	108
12.25 Авто генерация документации	108
12.26 Какие папки должны быть в репозитории?	109

12.27 Итог	109
13 Архитектура Хорошо Поддерживаемого драйвера	111
13.1 Пролог	111
13.2 xxx_drv.c, xxx_drv.h с функционалом	111
13.3 xxx_isr.c, xxx_isr.h	112
13.4 Файл xxx_types.h с типами	112
13.5 Отдельный xxx_const.h файл с перечислением констант	113
13.6 xxx_param.h файл с параметрами драйвера	113
13.7 config_xxx.c config_xxx.h файл с конфигурацией по умолчанию	114
13.8 xxx_commands.c xxx_commands.h файл с командами CLI	115
13.9 xxx_diag.c/xxx_diag.h файлы с диагностикой	116
13.10 test_xxx.c/test_xxx.h Файлы с модульными тестами диагностикой*	116
13.11 Make файл xxx.mk для правил сборки драйвера из Make	117
13.12 xxx_dep.h файл с проверками зависимостей	118
13.13 xxx_preconfig.mk	119
13.14 Функционал драйвера	120
13.15 Итоги	123
13.16 Гиперссылки	124
14 DevOps для производства Firmware	125
14.1 Пролог	125
14.2 Репозиторий с кодом (репа/общак)	125
14.3 Код-генерация	126
14.4 Проект должен собираться скриптами	126
14.5 Сборок должно быть много	126
14.6 Проект собирать Make файлами	127
14.7 Статические анализаторы	127
14.8 Автосборки	127
14.9 Сервер сборки	128
14.10 Модульные тесты (скрепы)	128
14.11 Hardware In The Loop (HIL) стенд	128
14.12 Code Coverage (высший пилотаж)	129
14.13 Ежедневные планерки	129
14.14 Эпilog	129
14.15 Гиперссылки	129
15 Способы Отладки и Диагностики FirmWare	131
15.1 Пролог	131
15.2 Модульные тесты (скрепы / гипс)	131
15.3 Health Monitor (Мед брат)	131
15.4 CLI(шка) (Command Line Interface) (или Компьютерная томография МРТ)	132
15.5 HeartBeat LED (медленный стетоскоп)	134

15.6	Пошаговая отладка GDB через SWD/JTAG (или Хирургическое вмешательство с наркозом или КТ)	134
15.7	Утилита arm-none-eabi-addr2line.exe (УльтраЗвуковое Исследование УЗИ)	134
15.8	Функции Assert (или ПЦР тест)	135
15.9	GPIO и осциллограф (Кардиограмма)	135
15.10	GPIO и логический анализатор (Электроэнцефалография)	135
15.11	DAC (Цифро-Аналоговый Преобразователь) (или микроскоп)	136
15.12	Логирование на дисплей (Глюкометр)	136
15.13	Утилита STM-Studio (Электроэнцефалография)	137
15.14	Логирование в SD карту (Копрограмма)	137
15.15	Эмуляция прошивки как процесса на PC (клонирование)	138
15.16	DMM, цифровой мультиметр (градусник)	139
15.17	Контроль качества пайки (Load-detect)	139
15.18	Вывод системной частоты наружу (стетоскоп)]	139
15.19	Итог	140
15.20	Гиперссылки	140
16	Почему Нам Нужен UART-CLI?	
	(Добавьте в Прошивку Гласность)	141
16.1	Пролог	141
16.2	Почему люди используют CLI?	143
16.3	Аналогии CLI на бытовом уровне	151
16.4	Где уже используют CLI	152
16.5	Список наиболее часто употребительных команд CLI	153
16.6	Некоторые незначительные недостатки CLI	155
16.7	Итоги	156
16.8	Гиперссылки	156
17	Почему Сборка с Помощью Eclipse ARM GCC Плагинов это Тупиковый Путь.	159
17.1	Пролог	159
17.2	Теория сборки Си программ в IDE с плагинами	160
17.3	Пошаговая отладка	165
17.4	Минусы сборки Eclipse плагинами из-под IDE	165
17.5	Достоинства сборки из-под Eclipse с ARM плагинами	167
17.6	Аналогии	167
17.7	Что делать?	168
17.8	Итоги	168
17.9	Гиперссылки	169
18	Почему важно собирать код из скриптов?	171
18.1	Пролог	171
18.2	Как отлаживаться	178
18.3	Достоинства сборки кода из Make файлов	178

18.4	Аналогии	182
18.5	CMake или GNU Make?	183
18.6	Кто собирает код из скриптов?	184
18.7	Вывод	184
18.8	Гиперссылки	186
19	ЛикБез по GNU Make	187
19.1	Переменные make	188
19.2	Как организовать скрипты сборки GNU Make?	188
19.3	Итоги	198
19.4	Гиперссылки	199
20	Диспетчер Задач для Микроконтроллера	201
20.1	Постановка задачи	201
20.2	Определимся с терминологией	202
20.3	Отладка планировщика	206
20.4	Достоинства данного планировщика	208
20.5	Недостатки данного планировщика	208
20.6	Итог	208
20.7	Гиперссылки	209
20.8	Контрольные вопросы	209
21	NVRAM для микроконтроллеров	211
21.1	Пролог	211
21.2	Nvram по-русски	211
21.3	Достоинства on-chip NorFlash	212
21.4	Типичные требования к NVRAM	213
21.5	Терминология	213
21.6	Основная идея механизма NVRAM	214
21.7	Добавление IDшников и интерпретатора (PARAM)	217
21.8	Итоги	218
21.9	Гиперссылки	218
22	Конечные автоматы на службе программирования микроконтроллеров	221
22.1	Пролог	221
22.2	Фазы проектирования конечного автомата	222
22.3	Терминология конечных автоматов	222
22.4	Демонстрационная задача на FSM	222
22.5	Основная идея	223
22.6	Какие могут быть проблемы в Н-мосте?	224
22.7	Определить входы конечного автомата	225
22.8	Перечислить все возможные состояния конечного автомата	225
22.9	Определить действия для данного конечного автомата	226
22.10	Построить таблицу переходов	227

22.11 Построить таблицу выходов (Action Table)	227
22.12 Нарисовать граф конечного автомата	228
22.13 Сформировать Look-Up таблицу для принятия решения	229
22.14 Отладка	231
22.15 Достоинства cross-detect	231
22.16 Что можно ещё улучшить?	233
22.17 Итоги	234
22.18 Контрольные вопросы	235
22.19 Гиперссылки	235
 23 Коллоквиум по программированию микроконтроллеров	237
23.1 Вопросы по коду	237
23.2 Системы сборки	238
23.3 Структуры данных	239
23.4 Про DevOps	239
23.5 Про прерывания	239
23.6 Про ToolChain	239
23.7 Вопросы про RTOS(ы)	240
23.8 Про цифровые фильтры	241
23.9 Вопросы по аналоговой схемотехнике	241
23.10 Про железо (аппаратное обеспечение)	241
23.11 По интерфейсам	242
23.12 По протоколам	243
23.13 Вопросы про стек	243
23.14 Беспроводные интерфейсы	244
23.15 Про heap память	244
23.16 Про загрузчики (Bootloader)	244
23.17 Решение проблем (TroubleShooting)	244
23.18 Вопросы для развернутого устного ответа (System Design)	245
23.19 Вопросы для проверки навыков пользования компьютером	246
23.20 Трудные вопросы (со звездочкой *)	246
23.21 Вопросы на способность тестирования и отладки	246
23.22 Варианты для тестового задания дома	247
 24 Как Чинить Баги при Программировании Микроконтроллеров	249
24.1 Пролог	249
24.2 Примеры багов из реальной разработки	250
24.2.1 Курящий разработчик	250
24.2.2 Толстые пальцы	250
24.2.3 USB-A	251
24.2.4 Slip-ring	252
24.2.5 Сектор конфигурации SoC-а	252
24.2.6 Статическое электричество	254
24.2.7 Хрипящий голос	254
24.2.8 Забыли вставить в розетку	254

24.2.9 Не тот кварцевый осциллятор	254
24.2.10 Ток которого нет	256
24.2.11 Заклинивший CAN трансивер	256
24.2.12 Упс! Микроконтроллер перепутали...	257
24.2.13 Ошибка внутри sprintf()	258
24.2.14 Загадочные прерывания	258
24.2.15 Ошибка 71-ой минуты	259
24.2.16 Низкая дальность LoRa link-a	259
24.3 В чём проблема починки ошибок?	260
24.4 Как Чинить Программные Ошибки?	260
24.5 Терминология	260
24.6 Классификация программных ошибок	260
24.7 Классификация причин программных ошибок	260
24.8 Универсальная методичка починки багов	261
24.8.1 Фаза № 1. Подробно опишите программную ошибку	262
24.8.2 Фаза № 2. Научитесь стабильно воспроизводить баг	262
24.8.3 Фаза № 3. Выдвинете предположения (гипотезы)	262
24.8.4 Фаза №4. Делайте эксперименты	262
24.8.5 Фаза №5. Активировать более глубокий уровень логирования	262
24.8.6 Фаза №6. Прогоните модульные тесты	263
24.8.7 Фаза №7. Пройдите код пошаговым GDB отладчиком	263
24.8.8 Фаза №8. Прогоните код через статический анализатор.	263
24.8.9 Фаза №9. Задайте вопрос на профессиональных форумах и сообществах.	263
24.9 Итог	263
24.10 Гиперссылки	264
25 Литература	265

Глава 1

Об авторе

Автор учебника - это российский инженер программист Hi-Tech электроники. Питомец Национального Исследовательского Университета МИЭТ (Московский Институт Электронной Техники) 2015 года. Автор начинал работу в лаборатории министерства обороны. Разрабатывал компоненты для дистанционного управления бронетранспортёрами. Затем трудился в start-up проектах, которые разработали серию IoT приборов для умных домов. Главным образом это были счётчики электричества с подключением к интернету. Последние 7 лет разрабатывает system software для электроники в автопроме: телематика, infotainment, body-control units. Автор написал firmware более чем для шестидесяти электронных плат на основе микроконтроллеров с архитектурами AVR, ARM и Power PC. С 2012 года автор обладает 13-летним опытом и экспертизой в проблемах разработки программного обеспечения для встраиваемых систем на основе современных микроконтроллеров. Автор десятков публикаций по программированию в общем.

Глава 2

Предисловие

"Лучший способ в чём-то разобраться - написать про это учебник."
(Ричард Фейнман)

Предлагаемая вашему вниманию книга написана по материалам моих текстовых заметок в культовом русскоязычном сообществе ИТ разработчиков habr.

Эта книга адресована всем нынешним программистам микроконтроллеров, студентам технических ВУЗов, а также тем, кто думает заниматься программированием микроконтроллеров.

Чтобы работать с этим учебником надо знать синтаксис и семантику языка программирования Си.

Я решил написать учебник по программированию МК на основе своего инженерного опыта.

Тексты на habr были подвергнуты широчайшим общественным обсуждениям в комментариях. И, тем самым, мною были внесены улучшения по формулировкам и содержанию.

Изначально я очень скептически относился к своим текстам. Однако со временем стал регулярно получать благодарственные сообщения от искренних читателей из разных городов России, Латвии, Германии и даже США.

Если это такой хороший материал, то он не должен пойти прахом.

Благодарности сподвигли меня скомпоновать отдельную брошюру, а потом и полноценный учебник посвященный проблемам разработки системного программного обеспечения для микроконтроллеров.

Учебник я писал в свободное от основной работы время. По ночам, по выходным, в государственные праздники и в редкие отпуска. Этот учебник собирался по крупицам годами.

В довесок к этому, я работал в английском автопроме и с горечью осознавал как далеко отстали российские технологии программирования микроконтроллеров от техноМЫСЛИ Запада.

В российских реалиях у Embedder-ов очень много отрицательного опыта. Многие российские электронные разработки ушли в штопор, другие зациклились погрязнув в бесмысленной для дела ИТ-муштре.

Причина проста. Инженеры-программисты из передовых государств не приезжают работать в российские технические компании. Они опытом не делятся.

Высокие технологии не импортируются в Россию из-за экспортных ограничений ITAR установленных Западом. Из Запада в Россию поступала только готовая электронная продукция без какой бы то ни было документации, схемотехники и исходных кодов. Это автомобили, персональные компьютеры, мобильные телефоны и прочее.

Поэтому своевременность данного учебного материала объясняется убийственным отставанием российской школы программирования микроконтроллеров от практик программирования MCU в западной Европе и США.

Очень сложно научить программировать микроконтроллеры в ВУЗе. В ВУЗах преподают десятки других классических наук и спрашивают везде по первое число. Поэтому культура программирования приходит только с опытом и формируется годами.

В моей книге представлен эксклюзивный материал. Это сродни разоблачению фокусов. Таким обычно не делятся. Благодаря рекомендациям из этой книги у Вас будет разработка без хлопот (*hassle free development*).

Для работы с этим учебником Вам понадобится персональный компьютер (PC), любая твёрдотельная учебно-тренировочная электронная плата с микроконтроллером ARM Cortex-M, кабель USB, переходник с USB на UART и программатор для MCU на отладочной плате.

Из программного обеспечения понадобится текстовый редактор, компилятор GCC, терминал последовательного порта TeraTerm (или PuTTY) и утилита GNU Make.

Как автор учебника, я хотел бы выразить глубочайшую благодарность своему учителю, наставнику и визионеру Владимиру Романову опыт работы с которым помог мне освоить эту непростую профессию в значительно большей степени. Ни до, ни после работы с Владимиром, я так и не встречал людей с более прогрессивным опытом, взглядами и экспертизой.

Если бы я знал все вещи указанные в учебнике сразу после ВУЗ(за), то жизнь сложилась бы лучше

Александр Барунин
aabzele@gmail.com

Глава 3

Отзывы благодарных читателей

1. (г. Екатеринбург, Россия)

"Добрый день. Спасибо за статьи на хабре, дружище) зачётно пишешь, живой жизненный слог)) весело, толково, доходчиво и запоминаемо, приятно читать)) благодаря твоей статье я понял, что компания в которой я трудоустроен довольно неплохая, почти все критерии отбора исполняются и стиль написания фирмвари питерский) Спасибо ещё раз за статью, несколько раз перечитал её, каждый раз новое и интересное нахожу, просто мало авторов которые так хорошо пишут об этой профессии.

У вас писательский талант без преувеличения, понятно пишите и передаёте опыт новичкам. в общем вы правы... "

С уважением, Андрей

2. (г.Уфа, Россия)

"Повезло, что прочитал Ваши статьи: Как собрать Си программу в OS windows. Как Вы работали программистом, про правила написания кода. Вы так с юмором, просто и ясно пишете, что решил Вам написать. Человек, который имеет мужество написать: "Я только в 4й организации по-настоящему программировать микроконтроллеры научился. внушает уважение к своим уникальным знаниям.

С уважением, Раис

3. (г Санкт-Петербург, Россия)

"Давно читаю ваши статьи — с чем-то согласен, с чем-то нет. С уважением, Даниил "

4.

"Содержание книжки очень даже хорошее. Особенно понравилась девятнадцатая глава "Вы в самом деле хотите стать программистом микроконтроллеров?". Правда, в ней несколько провокационно и излишне "голо" раскрываются некоторые аспекты работы в области. "

5. (Бостон, USA)

"Привет, хочу немного поддержать Вас морально, видя полное непонимание со стороны читателей статьи "Градация Навыков ... на самом деле статья правильная, но аудитория находится далеко по своему опыту и ментальности .. статьи Ваши нравятся, толково написано .."
"

Всего доброго, Виктор (30+ лет в разработке) "

6. (г. Санкт-Петербург, Россия)

"Хорошая книжка, нестандартная. "Артем

7. (г. Санкт-Петербург, Россия)

"С технической точки зрения материал очень даже неплох" Андрей (FPGA разработчик)

8. (г. Москва, Россия)

"... данная работа не лишена интересных авторских наблюдений и искрометного юмора ..." Илья (Старший инженер-программист)

Глава 4

Аббревиатуры

При работе с микроконтроллерами Вам придется моментально узнавать вот эти акронимы:

Акроним	Расшифровка
ACI-	Arm Custom Instructions
ADC-	Analog-to-digital conversion
AHB-	Advanced High-performance Bus
AMBA-	Advanced Microcontroller Bus Architecture
APB-	Advanced Peripheral Bus
API-	Application Programming Interface
ARM	Advanced RISC Machines
ASIC-	Application-specific integrated circuit
ATB-	Advanced Trace Bus
AVR-	Advanced Virtual RISC (Alf and Vegard RISC)
AXI-	Advanced eXtensible Interface
BGA-	Ball grid array
BIOS	basic input/output system
BPU-	Breakpoint Unit
BSD -	Berkeley Software Distribution
C-AHB-	Code AHB
CAD-	Computer-aided design
CDE-	Custom Datapath Extension
CI-	Continuous Integration
CIC-	Cascaded integrator-comb
CLI-	Command-Line Interface
CMSIS-	Common Microcontroller Software Interface Standard
CPP-	C PreProcessor
CRC-	Cyclic Redundancy Check
CTI-	Cross Trigger Interface
CTM-	Cross Trigger Matrix
D-AHB-	Debug AHB
DAC-	digital-to-analog converter

DDS-	direct digital synthesis
DFM-	design for manufacturability
DFU-	device firmware update
DMA-	Direct memory access
DMM-	Digital MultiMeter
DSP-	Digital Signal Processing
DWT-	Data Watchpoint and Trace
EEPROM-	Electrically Erasable Programmable Read-Only Memory
EFI	Extensible Firmware Interface
EMC-	ElectroMagnetic Compatibility Электромагнитная совместимость
EPPB-	External Private Peripheral Bus
ESD-	ElectroStatic Discharge
ETM-	Instruction Trace Embedded Trace Macrocell
EoL-	End-of-life
FOTA-	Firmware Over The Air
FPU-	Floating-point Unit
FS-	File System
FSM-	Finite-State Machine
FW-	FirmWare
GCC-	GNU Compiler Collection
GDB-	GNU DeBugger
GNU-	GNU's Not UNIX
GPIO-	General-purpose input/output
GPU -	Graphics Processing Unit
GUI-	Graphical User Interface
HMI-	Human-Machine Interface
I2C-	Inter-Integrated Circuit
IAR-	Ingenjörsfirma Anders Rundgren
ID-	Identifier
IDAU-	Implementation Defined Attribution Unit
IDE-	integrated development environment
IEEE-	Institute of Electrical and Electronics Engineers
IF-	interface
IO-	Input/Output
ISR-	Interrupt Service Routine
ITAR-	International Traffic in Arms Regulations
ITM-	Instrumentation Trace Macrocell
JTAG-	Joint Test Action Group
KIIS-	Keep It ISOLATED Stupid
LED-	light-emitting diode
LLVM-	Low Level Virtual Machine
LSB-	least significant bit
MAC-	Multiply and Accumulate
MBR-	Master Boot Record

MCAL-	MicroController Absorption Layer
MCO-	Multiplexed Clock-Out
MCU-	Micro Controller Unit (Микроконтроллер)
MPSSE-	Multi-Protocol Synchronous Serial Engine
MPU-	Memory Protection Unit
MSB-	most significant bit
MTB-	Micro Trace Buffer
NCO-	numerically controlled oscillator
NDA-	Non-disclosure agreement
NMI-	Non-maskable Interrupt
NVIC-	Nested Vectored Interrupt Controller
NVRAM-	Non-Volatile Random-Access Memory
NVS-	Non-Volatile Storage
OBD-	On-Board Diagnostics
OTA-	Over The Air
PC-	personal computer
PCB-	Printed circuit board
PHY-	Physical layer
PMSA-	Protected Memory System Architecture
PMU-	Power Management Unit
PPB-	Private Peripheral Bus
PSA-	Platform Security Architecture
PWM-	pulse-width modulation
RAII-	Resource acquisition is initialization
RAM-	Rand Access Memory
RISC	reduced instruction set computer
ROM-	Read Only Memory
RTOS-	Real-Time Operating System
S-AHB-	System AHB
SAU-	Security Attribution Unit
SD-	Secure Digital (Memory Card)
SIMD-	Single Instruction, Multiple Data
SMD-	Surface Mount Devices
SPI-	Serial Peripheral Interface
SRAM-	Static RAM
SWC-	SoftWare Component
SWD-	Serial Wire Debug
SWO-	Serial Wire Output
TPA-	Trace Port Analyzer
TPIU-	Trace Port Interface Unit
TUI-	Text-based user interface
UART-	Universal Asynchronous Receiver-Transmitter
UDS-	Unified Diagnostic Services
UNICS-	Uniplexed Information and Computing System

UNIX-	Uniplexed Information and Computing System
UTF-	Unicode Transformation Format
V-	Volts
WFE-	Wait for Event
WFI-	Wait for Interrupt
WIC-	Wake-up Interrupt Controller
bash-	Bourne again shell
grep-	search Globally for lines matching the Regular Expression, and Print them
sed-	Stream EDitor
КЗ-	короткое замыкание
ЛУТ-	Лазерно-утюжная технология изготовления печатных плат
МК-	Микроконтроллер
ОКРы-	Опытно-конструкторские работы
ОС-	operating system
ОТК-	Отдел технического контроля
ПМ -	Программный Модуль
ПО-	Программное обеспечение
РПЦ-	Русская Православная Церковь
CCS-	Code Composer Studio
ТВЭЛ	ТеплоВыделяющий Элемент
ЦАП-	цифро-аналоговый преобразователь
ЧВК-	Частная военная компания
ЭВМ-	Электронная вычислительная машина

Глава 5

Определения

При чтении этого учебника Вам надо будет понимать набор определений. Я постарался дать Вам эти определения своими словами, на максимально понятном рабоче-крестьянском языке.

1. Сериализатор - это функция, которая берёт Си-структуру и возвращает её представление в виде текстовой строчки. Сериализаторы нужны для printf отладки кода.
2. Переменная окружения (environment variable) — это текстовая переменная, которая используются в различных командах и программных скриптах, выполняемых в операционной системе. Принципиально они работают точно так же, как переменные в языках программирования. Они представляют знакомые нам пары "ключ-значение" и используются для хранения параметров, настроек приложений, хранения ключей и других информационных данных. По сути это переменная определённая в операционной системе. Через переменные окружения мы будем передавать конфиги с вистемы сборки.
3. ToolChain- набор инструментов программирования. То есть программ для сборки других программ. Под ToolChain подразумевают текстовый редактор, систему сборки, компилятор, компоновщик, отладчики и прочее.
4. Микроконтроллер- программируемая микросхема, которая содержит в себе процессорное ядро, память, интерфейсы и прочие подсистемы. По сути, это компьютер в одной микросхеме. Можно образно сказать, что микроконтроллер - это интерпретатор прошивок.
5. Binding(и) - это функции, которые просто вызывают другие функции. В переводе на кухонный язык - программный клей. Binding(и) нужны, чтобы связать два совершенно разных API.
6. Артефакты (Artifact) в контексте программирования микроконтроллеров артефакты - это то полезное, что кристаллизируется на выходе tool chain(a). В самом общем случае это *.hex *.bin *.elf *.map

7. Сериализация типа данных- превращение бинарной структуры какого-либо типа данных в строку. Обычно это нужно для печати в UART содержимого структуры в человеко-читаемом виде.
8. Прерывание (Interrupt, Trap)- событие в микропроцессоре, которое провоцирует вызов отдельной функции ISR. После исполнения функции ISR управление возвращается обратно в прерванную функцию main. Как правило, прерывания используются для работы с периферийными устройствами: Timer, UART, DMA и пр.
9. Вектор прерывания — это пара чисел. Первое число - закреплённый за устройством номер, который идентифицирует соответствующий обработчик прерываний. Второе - адрес функции обработчика этого прерывания в физической памяти процессора.
10. Прошивка (firmware) - содержимое энергонезависимой памяти электронного устройства с микроконтроллером (*.bin). В прошивке всегда есть код, а иногда ещё образ файловой системы NVRAM, конфиги процессора. Монолитная прошивка может содержать еще и загрузчик. Под словом прошивка понимают не только бинарный файл, но и процесс исполнения на MCU. Порой можно услышать "прошивка зависла". Прошивка - это по сути один единственный процесс, который исполняется на одном единственном микропроцессоре. Она же и ядро и приложение.
11. Тыква (или кирпич) - электронное устройство с бракованной прошивкой, которое либо зависло, либо постоянно без конца перезагружается. Такое устройство можно восстановить только программатором.
12. PCB bring-up - итеративный процесс пуско-наладки новой электронной платы с производства, включающий в себя выявление неисправностей, починку, доводку, перепрошивку, конфигурацию, калибровку, тестирование и отладку устройства.
13. Системный вызов - (system call) — функция прикладной программы которая обращается к ядру операционной системы для выполнения какой-либо операции. Примеры системных вызовов: _sbrk(), _kill(), _getpid(), _write(), _close(), _fstat(), _isatty(), _lseek(), _read(). В программировании МК надо либо отключать системные вызовы, либо самим их реализовывать.
14. DevOps- это методология, направленная на автоматизацию процессов сборки, настройки и развёртывания программного обеспечения. Основная цель DevOps— повышение эффективности создания и обновления программных продуктов за счёт устранения барьеров между разработкой и эксплуатацией.
15. Модульный тест - функция, которая вызывает другую функцию с известными аргументами и проверяет наличие ожидаемого результата. В случае противоречия сигнализирует об ошибке.

16. Репозиторий- диск, где хранится исходный код программы и проекта. Репозиторий состоит из множества файлов и рабочающей над файлами системы контроля версий.
17. Боевая flash - диапазон flash памяти, в котором исполняется Generic прошивка микроконтроллера.
18. Загрузчик(Bootloader) - это отдельная прошивка, которая загружает другую прошивку. Обычно загрузчик стартует сразу после подачи питания перед запуском приложения. Это чисто системная часть кода.
19. CLI - Интерфейс командной строки. Это способ взаимодействовать с программой обменом текстами по принципу запрос-ответ.
20. ADT - Абстрактный тип данных (АТД). Набор данных (чисел) и алгоритмы над ними.
21. Пин- отдельный металлический проводник, который выступает из микросхемы.
22. Up-Time - переменная показывающая количество секунд бесперебойной работы с момента подачи на устройство электропитания. Up-time является показателем надежности. Чем Up-time выше, тем меньше рисков связано с использованием системы. Up-time измеряется в секундах.
23. Граф— это множество вершин и ребер (палочки и кружочки).
24. Подтяжка к GND- соединение пина и GND резистором.
25. Файл (File)- это именованный бинарный массив байтов в памяти. В качестве памяти может выступать RAM, ROM (Flash), FRAM, EEPROM, Flash-NAND в SD картах.
26. Подтяжка к VCC- соединение пина и VCC резистором 40kΩ
27. Конечный автомат - ориентированный граф. Вершины - это состояния, ребра - события
28. Ориентированный граф— граф у которого ребра имеют направление
29. Автомат Мили— это конечный автомат у которого действия происходят в момент переходов
30. Автомат Мура— это конечный автомат у которого действия происходят в состоянии
31. Байт (byte)- целое неотрицательное 8-битное число. От нуля до $0xFF = 255$
32. Слово (Word)- целое неотрицательное 16-битное число. От нуля до $0xFFFF = 65536$

33. Graceful degradation (изящная деградация/отказоустойчивость) - свойство системы продолжать выполнять свою основную функцию, даже если значительная часть системы вышла из строя. Если отказала одна подсистема, то остальные подсистемы должны продолжать работать . Это как кошка. Если она разучилась прыгать через обруч, она не должна разучиться есть, спать и мяукать.
34. Cross-компиляция- это когда на одной процессорной архитектуре собирают программу для абсолютно другой процессорной архитектуры. Например на x86 64 собирают прошивку для ARM Cortex-M33. Дело в том, что персональный компьютер- это по сути универсальный вычислитель и ему абсолютно всё равно какие программы и для чего собирать.
35. Токен— абстрактная строка символов без пробелов, служащая для компактной записи более длинного текста.
36. Чувствительность (Sensitivity) - отношение изменения выходного сигнала измерителя к вызывающему его изменение измеряемой величины.
37. Погрешность (Observational error) - отклонение измеренного значения величины от её истинного (действительного) значения. Единица измерений совпадает с измеряемой величиной.
38. Точность (precision) - отдельные измерения могут сильно отличаться друг от друга. Это показывает ширина кривой распределения. Степень такого разброса данных и называется точностью измерения. Единица измерения точности совпадает с размерностью измеряемой величины.
39. Активный уровень— это то напряжение, которое устанавливается на пине микроконтроллера при получении сигнала, например нажатии на кнопку. В зависимости от схемотехники это может быть 0В или 3.3В.
40. Тестирование (testing) - процесс проверки того, что программа удовлетворяет установленным требованиям. Проводится для обнаружения аномалий, для подтверждения того, что требования подходят в данном контексте и для создания уверенности в поведении программы.
41. Встраиваемое ПО (embedded software) - полностью объединенное ПО, которое исполняется на микроконтроллере.
42. Данные конфигурации (configuration data) - данные которые назначаются во время построения, которые управляют процессом сборки программных элементов. Это, например, макроопределения препроцессора, которые выбирают код для сборки. Это те же пресловутые XML файлы настройки IDE для выбора ToolChain(a) или инструментов сборки. Эти данные управляют построением ПО. Эти данные используются для выбора нужного кода из общей кодовой базы.

43. Данные калибровки (calibration data)- данные, которые будут применены как значения параметров программного обеспечения после построения артефактов в процессе разработки. Например смещение и чувствительность акселерометра. Калибровочные данные не содержать исполняемый или интерпретируемый код.
44. Connectivity - всё что связано с интерфейсами и протоколами. Часто отдельная папка в репозитории.
45. Таблица векторов прерываний - это таблица адресов функций обработчиков прерываний. Первая колонка это номер прерывания, вторая колонка абсолютный адрес обработчика прерываний в on-chip NOR-Flash памяти. Обычно в этой таблице сотни записей. У каждого микроконтроллера она своя. В бинарном файле перечислена вторая колонка как массив uint32 индекс массива номер прерывания, значение - адрес ISR. Чаще всего таблица векторов прерываний расположена в самом начале бинарного файла с прошивкой. Сразу после указателя на верхушку стековой RAM памяти.
46. ASIC- специализированная микросхема, которая выполняет только одну специфическую задачу аппаратно.
47. NVRAM - энергонезависимая память с произвольным доступом. По сути Key-Val-Map(ка). В ней могут храниться любые бинарные данные, ассоциированные со своим ID числом.
48. Сервер сборки- отдельная автономная программа, которая непрерывно только собирает другие программы согласно скрипту сборки и запускает модульные тесты.
49. Зомби - отдельный автономный NetTop PC, который непрерывно исполняет сервер сборки 24 часа в сутки и 7 дней в неделю.
50. Осциллограф- измерительный прибор, который показывает график сигнала от времени.
51. BOM-список всех компонентов (микросхем, разъёмов, резисторов, конденсаторов и пр) из которых собрана электронная плата.
52. Накатить ПО - обновить прошивку
53. Padding (набивка) - значения, которыми заполняют оставшиеся ячейки бинарных пакетов, например в CAN или блоках шифрования.
54. Patch (заплатка) - это дополнительные файлы или изменения существующих конфигурационных файлов компьютерной программы по умолчанию для того, чтобы открылась новая ценная функциональность или возможность у изначальной программы. Добавление к J-link новых файлов для поддержки других микроконтроллеров - это яркий пример патча.

55. "Патчить" программу - устанавливать к ней патч. Чаще всего патчи создают сами авторы программ - если это удобнее, чем переписывать программу полностью.
56. Метаданные — информация о другой информации, или данные, относящиеся к дополнительной информации о содержимом или объекте. Метаданные раскрывают сведения о признаках и свойствах, характеризующих какие-либо сущности, позволяющие автоматически искать и управлять ими в больших информационных потоках.
57. Библиотека — это сборник откомпилированного кода в виде классов или функций. Обычно библиотеки имеют расширение *.a.
58. bitbanding - это возможность получить доступ к определенным битам обращаясь ячейкам расположенным в специальных областях адресного пространства. Доступ может быть и на чтение и на изменение бита.
59. Микрокод (microcode) — программа, реализующая набор инструкций процессора. Так же, как одна инструкция языка высокого уровня преобразуется в серию машинных инструкций, в процессоре, использующем микрокод, каждая машинная инструкция реализуется в виде серии микроинструкций — микропрограммы, микрокода. Например именно микрокод сохраняет и восстанавливает регистры процессора при обработке прерываний. Или другой например, операция деления на RISC процессоре может быть сгенерирована компилятором как микрокод. Микрокод нужен, когда есть очень частое действие, и при этом нехватает инструкций процессора.

В целях ускорения производительности микрокод может быть реализован даже аппаратно в виде ОТР памяти прямо на кристалле процессора.

Глава 6

Настройка ToolChain-на для Разработки

6.1 Пролог

В этой главе Вы узнаете какой путь проходит си файл с момента написания до момента попадания во Flash память микроконтроллера.

6.2 Почему микроконтроллеры программируют на Си?

Какие есть варианты языков для программирования микроконтроллеров? Теоретически подойдет любой компилируемый язык программирования: Assembler, Си, С++, Rust, Pascal.

Однако выбирают обычно между Си и С++.

1. Язык Си это самый простой язык программирования. Язык Си простой как ножик. Одни только функции и переменные. Прост в освоении. Тут не будет виртуальных деструкторов, делегатов, шаблонов и прочего. Код выглядит как математические формулы, которые и так все видели в школе.
2. Си язык очень легко подается рефакторингу. Переименовывать функции, константы и переменные можно буквально утилитой sed. Прямо из командной строки внутри всего репозитория. При этом код по-прежнему будет собираться и проходить тесты.
3. У Си есть циклическое Legacy. Нужный вам код можно взять из ядра Linux или Zephyr.
4. Программировать на Си проще чем программировать на Assembler.
5. Высокая скорость разработки. В сравнении с Assembler программирование на Си позволяет повысить производительность.
6. У компилятора GCC стало очень много ключей для гибкой настройки различных видов предупреждений на тот или иной синтаксис и семантику.

7. Существует бесплатный компилятор GCC, Clang, TCC и прочие.
8. Это компилированный язык. Значит он будет исполняться быстрее, чем интерпретированный язык.
9. В Си есть указатели. Можно прямо из кода получить доступ к любой памяти.
10. Есть слово volatile. Это позволяет принудительно выграбать из физической памяти настоящие данные.
11. Вендоры микроконтроллеров дают MCAL именно на Си. Поэтому чтобы сэкономить время и использовать их код надо тоже продолжать писать на Си.
12. Достоинство языка Си в том, что он старый 50+ лет и поэтому появились очень зрелые компиляторы. В частности в GCC появилось много опций для выявления UB.

Си- это идеальный язык для программирования микроконтроллеров. Компромисс между сложностью и эффективностью.

6.3 Каков план?

1. Научиться собирать прошивки для микроконтроллера компилятором GCC. Причем собирать сорцы из make файлов.
2. Научиться прошивать *.hex файл с прошивкой во Flash память
3. Научиться отлаживать *.elf прошивку по шагам через интерфейс SWD.
4. Написать первую базовую, полноценную, тестировочную NoRTOS прошивку в которой будет работать драйвер SysTick, Interrupt, GPIO, UART, HardBeat LED, парсер CSV, CLI и модульные тесты, чтобы можно было при помощи CLI и встроенных тестов отлаживать любой другой функционал. То есть довести прошивку до ортодоксально-канонической формы.

В принципе это универсальный план действий для освоения абсолютно любого другого микроконтроллера. Надо просто выполнить эти 4 шага. Остальное уже зависит от специфики конкретного приложения. А эти 4 пункта, как правило всегда присутствуют в любой сборке. Это база.

6.4 Что надо из оборудования?

1. Учебно-тренировочная электронная плата .
2. Кабель переходник с USB-A на USB Type-C. Для соединения отладочной платы и РС

3. Программатор отладчик J-link V9 (или V12) USB-JTAG для ARM. Программатор для загрузки прошивки.
4. шлейф для соединения MCU и программатора

6.5 Что надо из документации?

Для программирования микроконтроллеров надо очень хорошо ориентироваться в множестве официальных документов.

1. Data Sheet. , Флаер на продукт.
2. Reference Manual. , Спецификация на микроконтроллер. Разметка памяти. Список прерываний.
3. GNU Make. 224 стр., Спека на GNU make.
4. The GNU linker. 122 стр., Спека на компоновщик.
5. Using the GNU Compiler Collection. 1006 стр., Спека на компилятор GCC.
6. Architecture Reference Manual, Спецификация ядра.

6.6 Что надо из ПО?

Вот список утилит с которыми скорее всего предстоит столкнуться при разработке прошивок .

1. JRE. Виртуальная машина Java для запуска Eclipse. Можно скачать на сайте Oracle.
2. Eclipse IDE for C/C++ Developers. Текстовый редактор для написания кода
3. Notepad++. Вспомогательный текстовый редактор специально для быстрых правок.
4. GNU Tools ARM Embedded. Он же arm-none-eabi-gcc.exe (GNU Arm Embedded Toolchain 10.3-2021.10) 10.3.1 20210824 (release) GCC компилятор языка программирования Си. Версия 10.3.1
5. GNU Make. Система сборки.
6. pdf reader. Браузер PDF файлов с документацией
7. WinMerge. Сравнение текстовых файлов с подсветкой
8. WinRAR. распаковка архивов с документацией от вендора
9. hexdump/hexedit. Просмотрщик бинарных файлов

10. grep. Утилита поиска подстрок в кодовой базе.
11. find. Поиск файла в файловой системе по регулярному выражению для его имени.
12. Cygwin. Набор Unix утилит для Windows
13. Tera Term. Терминал последовательного порта для полнодуплексного доступа к UART-CLI.

Все программы добавляем в переменную PATH, чтобы можно было их вызывать просто по имени.

6.7 Установка текстового редактора

Так как любая программа это прежде всего текст, то надо установить какой-нибудь текстовый редактор.

Предлагаю Eclipse IDE for C/C++ Developers так как у него весьма удобные HotKeys для ускорения написания кода. Плюс в Eclipse приятное синее выделение. Еще Eclipse мгновенно запускается.

Установить Eclipse IDE for C/C++ Developers можно отсюда
<https://www.eclipse.org/downloads/packages/>

Установка заключается в распаковке скаченного архива в удобное место. В ОС Windows можно устанавливать Eclipse в корень диска С.

Программа запускается с помощью исполняемого файла eclipse.exe, который находится в распакованной папке.

При первом запуске Eclipse надо выбрать рабочую папку (workspace). В этой папке Eclipse будет по умолчанию создавать проекты.

Настройки текстового редактора содержатся в файлах .project, .cproject. Важно снять галочку с пункта Makefile generation. В поле Build location прописать относительный путь к папке, которая содержит Makefile.

6.8 Установка GCC Cross-компилятора для ARM

Прежде всего нужен набор инструментов. Он же ToolChain:

1. препроцессор (cpp). Сначала в работу включается препроцессор и готовит полный текст программы.
2. компилятор ASM(as)
3. компилятор С (gcc) С выхода препроцессора программа передается компилятору, который формирует ассемблерные листинги из которых получаются объектные файлы (либо сразу объектные файлы, минуя этап выдачи ассемблерных листингов)

4. компилятор C++(g++)
5. компоновщик(ld) По заранее объявленным правилам, линковщик собирает из obj файлов выполняемый файл elf, который потом можно передать в программатор и залить на микроконтроллер.
6. отладчик(gdb)
7. binutils(ы) для диагностики полученных артефактов(nm, size, readelf)
8. архиватор статических библиотек(ar)
9. и прочее

Это основные утилиты, которые и делают всю работу по превращению исходников (*.c, *.h) в артефакты (*.hex *.bin *.map *.elf *.out файлы).

Поскольку микроконтроллеры YTM32 изготовлены с использованием процессорного ядра ARM Cortex-M33, одним из вариантов набора используемых инструментов является ARM GNU Toolchain, который сможет подготовить исполняемые файлы на x86-64 платформе для платформы Armv8-M. В составе тулчайна имеется все необходимое: компилятор, ассемблер, линковщик и целая куча других полезных утилит. Это называется cross-компиляция.

Cross-компиляция - это когда на одной процессорной архитектуре собирают программу для абсолютно другой процессорной архитектуры. Например на x86-64 собирают прошивку для ARM Cortex-M33. Дело в том, что персональный компьютер - это по сути универсальный вычислитель и ему абсолютно всё равно какие программы и для чего собирать.

ToolChain следует брать с официального сайта компаний ARM.

Toolchain устанавливается как обычная win программа прямо из файла gcc-arm-none-eabi-10.3-2021.10-win32.exe.

Почему компилятор так называется — gcc-arm-none-eabi? Все эти слова несут за собой вполне конкретный и определенный смысл:

1. gcc это название компилятора
2. arm целевая архитектура процессора, под которую будет производиться компиляция
3. none означает, что компилятор не вносит никакого дополнительного bootstrap-кода от себя
4. eabi сообщает, что код соответствует спецификации двоичного интерфейса EABI

В GCC ARM всего около 30 утилиты. Вот основные из них

Название утилиты	Назначение утилиты
addr2line	преобразует адрес из flash в строку кода в файле
cpp	Утилита вставки и замены текста
gcc	компилятор языка Си
ar	архиватор
as	ассемблер
elfedit	парсер elf файлов
gdb	отладочный клиент
ld	компоновщик
nm	показывает список символов в объектном файле
objcopy	преобразователь elf файлов в bin hex
objdump	показывает информацию про объектные файлы
size	показывает размер секций в бинарном файле
strings	показывает печатаемые строки из бинарного файла
strips	удаляет символы и секции из файла

Каждый свой шаг надо проверять. Как проверить, что терминал cmd находит ToolChain? Откройте cmd из любой папки и наберите arm-none-eabi-gcc.exe –v

Версия должна отобразиться в соответствии с версией скаченного дистрибутива. ToolChain может находится примерно по такому адресу:

C:/Program Files (x86)/GNU ARM Embedded Toolchain/10 2021.10/bin

6.9 Установка Build Tools binaries

Самый классический способ сборки программ на С(ях) это, конечно же, Make файлы. Не перестаю удивляться насколько элегантна сборка программ из make. В 1970е года когда появились утилита make программировали только настоящие ученые со степенями докторов наук. В 197x программировали по настоящему достойные люди. Поэтому и появились такие утилиты-шедевры как make, grep, find, sort и прочее. Благодаря Make файлам можно управлять модульностью сборок программных компонентов по-полней. Мгновенно, одной строчкой включать и исключать сотни файлов одновременно для сотен сборок.

Вот минимальный набор утилит для процессинга Make файлов.

Утилита	Назначение
make	build automation tool
sh	Unix shell interpreter
busybox	a software suite that provides several Unix utilities
echo	Echo the STRING(s) to standard output.
cp	Copy SOURCEs to DEST
mkdir	Create DIRECTORY
rm	Remove (unlink) FILEs

6.10 Скачивание SDK от вендора

Чтобы можно было начать разрабатывать какое-то приложение и бизнес логику нужно целая куча системного софта. Это базовый софт или System Software. В его состав входит всяческая инициализация. Инициализация тактирования(TIM, RCC, RTC), интерфейсов (CAN, SPI, I2S, I2C, UART, USB, GPIO, SDIO, SAI, Ethernet), системных компонент (FLASH, SRAM, DMA, MPU, IWDG, RNG, FPU, NVIC), возможностей генерации (DAC, PWM). Это десятки тысяч строк кода.

Компания ARM выпускает так называемый CMSIS (Common Microcontroller Software Interface Standard). Это С обертки для ASM команд под Cortex-Mx чипы. Так же *.ld файлы для компоновщика. Это скачивается с сайта arm после регистрации.

Исходники HAL можно взять с официального сайта производителя МК. Называется это обычно словом SDK. Надо только зарегистрироваться и получишь много бесплатного Си-кода. Причем сам код HAL очень даже хорошо написан.

1. `startup_gcc.S` - это код процедуры `Reset_Handler`, который отрабатывает до вызова функции `main()`. Он написан на ассемблере. Ассемблер - это такой язык программирования, где нет переменных. Вместо переменных регистры общего назначения (16 регистров процессора на все). Тут происходит отключение прерываний, обнуление регистров процессора, инициализация глобальных переменных, инициализация регистра указывающего на начало стековой RAM памяти, вызов функции `SystemInit` и вызов функции `main()`. Тут же можно посмотреть как называются функции обработчики прерываний, например `SysTick_Handler`.

Настоятельно рекомендую вам сразу определить все обработчики прерываний из файла `startup_gcc.S`, как `weak` функции. Их там много. Это избавит вас от неожиданного сваливания прошивки в `DefaultISR` по непонятным причинам. Сэкономите потом на отладке программ.

2. `linker/gcc/flash.ld` - это конфигурация для компоновщика. Тут указывается сколько у микроконтроллера RAM и ROM памяти, сколько стековой памяти, сколько памяти в куче. Указывается начало RAM памяти, начало ROM памяти. Указывается название массива `isr_vector`, который отвечает за таблицу векторов прерываний. Тут указано из каких секций состоит программа и в какой последовательности эти секции следуют. LD скрипт - это целый язык программирования со своими переменными, операторами, функциями и комментариями. Небольшой ликбез по LD можно посмотреть тут.

3. `startup MCU.c` - в сорце `system MCU.c` определена функция `SystemInit()`. Эта функция вызывается до функции `main()`. `SystemInit` включается сопрессор для вычислений в плавающей точке, отключает сторожевой таймер, включает модуль защиты памяти MPU и прочее. Тут же лежит функция для программной перезагрузки микроконтроллера `SystemSoftwareReset`.

Далее исходный код добавляется по мере потребности учитывая специфику проекта. На GitHub тоже находятся множество образцовых проектов прошивок для микроконтроллера.

6.11 Система сборки

В качестве системы сборки я по-прежнему использую make скрипты. Как же система сборки make узнает, что нужен именно этот toolchain? Ведь их может быть установлено несколько toolchain-ов в разные папки. Ответ прост. Надо прописать адрес toolchain в переменной Path. А старый путь удалить или поместить в конец.

6.12 Препроцессор

Препроцессор это консольная утилита, которая просто выполняет автозамены в тексте. Причем в любом тексте. Препроцессору абсолютно не важно с каким языком программирования он работает. Более того есть множество реализаций препроцессора.

1. cpp
2. m4

Однако в программировании прижился именно препроцессор cpp.

Его задача — выполнить все указанные в исходном коде директивы, т.е. специальные команды, которые препроцессор распознает и исполняет:

1. Удаление комментариев из кода;
2. Подключение файлов через директивы #include, #include_next;
3. Условное подключение/удаление фрагментов кода: #if, #ifdef, #ifndef, #else, #elif, #endif;
4. Вывод диагностических сообщений: #error, #warning, #line;
5. Передача инструкций компилятору: #pragma;
6. Определение макросов: #define;
7. Расстановка специальных маркеров, которые помогают передавать указания на конкретные строки (помогает указывать на строки в которых содержатся ошибки);
8. Прочие служебные функции.

Можно посмотреть результат работы препроцессора если передать ключ -E
cpp.exe main.c -E > main.pp

6.13 Компиляция

Компилятор это и есть программирующая программа. А всесь ваш Си код - это не что иное, как развернутый комментарий для программы компилятора.

Чтобы был скомпилирован корректный файл, компилятору нужно помимо самого кода указать еще и некоторые флаги, которые влияют на конечный результат. Рассмотрим эти аргументы командной строки для консольной утилиты компилятора gcc.

Итак, обо всём по-порядку...

1. Архитектура. (опции `-mcpu=xxx` и `-march=xxx`) Внутри каждого микроконтроллера заложен микропроцессор. Существует большое количество всяческих процессорных архитектур: 8051, KOMDIV-64, ARM7T, ARM7TDMI, ARM9, ARMv5, ARMv6, ARMV7M, AVR, AVR32, C166, Cortex-M0, Cortex-M23, Cortex-M3, Cortex-M33, Cortex-M4, Cortex-M4F, Cortex-R, e200z0, LEON3, MCS-51, Microblaze (DLX), MIPS, MIPS32, MIPS32 M4K, MK211, MSP430, NIOS, Nios II, PowerPC, RISC-V, RISCore32, RV32IMC, SCR1, SPARC, STM8, Xtensa, Z80.

Поэтому компилятору надо явно сообщить для какой целевой процессорной архитектуры мы собираемся собирать машинный код. Для этого существуют две опции компилятора: `-mcpu=` и `-march=`. Опция `-mcpu=` указывает семейство процессора. Для указания архитектуры опция `-march`. Вот типичные значения для этих ключей gcc.

микропроцессор	семейство	Архитектура
ARM Cortex-M0	<code>-mcpu=cortex-m0</code>	<code>-march=armv6-m</code>
ARM Cortex-M3	<code>-mcpu=cortex-m3</code>	<code>-march=armv7-m</code>
ARM Cortex-M4	<code>-mcpu=cortex-m4</code>	<code>-march=armv7e-m</code>
ARM Cortex-M7	<code>-mcpu=cortex-m7</code>	<code>-march=armv7e-m</code>
ARM Cortex-M33	<code>-mcpu=cortex-m33</code>	<code>-march=armv8-m.main</code>
Power-PC (spc58nn)	<code>-mcpu=e200z4</code>	<code>no-data</code>

2. Плотность кода(arm или thumb). В программировании микроконтроллеров ARM можно утрамбовать бинарный файл путем замены 32битных инструкций на 16битные инструкции. Это производится ключём `-mthumb`.
3. Floating Point Unit (FPU). (опция `-mfloat-abi=xxx`) Процессор это дискретное цифровое устройство. Одновременно с этим окружающий нас мир аналоговый. Все физические величины они являются действительными числами. То есть у них, как принято говорить, плавающая запятая. Поэтому в процессоры часто вмонтируют сопроцессоры для работы с плавающими числами. У простых процессоров этого нет. В таких случаях компилятор генерирует код для программной работы с плавающими числами. Это надо тоже явно указывать в пучке опций компилятору ключем `-mfloat-abi=` каким образом мы будем обрабатывать действительные числа.

Тут есть три варианта:

- (a) `-mfloat-abi=soft` Программная реализация. заставляет GCC генерировать вывод, содержащий библиотечные вызовы для операций с плавающей точкой
- (b) `-mfloat-abi=softfp` позволяет генерировать код с использованием аппаратных инструкций с плавающей точкой, но по-прежнему использует соглашения о вызовах с плавающей точкой
- (c) `-mfloat-abi=hard` позволяет генерировать инструкции с плавающей точкой и использует специфичные для FPU соглашения о вызовах.

Если не знаете, что выбрать для `float-abi` ставьте `-mfloat-abi=soft`. Просто программа будет медленнее работать.

В ARM Cortex-M0 процессорах нет аппаратного FPU, поэтому пишем опцию `-mfloat-abi=soft`.

Ещё можно активировать аппаратные инструкции для чисел с плавающей запятой ключём `-mhard-float`

4. Тип FPU (опция `-mfpu=xxx`). Если на предыдущем шаге вы выбрали `-mfloat-abi=hard`, то на следующем шаге надо указать какой именно FPU внутри вашего микроконтроллера. Такая инфа находится в спеке на чип. Чаще всего это аргументом будет `fpuv4-sp-d16`, `fpuv5-d16`, `fpuv5-sp-d16`.
5. Стандарт языка программирования. (опция `-std=xxx`) Сам язык программирования Си может быть разный. У Си как языка несколько версий. Чтобы компилятор правильно понял язык Си надо указать на какой стандарт диалекта надо ориентироваться. Варианты тут такие: `-std=c2x`, `-std=c17`, `-std=c11`, `-std=c99`, `-std=c90`, `-std=gnu89`, `-std=gnu90` `-std=gnu11`
6. Библиотеки. Библиотеки GNU ARM используют newlib для обеспечения стандартной реализации библиотек С. Чтобы уменьшить размер кода и сделать его независимым от аппаратного обеспечения, в микроконтроллерах используется облегченная версия newlib-nano. Однако newlib-nano не предоставляет реализацию низкоуровневых системных вызовов, которые используются стандартными библиотеками С, такими как `print()` или `scanf()`, но позволяет существенно сократить размер исполняемого файла. Соответственно чтобы использовать библиотеку newlib-nano и nosys нужно добавить следующее: `-specs=nano.specs` `-specs=nosys.specs`.
7. Предупреждения во время компиляции. Чтобы видеть потенциальные ошибки, необходимо включить выдачу всех предупреждений во время компиляции: `-Wall`.
8. Уровень отладки. Для того, чтобы включить отладку, нужно добавить флаг `-g`.

6.14 Компоновка

Компоновщик используется для того, чтобы соединить секции в нескольких объектных файлах и правильно разместить их в памяти.

Для этого необходимо написать скрипт линковки и описать в нем, как будет это все размещено в памяти. Давайте по шагам разберем как это делается. Для того, чтобы правильно составить скрипт, необходимо создать файл линковки linker.ld и начать вносить в него содержимое.

6.14.1 Какую выбрать libc?

При программировании на Си нужны стандартные функции, такие как printf() sprintf() и прочее. Они обычно распространяются в виде предварительно скомпилированных бинарных *.a файлов и лежат в папке с компилятором.

Называется эта библиотека libc. Их очень много реализаций: glibc, picolibc, nanolib, Newlib и прочее. От выбора той или иной libc зависит размер получившегося бинаря.

Собирая код внутри статической библиотеки libc.a у вас может возникнуть ошибка компоновщика про отсутствие определения тел функций для системных вызовов. Системные вызовы - это функции _sbrk, _write, _close, _fstat, _isatty, _lseek, _read, abort, _exit, _kill и _getpid. Именно эти функции и вызываются внутри libc.

Вам надо решить какую Вы выберете реализацию libc. libc - это статическая библиотека с реализацией таких функций как printf(), scanf(), snprintf() и прочее.

Реализация libc ключ компоновки

nosys	-specs=nosys.specs
newlib-nano	-specs=nano.specs
rdimon	-specs=rdimon.specs

Я пока выбрал rdimon.

6.14.2 Секции

Любая собранная си программа в obj и elf файле разбита на секции.

Листинг 6.1: показать размер каждой секции в бинарном файле

```
>arm-none-eabi-size -d -B main.o
    text      data      bss      dec      hex filename
    234        0        0     234      ea main.o

>arm-none-eabi-size -d -B ytm32b1m_evb_0144_rev_b_can_fd_gcc_m.elf
    text      data      bss      dec      hex filename
 188308     6984    61160   256452   3e9c4 ytm32b1m_evb_0144_rev_b_can_fd_gcc_m.elf

>arm-none-eabi-readelf.exe -S ytm32b1m_evb_0144_rev_b_can_fd_gcc_m.elf
There are 25 section headers, starting at offset 0x1f347c:
```

Section Headers:

[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[0]		NULL	00000000	000000	000000	00		0	0	0
[1]	.interrupts	PROGBITS	00000000	010000	000340	00	A	0	0	4
[2]	.text	PROGBITS	00000340	010340	02dc4c	00	AX	0	0	8
[3]	.ARM	ARM_EXIDX	0002df8c	03df8c	000008	00	AL	2	0	4
[4]	.interrupts_ram	NOBITS	1fff0000	050000	000400	00	WA	0	0	1
[5]	.data	PROGBITS	1fff0400	040400	001b3c	00	WA	0	0	8
[6]	.code	PROGBITS	1fff1f3c	041f48	000000	00	W	0	0	1
[7]	.array	INIT_ARRAY	1fff1f3c	041f3c	00000c	04	WA	0	0	4
[8]	.bss	NOBITS	1fff1f48	041f48	001ae4	00	WA	0	0	8
[9]	.heap	NOBITS	1fff3a2c	041f48	00c004	00	WA	0	0	1
[10]	.stack	NOBITS	2000f000	041f48	001000	00	WA	0	0	1
[11]	.ARM.attributes	ARM_ATTRIBUTES	00000000	041f48	000034	00		0	0	1
[12]	.debug_info	PROGBITS	00000000	041f7c	049aa7	00		0	0	1
[13]	.debug_abbrev	PROGBITS	00000000	08ba23	00c1db	00		0	0	1
[14]	.debug_loc	PROGBITS	00000000	097bfe	01c928	00		0	0	1
[15]	.debug_aranges	PROGBITS	00000000	0b4528	003838	00		0	0	8
[16]	.debug_ranges	PROGBITS	00000000	0b7d60	0030b8	00		0	0	1
[17]	.debug_macro	PROGBITS	00000000	0bae18	02b31f	00		0	0	1
[18]	.debug_line	PROGBITS	00000000	0e6137	0729e3	00		0	0	1
[19]	.debug_str	PROGBITS	00000000	158b1a	077736	01	MS	0	0	1
[20]	.comment	PROGBITS	00000000	1d0250	000049	01	MS	0	0	1
[21]	.debug_frame	PROGBITS	00000000	1d029c	00f6c4	00		0	0	4
[22]	.symtab	SYMTAB	00000000	1df960	00d690	10		23	2426	4
[23]	.strtab	STRTAB	00000000	1ecff0	006397	00		0	0	1
[24]	.shstrtab	STRTAB	00000000	1f3387	0000f3	00		0	0	1

Key to Flags:

W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
L (link order), O (extra OS processing required), G (group), T (TLS),
C (compressed), x (unknown), o (OS specific), E (exclude),
y (purecode), p (processor specific)

Каждая из секций имеет собственное уникальное имя и набор атрибутов, определяющих целый ряд параметров:

1. bss (Block Started by Symbol) Неинициализированные переменные. Некоторые переменные не имеют значения на старте и нет необходимости сохранять их значения — под них нужно лишь зарезервировать память. Обычно эти переменные просто зануляются в start up коде ещё до вызова функции main.
2. text Код и данные. Секция содержит код выполняемых инструкций, которые будут располагаться во Flash памяти и константные значения. Секция .text содержит реальные машинные инструкции, которые составляют Вашу программу.
3. data Инициализированные переменные. Эта секция содержит статические инициализированные данные, которые были определены в Вашем коде. Она содержит глобальные переменные, которые проинициализированы на старте и при запуске программы будут перенесены в RAM.
4. rodata Данные только для чтения. Содержит переменные с постоянным значением, которые будут сложены во Flash-памяти. Например, в этой секции будет

сохранено значение переменной const uint32_t DELAY_MAX = 0x7A120;

5. comment Содержит информацию о версии компилятора.
6. ARM.attributes Содержит служебные сведения, которые в конечной прошивке не используются.

По сути сумма секций data и bss показывают требования к необходимому количеству оперативной памяти. А секция text показывает требования к Flash памяти.

Помимо секций, в объектном файле есть еще одна важная сущность: таблица символов. Это своего рода хеш: имя — адрес (и дополнительные атрибуты). В таблице символов, например, указаны все используемые функции и их адреса (которые будут указывать куда-то в секцию .text).

Листинг 6.2: так можно посмотреть версию компилятора

```
arm-none-eabi-objdump *.o -j .comment -s
```

Листинг 6.3: так можно посмотреть атрибуты компилятора

```
arm-none-eabi-readelf *.o -A
```

Листинг 6.4: так можно посмотреть символы данных

```
arm-none-eabi-objdump *.o -j .data -s
```

Листинг 6.5: так можно посмотреть символы методо

```
arm-none-eabi-objdump *.o -j .text -t
```

6.14.3 Таблица векторов прерываний

Таблица векторов прерываний - это таблица адресов функций обработчиков прерываний. Первая колонка это номер прерывания, вторая колонка абсолютный адрес бработчика прерываний в on-chip NOR-Flash памяти. Обычно в этой таблице сотни записей. У каждого микроконтроллера она своя. В бинарном файле перечислена вторая колонка как массив uint32 индекс массива-номер прерывания, значение - адрес ISR. Чаще всего таблица векторов прерываний расположена в самом начале бинарного файла с прошивкой. Сразу после указателя на верхушку стековой RAM памяти. Однако иногда таблица векторов прерываний хранится в RAM памяти и заполняется во время исполнения программы. Главное чтобы процессорное ядро знало, где она расположена. Это прописывается в спец регистрах ядра микроконтроллера.

Информацию о прерываниях можно найти в Reference Manual на используемый микроконтроллер, в разделе в котором описаны прерывания.

6.15 Настройка пошаговой отладки.

Классическая ситуация: накатили прошивку и плата зависла. Не мигает heartbeat LED. Как вариант может помочь пошаговая отладка.

Чтобы работала пошаговая отладка вам надо две утилиты.

1. Отладочный сервер.
2. Отладочный клиент.

Отладочный сервер и отладочный клиент обмениваются данными через сокеты.

6.15.1 Отладочный сервер

Отладочный сервер это утилита, которая непосредственно взаимодействует с микроконтроллером через программатор. Наружу отладочный сервер создает TCP порт для работы с отладочным клиентом. На вход сервер получает конфигурационный файл с настройками аппаратуры. Выбор интерфейса: SWD или JTAG. Выбор битовой скорости. Выбор микроконтроллера. Существует множество различных отладочных клиентов: OpenOCD, Ozone, ST-LINK_gdbserver, JLinkGDBServer, VisualGDB.

6.15.2 Отладочный клиент

Отладочный клиент это консольная утилита, которая бедется из средств разработки GCC. Называется эта утилита arm-none-eabi-gdb.exe. Отладочный клиент выступает как front-end для работы с программистом. На вход клиент получает *.elf файл. Показывает кусок кода и курсор показывающий на какой строке сейчас исполняется код.

6.16 Блок-схема ToolChain-a

Для нас исходниками являются файлы с расширениями *.ld, *.c, *.h, *.S, *.project, *.project, *.Makefile, *.mk, *.bat - их и надо подвергать версионному контролю в GIT.

Автогенеренными для нас являются файлы с расширениями *.o, *.bin, *.hex, *.map, *.elf, *.out, *.d, *.a. Отгружать следует файлы *.bin, *.hex, *.map, *.elf. Это артефакты или конечный продукт работы программиста микроконтроллера.

6.17 Последовательность пуска микроконтроллера

Итак, вы подали на микроконтроллер электропитание. Что же станет происходить и в какой последовательности? В этом процессе главная роль у таблицы

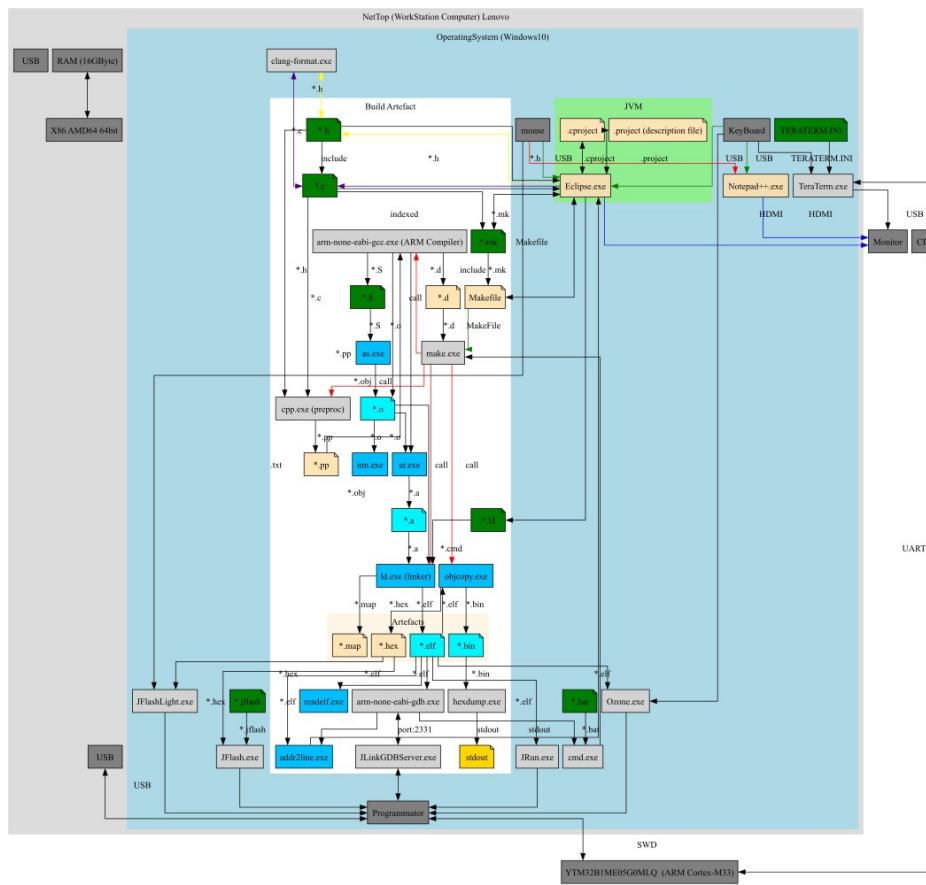


Рис. 6.1: Блок-схема ToolChain-a

векторов прерываний (`__isr_vector`). Тут надо сразу упомянуть, что прежде, чем вызовется всем известная функция `main()` произойдет длинная череда событий, настоящая закулисная суэта. Подковёрные игры. Но обо всём по порядку...

- Вы подали питание на микроконтроллер.
- Электрическая цепь под названием brown-out detector ждёт пока установится напряжение электропитания до уровня срабатывания цифровой CMOS логики.
- Как только напряжение достигло нужного уровня brown-out detector генерирует отрицательный перепад на проводе RESET.
- Процессор стартует от внутреннего неточного RC генератора тактирующих сигналов.
- Запускается заводской загрузчик с адреса 0x0000_0000. Это загрузчик написанный производителем микроконтроллера.
- В регистр счетчика инструкций PC процессор прописывает адрес 0x4. Это адрес на Reset_Handler. PC прыгает в Reset handler. Запускается процедура (`resetISR`). Это самая первая инструкция.

7. ResetISR в регистр R13(SP) Stack pointer присваивает значение указателя на начало стек памяти (initialize the stack pointer). Обнуляется бит ASP. В регистр программный счетчик R15 (PC) Program counter присваивается значение 0.
8. ResetISR обнуляет первые 12 регистров общего назначения данного микропроцессора.
9. ResetISR обнуляет всю RAM память (необязательно).
10. ResetISR обнуляет часть та часть RAM памяти, которая называется BSS. Там как раз хранятся неинициализированные глобальные переменные и локальные статические переменные.
11. ResetISR прописывает в RAM памяти значения глобальных переменных для которых в программе были определены ненулевые значения. Сами значения берутся из ROM памяти (flash).
12. ResetISR производит раскраску стека. То есть происходит заполнение области стека значениями 0xCC. Это позволит далее в run-time проверить, как далеко продвинулся стек по мере работы программы.
13. ResetISR отключает сторожевой таймер.
14. ResetISR вызывает функцию SystemInit()
15. SystemInit включает сопроцессор отвечающий за модуль вычисления с действительными числами (FPU).
16. SystemInit прописывает адрес начала таблицы векторов прерываний для данной прошивки в регистр процессора SCB->VTOR.
17. SystemInit выполняет починки согласно errata sheet.
18. ResetISR вызывает инициализацию глобальных переменных из библиотеке libc. Вызывается функция __libc_init_array.
19. В случае сборок на C++ ResetISR вызывает функции конструкторы классов.
20. ResetISR иногда копирует таблицу векторов прерываний ещё и в RAM память.
21. ResetISR разрешает маскированные прерывания.
22. ResetISR вызывает функцию main().
23. В случае выхода из функции main() ResetISR прыгает в бесконечный цикл и прошивка зависает.
24. Заводской загрузчик смотрит на два boot пина и решает откуда загружать прошивку. Тут обычно три варианта: (RAM, ROM, UART)

25. Запускается первичный загрузчик.
26. Первичный загрузчик запускает вторичный загрузчик.
27. Вторичный загрузчик запускает generic приложение.
28. и так далее...

Стоит заметить, что функция ResetISR есть абсолютно в любой прошивке. Будь, то заводской загрузчик от вендора MK, ваш первичный загрузчик mbr, ваш вторичный загрузчик или вообще generic приложение. ResetISR - это самая первая функция адрес которой хранится в таблице векторов прерываний. А таблица векторов прерываний это начало абсолютно любой прошивки для ARM процессоров. ResetISR как правило пишут на языке ассемблер, поэтому эту функцию можно найти в файле с расширением *.S . Обычно ResetISR у всех на 90 процентов одинаковый. Отличия косметические и их обычно прячут в сишиную функцию SystemInit(). Разная у всех прошивок будет именно функция main().

Вот тут-то в main и будет происходить настройка всех остальных подсистем микроконтроллера. Запуск PLL, настройка GPIO, аппаратных таймеров, контроллера FLASH памяти и прочее, и прочее по мере специфики приложения на микроконтроллера.

После инициализации всех подсистем SoCa прошивка либо начинает прокручивать суперцикл, либо пускает на исполнение планировщик какой-нибудь RTOS. И понеслась...

Только после этого начнёт работать та самая программная бизнес логика ради которой и была вся эта разработка.

6.18 Обработка прерываний на микроконтроллерах

Прерывания — это события в процессоре или вовне, которые вызываются периферийными устройствами или внешними прерываниями, такими как таймеры, GPIO, UART, I2C и т. д. Переполнился аппаратный таймер - сработало одно прерывание. Из проводов по UART прибыл байт - сработало другое прерывание.

Массив всех возможных прерываний перечислен в так называемой таблице векторов прерываний.

По умолчанию таблица векторов размещается в начале пространства памяти, но может быть перемещена в другое место, например в RAM, загрузчиком или пользовательским программным обеспечением.

Однако по аналогии со внешними прерываниями могут быть и внутренние прерывания. Их принято называть исключениями (exceptions). Типичный пример исключения это HardFault, SVCall , BusFault, PendSV , SysTick и DebugMon . Все те прерывания, что имеют отрицательный номер прерывания - это и есть исключения.

Подготовка к прерываниям.

1. Необходимо прописать адрес начала таблицы векторов прерываний в регистр процессора VTOR. Обычно это строчка SCB->VTOR = 0x08000000;
2. разрешить глобально прерывания в регистре процессора. Это функция `__enable_irq`
3. Определить функции обработчики прерываний.
4. надо разрешить конкретные номера маскированных прерываний. Это функция `NVIC_EnableIRQ(SysTick_IRQn);`
5. установить приоритеты прерываний.

И вот возникло прерывание. Например внешнее прерывание от нажатия на кнопку. Как же происходит вход в прерывание (Interrupt Service Routine)?

1. Запоминается состояние, в котором находилось процессорное ядро перед переходом в режим обработки прерываний
2. Контроллер NVIC переключает ядро в режим обработки прерывания.
3. Сохраняются регистры процессора в стековой RAM памяти.
4. Микропроцессор помещает набор регистров в стек. В стек сохраняются регистры R0 – R3 и R12. Эти регистры используются в инструкциях для передачи параметров, поэтому, помещение в стек делает возможным их использование в функции обработки прерывания, а R12 часто выступает в роли рабочего регистра программы.

К числу помещаемых в стек данных относятся регистр статуса программы Program Status Register (PSR), счетчик программы Program Counter (PC) и регистр связи Link Register (LR). Регистры R0–R3 аргументы прерыванию R12 Эта операция выполняется с помощью специального микрокода, что упрощает прикладной код.

5. Процессор определяет адрес функции обработчика прерывания согласно таблице векторов прерываний. При возникновении прерывания процессор определяет начальный адрес ISR из таблицы векторов автоматически.
6. Процессор записывает в счетчик команд адрес соответствующего вектора прерывания в соответствии с таблицей векторов прерываний.
7. Процессор переходит на исполнение кода по PC, в котором уже прописан указатель на функцию обработчик прерывания.
8. Некоторые программисты считают хорошей практикой внутри некоторых обработчиков прерываний временно отключать прерывания глобально, чтобы не произошло нежелательного вложенного вызова другого прерывания.
9. ISR Сбрасывает pending флаг для данного прерывания. Иначе прерывание будет происходить снова и снова.

10. Функция ISR быстро выставляет какие-то флаги.
11. Процессор закончил выполнять функцию обработчик прерываний.

В ARM Cortex-M NVIC реагирует на пуск прерывания с задержкой (latency) всего лишь 12 тактов процессора. В них входит выполнение микрокода, который автоматически помещает набор регистров в стек.

После отработки прерывания надо вернуться обратно в функцию main(). По завершении обработки прерывания все действия выполняются в обратном порядке. Возврат из прерывания в функцию main.

1. Микрокодом извлекается содержимое стека и восстанавливаются аппаратные регистры процессора для функции main из основной стековой памяти. Параллельно с этим, осуществляется выборка адреса возврата.
2. Программному счетчику PC присваивается значение LR взятое из стека.
3. Продолжение исполнения функции main()

Таким образом, процедура в main даже не догадывается, что она была прервана во время исполнения.

6.19 UART-CLI Для отладки

Любая разработка начинается там, где есть возможность наблюдать за тем, что получилось после изменений. Поэтому вам понадобится интерфейс командной строки CLI для отладки софта и железа. Крайне желательно использовать раскраску логов, чтобы концентрировать внимание на ошибках (красный) и предупреждениях (желтый). Плюс появление цвета в консоли это признак того, что поток символов в UART непрерывный так как коды цветов распарсились корректно. Своего рода, эрзац контрольной суммы для принятых данных. Просматривать UART-CLI логи можно в бесплатной утилите TeraTerm или PuTTY. Они точно понимают символы цветов.

Поэтому пришлось активировать свободный UART, написать в прошивке интерпретатор команд CLI для RunTime отладки по интерфейсу UART.

Это открывает прямую дорогу для полноценной отладки абсолютно любой остальной подсистемы микроконтроллера: CAN, LIN, PWM и проч.

Также в UART-CLI можно попросить микроконтроллер прогнать модульные тесты. Вот так:

Тесты исполняются прямо на target устройстве и собираются вместе с бизнес логикой в одном монолитном *.hex файле.

Теперь, вы знаете, как собирать код и Вы знаете, как проверять бинарь. Так цепь и замкнулась. Это и есть, так называемая, V-модель разработки. При разработке двигаемся от общего к частному. А при проверке двигаемся от частного к общему.

Num	Acronym	CommandName	Description
1	ce	core_exception	CoreException
2	tst	core_stitch	CoreStitch RAM
3	reboot	soft_reboot	Reboot board
4	cd	core_diag	Cortex M4 diag
5	ti	task_info	Task info
6	tcr	task_ctrl	TaskControl

Рис. 6.2: UART-CLI в TeraTerm

Рис. 6.3: микроконтроллер прогнал модульные тесты

6.20 Итог

Удалось получить базовое представление о сборке программ для микроконтроллеров.

Теперь понятно, как собирать код, прошивать бинарь, как выполнять пошаговую отладку прошивки на target устройстве и как запустить UART-CLI. Всё это открывает зелёную улицу к полноценной разработке на микроконтроллерах.

Все микроконтроллеры программируются одинаково, если собирать код из make файлов. Это и есть основное достоинство системы сборки GNU Make.

6.21 Ссылки

1. MCU on Eclipse
2. Процесс компиляции и сборки прошивки
3. Ozone - The J-Link Debugger

4. Interrupt Exception handling With ARM Cortex-M
5. ARM: Тонкости компиляции и компоновщик
6. AVR GCC: секции памяти
7. обзор вспомогательных утилит из GCC toolchain

Глава 7

Атрибуты Хорошой Прошивки (Firmware)

"Сделайте мне хорошо!"
(руководитель на планёрке)

В этой главе я бы хотел перечислить и обсудить некоторые общие, системные, поведенческие атрибуты хорошего firmware (прошивки) для микроконтроллерных проектов, которые не зависят от конкретного приложения или проекта. Некоторые атрибуты могут показаться Вам очевидными, однако по издёвке судьбы в 9 из 10 российских embedded компаний, к сожалению, нет и не знают ни одного из перечисленных атрибутов. Вот такие пирожки с капустой... Понимаете?

7.1 Микросекундный UpTime счетчик (камертон)

В хорошей прошивке должен быть точный аппаратный микросекундный up-time счётчик. Это для программных компонентов которые используют время. Например TimeStamp(ы) для логирования, limiter, планировщик, функции выдержки пауз, load-detect. Реализовать этот непрерывный счетчик можно на SysTick таймере или на одном из многочисленных аппаратных таймеров.

7.2 Сторожевой таймер (сторожевой пёс)

Прошивка может зависнуть при некорректных входных данных или в результате стресс тестирования. Сторожевой таймер позволяет автоматически перезагрузиться и устройство не останется тыквой.

7.3 Загрузчик (BootLoader)

Программатор есть далеко не всегда. Программатор часто не видит микроконтроллер из-за статического электричества или из-за длинного шлейфа. Зачастую программатор присутствует лишь в одном единственном экземпляре на всю

компанию в целом здании. Загрузка программатором - это чисто developer-ская прерогатива. У customer-ов нет и никогда не будет отладчика и особого шлейфа для данной PCB. А загрузчик по UART позволит записать новый артефакт на дешевом переходнике USB-UART. Также загрузчик позволит наладить DevOps и авто-тесты внутри компании. В идеале загрузчик должен уметь загрузить бинарь по всем доступным интерфейсам, которые только есть на электронной плате (PCB): USB, UART, RS485, RS232, CAN, LIN, BLE, 100 Base-TX, WiFi, LoRa и пр.

7.4 NVRAM (числохранилище)

Энергонезависимая Key-Value Map(ка) или NVRAM, FlashFs. Есть десятки способов ее реализовать. Это простая файловая система для хранения многочисленных параметров: калибровочные коэффициенты, ключи шифрования, IP адреса, TCP порты, счетчик загрузок, наработки на отказ, настройки трансиверов, серийные номера и многое другое. В моей нынешней прошивке уже 60 параметров. Это позволит не настраивать устройство заново каждый раз после пропадания питания и не плодить зоопарк прошивок с разными параметрами. Устройство всегда можно будет до-программировать уже в run-time(e) просто исполнив несколько команд в CLI. Также FlashFS позволяет передать сообщение от приложения загрузчику и наоборот.

7.5 Модульные тесты (скрепы)

Тесты позволяют делать безопасное перестроение упрощение кода, локализовать причины сбоев. Тесты выступают как документация к коду. Код тестов должен быть встроен прямо внутрь кода прошивки. По крайней мере для Debug сборок. Тесты можно запускать как при старте питания так и по команде из CLI.

7.6 Health Monitor (медбррат)

Это отдельная задача, поток или периодическая функция, которая периодически проверяет все компоненты, счетчики ошибок и в случае, если возникли какие-то сбои, HM как-то сообщает об этом пользователю. HM должен работать непрерывно. Например при ошибке посылает красный текст в UART-CLI а при предупреждении - желтый текст. Health монитор повысит надежность изделия в целом и позволит найти ошибки, которые пропустили модульные тесты.

7.7 Command Line Interface (CLI-шка)

CLI - это наверное самое полезное. CLI это само собой разумеющееся и в любой прошивке. Как её только не называют: CUI, TUI, командный интерфейс,

shell, bash-подобная консоль, Real Time Terminal (RTT), просто терминал, printf отладка, CLI, консоль, UART Debug Termianl. Много названий - это уже намёк на супер полезность shell-a. Все серьёзные программисты микроконтроллеров приходят к необходимости CLI с разных сторон. Однако суть одна. Это текстовый интерфейс командной строки поверх UART(реже UDP, TCP, RS232). Подойдет любой Full или Half Duplex интерфейс. Нужно это чтобы общаться с устройством на человеческом языке. Запрос-ответ. Как в оболочке Linux, только в случае микроконтроллера.

С помощью CLI можно запускать модульные тесты софта и железа, просматривать куски памяти, отображать диагностические страницы, управлять GPIO, пульять пакеты в SPI, UART, I2C, 1-Wire, I2S, SDIO, CAN, испускать PWM(ки), включать или выключать реле. В общем CLI необходима для тотального управления функционалом гаджета. Стоит заметить, что CLI это, к слову, единственный способ отлаживать прошивки в микроконтроллерах без JTAG, например в микроконтроллерах AVR, ESP32 и прочие.

Да и отладка по JTAG это тоже так себе подход. Ведь любая точка останова нарушает тайминги и с пошаговой отладкой Вы отлаживаете уже не ту программу, что будет работать в реальности. При этом устройство на другом конце CAN шины никто в пошаговую отладку не переключал. Только с CLI можно делать Non Invasive Debug. Когда есть CLI(шка), команды установки уровней логирования и чтения памяти по адресу, то отпадает даже как таковая необходимость в пошаговой отладке по JTAG. Вернее JTAG или SWD понадобится только для отладки самого UART и запуска CLI. А далее достаточно отладки только через CLI.

С помощью CLI Вы сможете допрограммировать поведение устройства уже после записи самой прошивки во Flash. Получится эдакий интерпретатор типа python. Вы сможете на плате выполнять разнообразные скрипты по мере необходимости в тестировании. Можно хранить CLI-скрипты на SD карте и запускать их подобно bat файлам. Красота несусветная! Как OS устанавливает и запускает утилиты, просто подключившись к UART через TeraTerm, Putty, HTerm, Hercules и отправив несколько команд. Вот и выходит, что CLI выступает как интерпретатор команд. А без CLI Вам бы пришлось варить еще кучу сборок с какой-то специфической функцией на 1 раз. Например, схемотехник попросил Вас сварить отдельную прошивку, чтобы мигать GPIO раз в секунду. А потом поддерживать на плаву этот зоопарк проектов. Оно Вам надо? А с CLI Вы просто возьмете релизную прошивку и выполните в CLI одну строчку и GPIO будет мигать как надо и где надо. Easy!

Посмотрите какая классная CLIшка у российского Flipper Zero, нет только раскраски логов в красный, желтый, синий, зелёный, розовый.

Посмотрите какая классная CLI-шка у швейцарского U-Blox ODIN C099-F9P

Засените какая классная CLI-шка у китайского STM32 устройства NanoVNA V2

Zephyr RTOS из коробки со своей собственной CLI. Даже авторитетные авторы всеми любимого FreeRTOS, сами рекомендуют использовать CLI. Это как?

COM6 - PuTTY



Welcome to Flipper Zero Command Line Interface!
Read Manual <https://docs.flipperzero.one>

Firmware version: HEAD 0.26.1 (b629ael built on 17-08-2021)

>: |

Рис. 7.1: CLI y Flipper Zero

```
VT COM27:460800bps - Tera Term VT
File Edit Setup Control Window Help
wifi_getch wifi_setch wifi_setssid wifi_getssid blink_led bt_bond bt_getmac bt_getname bt_inquiry bt_sppcli bt_stream bt_visible mem_erase mem_store input_pin output_pin help print_version set_mux
[INFO] Waiting for user input ...
~$
```

Рис. 7.2: CLI y U-Blox ODIN C099-F9P

Очевидно же, что оглушительный успех этих электронных продуктов в значительной мере определен наличием удобной и развитой CLI.

А те, у кого не было CLI, те просто запутались в сложности своего жё функционала, не поняли, что работает, а что нет и ушли в штопр.

```

COM22 - Tera Term VT
File Edit Setup Control Window Help

ch>
ch>
ch>
ch> help
There are all commands
help:           lists all the registered commands
reset:          usage: reset
cwfreq:         usage: cwfreq {frequency(Hz)}
saveconfig:     usage: saveconfig
clearconfig:    usage: clearconfig {protection key}
data:           usage: data [array]
frequencies:   usage: frequencies
scan:           usage: scan {start(Hz)} {stop(Hz)} [points] [outmask]
sweep:          usage: sweep [start(Hz)] [stop(Hz)] [points]
touchcal:       usage: touchcal
touchtest:      usage: touchtest
pause:          usage: pause
resume:         usage: resume
cal:            usage: cal [load|open|short|thru|done|reset|on|off]
save:           usage: save {id}
recall:         usage: recall {id}
trace:          usage: trace [0|1|2|3|all] [{format}|scale|refpos|channel|off] [{value}]
marker:         usage: marker [1|2|3|4] [on|off|{index}]
edelay:         usage: edelay {value}
pwm:            usage: pwm {0.0-1.0}
beep:           usage: beep on/off
lcd:            usage: lcd X Y WIDTH HEIGHT FFFF
capture:        usage: capture
version:        usage: Show NanoUNA version
info:           usage: NanoUNA-F V2 info
SN:             usage: NanoUNA-F V2 Serial Number

```

Рис. 7.3: CLI y NanoVNA V2

The screenshot shows the FreeRTOS website's navigation bar with links for KERNEL, LIBRARIES (which is underlined), SUPPORT, PARTNERS, and COMMUNITY. A green button on the right says "Download FreeRTOS". Below the navigation, there's a sidebar titled "LIBRARIES" with links to Home, LTS Libraries, Library Categories, FreeRTOS Plus (which is expanded to show Overview, FreeRTOS+TCP, FreeRTOS+IO, and FreeRTOS+CLI), and an Introduction link. The main content area has a breadcrumb trail: Libraries > FreeRTOS Plus > FreeRTOS+CLI. It features a title "FreeRTOS+CLI" and subtitle "An Extensible Command Line Interface Framework". Under the title, there's an "Introduction" section with a paragraph about the framework's purpose and a "click each stage in the process individually" note.

Рис. 7.4: CLI y FreeRTOS

7.8 Диагностика

У каждого компонента есть внутренние состояния: Black-Box Recorder, режимы микросхем, драйверов, какие-то конкретные переменные: up-time счетчики, дата, время сборки артефакта, версия, ветка, последний коммит. Всё это надо просматривать через CLIшку. Для этого и нужна подробная диагностика. Без

CLI диагностики прошивок получилось бы то же самое, если у медицины отнять МРТ, УЗИ, рентгеновские аппараты и даже термометры.

Прошивка без диагностики это как автомобиль без приборной панели, зеркал, с грязным лобовым стеклом без дворников. Пробовали на таком ездить?

7.9 Black-Box Recorder (Чёрный ящик)

У этого программного компонента тоже много имен: черный ящик, LogBook, BlackBox. Суть в том, что прошивка может записывать логи не только в UART (который может никто не смотреть), но и в NOR-Flash или лучше SD карту. Если организовать циклический массив-строк на N-записей, то можно записывать, например, последние M минут работы. Лог сообщения с TimeStamp(ами). А затем можно делать post-processing логов для расследования инцидентов. В авиации же принято вставлять черные ящики в самолёты. Значит их надо делать и в остальных приборах и агрегатах. Надо использовать полезные аналогии из других областей техники.

7.10 Переход в энергосбережение

Как известно, чем выше частота, тем больше энергопотребление. Энергопотребление растёт очень быстро: пропорционально 4-ой степени от частоты тактирования. Очевидно, что чтобы уменьшить энергопотребление надо как то уменьшить частоту тактирования микропроцессорного ядра. Как это сделать? Ответ прост. Можно по команде из UART-CLI понизить частоту системной шины до минимума. Отключить тактирование от ненужной аппаратной периферии и, таким образом микроконтроллер войдет в режим низкого энергопотребления. Это особенно важно если устройство работает от батареи.

Работать это может так. По команде из UART-CLI поступает желаемая частота ядра. Значение желаемой частоты прописывается в NVRAM. Затем происходит процедура перезагрузка. В инициализации PLL драйвер читает желаемую частоту из NVRAM, решает Диофантово уравнение , получает коэффициенты PLL. Обычно это два или три натуральных числа. Эти коэффициенты прописываются в регистры PLL. Вот так просто и не затейливо можно в режиме исполнения firmware без перепрошивки управлять энергосбережением всей электронной платы.

7.11 Аутентификация бинаря (optional)

Рано или поздно придется защищать гаджет от того чтобы на него не накатили чужеродный софт и не превратили его в BotNet. Можно поставить внешний сторожевой таймер и он будет сбрасывать прошивки, которые не догадываются о реальной схемотехнике. А можно добавить в загрузчик Decrypter, который

будет записывать только тот артефакт, который после расшифровки содержит валидную контрольную сумму и подпись.

7.12 Итоги

Суммируя вышесказанное, если говорить про освоение нового микроконтроллера или очередной электронной платы, то надо довести прошивку до ортодоксально канонической формы. Что значит "ортодоксально каноническая форма"?

Вот перед Вами фундаментальные атрибуты ортодоксально канонического состояния прошивки:

1. Прошивка собирается из GNU Make скриптов. Это необходимо для сборки по клику на *.bat скрипт. Плюс сборка из скриптов нужна для подключения сборки к серверу автоматического построения Jenkins.
2. Сборка без RTOS. Например, загрузчику RTOS нужна, как собаке бензобак. Поэтому безусловно надо научиться собирать проекты без RTOS.
3. В сборке присутствуют драйверы для следующих базовых аппаратных подсистем: SysTick, NVIC, PLL, GPIO, PINMUX, FLASH, TIMER, DMA, UART.
4. В сборке присутствует HeartBeat LED. Чтобы просто глядя на устройство убеждаться, что прошивка не зависла.
5. В сборке присутствует драйвер кнопки. Если только плата поддерживает кнопки.
6. В прошивку добавлен драйвер аппаратных таймеров. Для вычисления микросекундных пауз. Для изменения up-time программы. Для анализа времени исполнения участков кода и для диспетчера задач.
7. В сборке присутствует диспетчер задач на базе компонента Limiter.
8. В сборке присутствует UART-CLI.
9. В сборке присутствует NVRAM.
10. В сборке присутствует API для пуска модульных тестов.
11. Модульные тесты можно вызывать из CLI.
12. Также следует сразу подготовить отдельную сборку загрузчика через UART-CLI.

Вот, пожалуй, и всё. Двенадцать пунктов.

Только на этой почве можно полноценно начинать наращивать какой бы то ни было функционал или приложение. А без этого минимума-минимумов будет не разработка, а стрельба из лука с завязанными глазами.

CLI, NVRAM и сборка из скриптов это норма. Это так же принято, как школьное образование, медицинское страхование и пенсионная система.

Как по мне эти атрибуты являются просто джентльменским набором инженера для абсолютно любой нормальной, современной, взрослой, промышленной прошивки. Своего рода коробочка в которую можно положить любой функционал. Если у Вас в репозитории и прошивке всего этого нет, то, наверное, говорить о разработке какого-либо устройства просто не следует. Как ни крути, но сначала надо поднять систему. Писать прошивку без этих 7-10 свойств - это как ходить по улице без одежды. Понимаете?...

7.13 Гиперссылки

1. Зачем Программисту Микроконтроллеров Диофанты Уравнения
2. DevOps для производства Firmware
3. NVRAM для микроконтроллеров
4. Почему Нам Нужен UART-Shell?
5. Модульное Тестирование в Embedded
6. Диспетчер Задач для Микроконтроллера
7. Атрибуты Хорошего Загрузчика
8. Способы Отладки и Диагностики FirmWare

Глава 8

Архитектура Хорошего Кода Прошивки (Массив - наше всё)

"Массив - это наше всё"

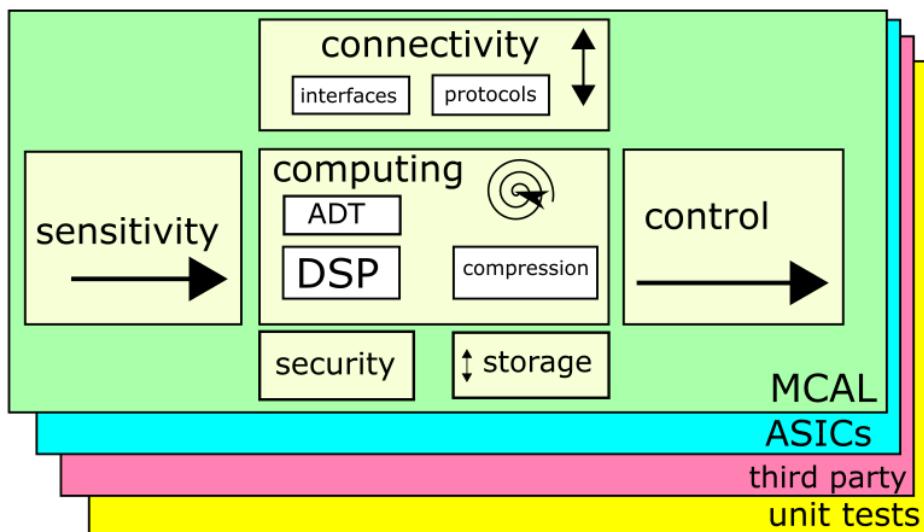


Рис. 8.1: Архитектура хорошей прошивки

8.1 Пролог

Массивы окружают нас всегда и везде: ступеньки на лестнице, многоквартирные дома, кирпичи в кирпичной кладке, вагоны в составе поезда, шпалы под поездом, дачные домики вдоль улицы, деревья вдоль аллеи, товары в универмаге, книги на полке, буквы в строках на страницах книг, строевая ходьба в армии, бусы на груди у женщин, кресла в кинотеатре, плитка на полу, тарелки на кухне, аминокислоты в молекуле ДНК, сокеты на удлиннителе розеток, кнопки на клавиатуре.

Всё это массивы, массивы, массивы. Никуда не деться от этих пресловутых массивов. Господа. Понимаете? Даже кисти рук и ног это тоже массивы пальцев.

Не нужно завершать университеты, чтобы осознать, что Си-код надо тоже называть на массивы.

Вот и получается, что, как ни крути, а и архитектуру кода программ тоже интуитивно так и хочется везде, где можно, организовывать в массив. Это нормально и естественно. Так и надо делать.

Поэтому в этом тексте я накропал о некоторых трюках в организации кода для микроконтроллеров. Кому-то при прочтении может показаться, что это всё очевидно же, обыденно и прозаично, однако за 12 лет работы в России я видел что-то похожее только в одном единственном английском проекте, и то лишь от части. Итак, поехали...

8.2 Массивы конфигурационных структур

Дело в том, что для большинства прошивок конфиги на 101 процент статические. То есть конфиги для прошивки, как ни крути, а известны уже до этапа компиляции программы. Да, это так... Это особенность embedded программирования. У автомобиля не вырастет пятое колесо во время езды. Понимаете?

Вот и получается, что один из самых нормальных способов передавать конфиги в прошивки - это через переменные окружения. Про это у меня есть вообще отдельный текст .

Это удобно с точки зрения масштабирования кодовой базы. Плюс в том, что переменные окружения можно определять прописывая прямо в скриптах (Make, CMake и т.п.).

Однако не всё удобно передавать через переменные окружения. Это происходит из-за того, что переменные конфигов имеют разные типы данных: целые числа, вещественные числа, комплексные числа, строки. К этому приходится приспособливаться. Большинство просто добавляет тонны макроопределений и превращает свой код в свалку. Не надо так. Это не наш путь.

Я решил пойти другим путём и остальные более детальные конфиги стал передавать через массивы структур. Вот как тут конфиг монохроматических светодиодов:

Листинг 8.1: конфиг монохроматических светодиодов

```
#include "led_mono_config.h"
#include "data_utils.h"

const LedMonoConfig_t LedMonoConfig[] = {
{
    .num = 1, .period_ms = 500,
    .phase_ms = 500, .duty = 10,
    .pad = { .port = PORT_D, .pin = 15 },
    .name = "Green", .mode = LED_MODE_BAM,
    .active = GPIO_LVL_LOW, .valid = true,
},
{
```

```

    .num = 2, .period_ms = 1000, .phase_ms = 0,
    .duty = 10, .pad = { .port = PORT_D, .pin = 13 },
    .name = "Red", .mode = LED_MODE_OFF,
    .active = GPIO_LVL_LOW, .valid = true ,
},
{
    .num = 3, .period_ms = 1000,
    .phase_ms = 0, .duty = 10,
    .pad = { .port = PORT_D, .pin = 14 },
    .name = "Yellow", .mode = LED_MODE_PWM,
    .active = GPIO_LVL_LOW, .valid = true ,
},
};

LedMonoHandle_t LedMonoInstance[] = {
{ .num = 1, .valid = true },
{ .num = 2, .valid = true },
{ .num = 3, .valid = true },
};

uint32_t led_mono_get_cnt(void) {
    uint32_t cnt = 0;
    uint32_t cnt1 = 0;
    uint32_t cnt2 = 0;
    cnt1 = ARRAY_SIZE(LedMonoInstance);
    cnt2 = ARRAY_SIZE(LedMonoConfig);
    if (cnt1 == cnt2) {
        cnt = cnt1;
    }
    return cnt;
}
}

```

Далее одна универсальная функция правильно обрабатывает каждый узел с конфигами. Один за другим для каждого элемента конфигурационного массива. Очень удобно.

Массивы структуры ещё и потому удобны, что *.tar файл покажет вам смещение в *.bin файле, где лежит эта структура. И вы сможете вручную аккуратно изменить константу перед прошивкой. Но это, конечно же, как опция. В реальности так даже не придется делать. Но вдруг...

Массивы конфигурационных структур должны быть буквально для всего.

В каждом микроконтроллере есть десятки источников аппаратных прерываний. В связи с этим, в конфигах прошивки также должен быть массив структур, который укажет, каждый источник прерывания, его номер, имя прерывания и с каким приоритетом следует прерывание включать, а какие прерывания и вовсе принудительно отключить при старте по умолчанию.

8.3 Массив функций инициализаций прошивки

Что такое инициализация? Это ведь упорядоченное множество Си-функций. В переводе на кухонный язык инициализация - это последовательность запуска Си-функций в правильном порядке. Только и всего... А почему бы тогда не

организовать эту последовательность в массив функций? Да запросто... Вот.

Листинг 8.2: Массив функций инициализаций прошивки

```
#ifdef HAS_MICROCONTROLLER
#include "board_config.h"
#define BOARD_INIT \
{ .init_function = board_init, .name = "board", },
#else /*HAS_MICROCONTROLLER*/
#define BOARD_INIT
#endif /*HAS_MICROCONTROLLER*/

/*Order matters!*/
define INIT_FUNCTIONS \
MCAL_INIT \
HW_INIT \
INTERFACES_INIT \
PROTOCOLS_INIT \
CONTROL_INIT \
STORAGE_SW_INIT \
SW_INIT \
UNIT_TEST_INIT \
ASICS_INIT \
BOARD_INIT

/*Order matter!*/
const SystemInitInstance_t SystemInitInstance[] = {INIT_FUNCTIONS};

bool system_init(void) {
    bool res = true;
    System.init = false;
    System.init_finish = false;
    uint32_t init_cnt = system_init_get_cnt();
    if (init_cnt) {
        uint32_t i = 0;
        uint32_t ok = 0;
        char InitOrder[400] = {0};
        strcpy(InitOrder, "");
        for (i = 0; i < init_cnt; i++) {
            res = watchdog_proc();
            res = SystemInitInstance[i].init_function();
            if (res) {
                ok++;
            }
            printf("%u:----^" CRLF, i + 1);
            sprintf(InitOrder, sizeof(InitOrder), "%s%s",
                    InitOrder, SystemInitInstance[i].name);
        }
        LOG_INFO(SYS, "InitOrder:[%s]", InitOrder);
        if (ok == init_cnt) {
            res = true;
            LOG_INFO(SYS, "InitComplete %u", init_cnt);
            System.init = true;
        } else {
            System.init = false;
            double init_completeness = 100.0 * ((double)ok) /
                ((double)init_cnt);
        }
    }
}
```

```

LOG_ERROR(SYS,
           "InitIncomplete %u/%u, Only %5.2f %%",
           ok, init_cnt, init_completeness);
res = false;
}

bool all_uniq = system_init_array_uniq();
if (all_uniq) {
    LOG_INFO(SYS, "AllInitsUniq!");
} else {
    LOG_ERROR(SYS, "SpotDoubleInits!!");
}
res = res && all_uniq;
}

System.init_finish = true;
LOG_INFO(SYS, "InitCnt:%u", init_cnt);
return res;
}

```

Выигрыш тут тройной:

1. Вся инициализация в одном месте.
2. Порядок инициализации определён индексом в массиве.
3. Для каждой функции инициализации легко выполнить один какой-то общий пролог и эпилог-код. Например печать порядкового номера в UART, дергание GPIO импульса или сброс сторожевого таймера. Если прошивка зависнет в инициализации, то посчитав импульсы в GPIO вы поймете на какой функции (индекс в массиве указателей) прошивка зависла в инициализации.

Да и потом, прошивка, по хорошему, должна печатать и отчет о своей загрузке в UART или в SD карту. Это так называемый лог начальной загрузки (ЛНЗ).

```

12.739.988 I, [Nau8814] Init 1 Nau8814
12.755.899 I, [IMONI] Init Nau8814 OK
42:----^
12.774.980 V, [SYS] BoardInit
12.787.981 I, [SYS] BoardName: [TEST-TEST]
12.805.982 I, [SYS] XTall:8000000 Hz
12.820.983 I, [IMONI] Init board OK
43:----^
12.839.984 E, [SYS] InitIncomplete 42/43, Only 97.67 %
12.861.985 I, [SYS] AllInitsUniq!
12.875.986 I, [SYS] InitCnt:43
12.889.987 E, [IMONI] Init SYS Error
12.905.988 I, [SYS] init Error!

12.928.989 I, [SYS] ProgramLaunched!
12.936.919 I, [recom] Date:Mon 22 2024

```

Рис. 8.2: лог начальной загрузки (ЛНЗ)

Лог позволит анализировать ошибки в конфигурациях или брак в аппаратуре PCB (железе) еще до запуска суперцикла.

8.4 Массив функций для суперцикла

У каждой прошивки так или иначе есть суперцикл. Либо он прописан явно внутри main(), либо в составе bare-bone потока на какой-нибудь RTOS. Тут тоже, функции суперцикла можно объединить в массив структур. При этом каждую функцию можно пропускать через компонент limiter и, тем самым, вызывать её с определённым в конфиге периодом. Не чаще чем, скажем, 500ms.

Про это у меня есть отдельный текст. Называется Диспетчер Задач для Микроконтроллера

Листинг 8.3: Массив функций для суперцикла

```
#ifdef HAS_BOARD_PROC
#include "board_at_start_f437.h"
#define BOARD_TASK { .name="board" ,
    .period_us=BOARD_POLL_PERIOD_US,
    .limiter.function=board_proc },
#else
#define BOARD_TASK
#endif /* */

#define TASK_LIST_ALL \
    ASICS_TASK \
    APPLICATIONS_TASKS \
    BOARD_TASK \
    MCAL_TASKS \
    TASK_CORE \
    COMPUTING_TASKS \
    CONNECTIVITY_TASKS \
    CONTROL_TASKS \
    SENSITIVITY_TASKS \
    STORAGE_TASKS

TaskConfig_t TaskInstance[] = {
    TASK_LIST_ALL
};
```

8.5 Массив параметров в NVRAM

Любой прошивке надо запоминать какие-то параметры в энергонезависимой памяти. Это происходит по разным причинам. Про это есть отдельный текст:

NVRAM для микроконтроллеров

Листинг 8.4: Массив параметров в NVRAM

```
const ParamItem_t ParamArray[] = {
    FLASH_FS_PARAMS
    PARAMS_GNSS
    IWDG_PARAMS
    PARAMS_PASTILDA
    PARAMS_SDIO
    PARAMS_TIME
    PARAMS_BOOTLOADER
```

```

    { .facility=BOOT, .id=PAR_ID_BOOT_CMD, .len=1,
    .type=TYPE_UINT8, .name="BootCmd" },
    { .facility=BOOT, .id=PAR_ID_REBOOT_CNT, .len=2,
    .type=TYPE_UINT16, .name="ReBootCnt" },
    { .facility=SYS, .id=PAR_ID_SERIAL_NUM, .len=4,
    .type=TYPE_UINT32, .name="SerialNum" },
};


```

У каждого NVRAM параметра есть минимум такие свойства как

Поэтому в прошивке определяем массив структур, который явно покажет с какими параметрами прошивка будет работать после запуска.

8.6 Массив отладочных токенов для диагностики

Как правило в прошивках есть UART для printf() отладки. Одновременно с этим, каждая взрослая прошивка состоит из десятков программных компонентов. Для того, чтобы отличать какому именно программному компоненту принадлежат те или иные отладочные сообщения в коде прошивки должно быть определено перечисление, где каждому программному компоненту будет присвоено натуральное число. Таким образом, при печати лога (в UART или SD карту) каждому такому числу ставится в соответствие текстовый токен. Поэтому в прошивке должен быть определен массив структур, где каждая структура ставит в соответствие числу его текстовый токен.

Листинг 8.5: Массив отладочных токенов для диагностики

```

const static FacilityInfo_t FacilityInfo [] = {

#ifndef HAS_AES
    { .facility = AES, .name = "AES" },
#endif /*HAS_AES*/

#ifndef HAS_AD9833
    { .facility = AD9833, .name = "AD9833" },
#endif /*HAS_AD9833*/

#ifndef HAS_NOR_FLASH
    { .facility = NOR_FLASH, .name = "NorFlash" },
#endif /*HAS_NOR_FLASH*/

#ifndef HAS_NVRAM
    { .facility = NVRAM, .name = "NvRam" },
#endif /*HAS_NVRAM*/
    ...
};


```

Благодаря этим токенам в этом логе явственно видно какому именно программному компоненту принадлежит каждая строчка в логе.

Удобно? Очень!

```

COM38 - Tera Term VT
File Edit Setup Control Window Help
2:8-->
2:8-->ta
129.521.906 W,[Board] RunTestAudio
129.537.907 W,[Board] SERVICE PressOk Duration:2500 ns
132.839.908 I,[Relay] RELAY7 BlinkDone!
132.259.909 I,[Nau8814] Play1DHz,Ampplitude:1000
132.278.910 I,[Nau8814] Play 1DHz Ampplitude: 1000
132.297.911 I,[I2S] I2S2, DAC1, Amp:1000, Phase:0
132.318.912 W,[SwDACL] Mode:Sin, Freq 1000 Hz, Amp 1000, Phase 0 ms, OffSet 0
132.346.913 I,[SwDACL] CalcArray:Dac1,Freq:48 kHz,1 Period
132.369.914 I,[SwDACL] SampleSize 2 Byte
132.385.915 I,[SwDACL] SampleTime 20033 ns
132.403.916 I,[SwDACL] Amplitude: 1000,Freq:1 kHz,Sample:2byte,Patt:XX
132.429.917 I,[SwDACL] MaxTime 1ns
132.444.918 I,[SwDACL] SampleCnt 48
132.459.919 I,[SwDACL] SampleMode SIN
132.507.920 I,[SwDACL] RecReady
132.522.921 I,[I2S] Write I2S_2_Words:48
132.539.922 W,[I2S] I2S2_Set,Role:MasterTx
132.557.923 W,[I2S] Set,I2S2,Role:MasterTx
132.575.924 W,[I2S] I2S_2_SampleRate:48000 Hz,RxBuff:600 Sample,BusRole:MasterTx,Ds
132.621.925 W,[GPIO0] PB15,I2S2_SD,Dir:Out,Mode:ALT1,Mux:5,LL:H,Pull:Air,Pin:0,LD:0,
132.651.926 I,[GPIO0] B,ClockOn
132.665.927 I,[GPIO0] 3,ClockOn
132.679.928 W,[I2S] IntOn
132.691.929 I,[I2S] I2S2_PlayStarted...
132.760.930 I,[Nau8814] I2sPlayOk
132.823.931 I,[Nau8814] PlayConOk
140.388.932 W,[Nau8814] Listen:1,Dur:1000 ms
140.464.933 W,[I2S] Listen:2,Dur:1000 ms
140.538.934 I,[I2S] SampleTime:20.833u
140.606.935 I,[I2S] RecDuration:6.250m
140.673.936 I,[I2S] Parts:160-160
140.736.937 W,[I2S] I2S2_Stop,Role:MasterTx,Ds

```

Рис. 8.3: У каждой строчки есть префикс указывающий какому компоненту она принадлежит

8.7 Массив команд для CLI

В каждой нормальной взрослой прошивке есть UART-CLI. Для отладки очень полезна UART-CLI. Функции CLI тоже складируем штабелями в массив структур. Подробнее про это можно почитать в тексте: Почему Нам Нужен UART-Shell?

Листинг 8.6: Массив команд для CLI

```

#define CLI_CMD(LONG_CMD, SHORT_CMD, FUNC) \
{ .short_name = SHORT_CMD, \
  .long_name = LONG_CMD, \
  .handler = FUNC }

bool nau8814_i2c_ping_command(int32_t argc, char* argv[]);  

bool nau8814_reg_map_command(int32_t argc, char* argv[]);

#define NAU8814_COMMANDS \
NAU8814_DAC_COMMANDS \
NAU8814_ADC_COMMANDS \
CLI_CMD("nau8814_ping", "nap", nau8814_i2c_ping_command ), \
CLI_CMD("nau8814_map", "nrm", nau8814_reg_map_command ),

#define CLI_COMMANDS \
ASICS_COMMANDS \
APPLICATIONS_COMMANDS \
CONTROL_COMMANDS \
CONNECTIVITY_COMMANDS \
COMPUTING_COMMANDS \
MCAL_COMMANDS \
MULTIMEDIA_COMMANDS \
PROTOTYPE_COMMANDS \

```

```

STORAGE_COMMANDS \
SENSITIVITY_COMMANDS

const CliCmdInfo_t CliCommands[] = {CLI_COMMANDS};

```

8.8 Массив функций-модульных тестов

Прошивка может из CLI вызывать модульные тесты, которые есть у неё на борту. Список модульных тестов это тоже массив структур, где каждая содержит указатель на функцию с тестом и название самого теста.

Листинг 8.7: Массив функций-модульных тестов

```

bool test_c_types(void) {
    LOG_INFO(TEST, "%s() ..", __FUNCTION__);
    bool res = true;

    EXPECT_EQ(4, sizeof(long));
    EXPECT_EQ(4, sizeof(1UL));
    EXPECT_EQ(4, sizeof(1L));
    EXPECT_EQ(4, sizeof(size_t));
    EXPECT_EQ(4, sizeof(1l));

    EXPECT_EQ(4, sizeof(1));
    EXPECT_EQ(4, sizeof(-1));
    EXPECT_EQ(4, sizeof(0b1));
    EXPECT_EQ(4, sizeof(1U));
    EXPECT_EQ(4, sizeof(1u));
    EXPECT_EQ(4, sizeof(1.f));

    LOG_INFO(TEST, "%s() Ok", __FUNCTION__);
    return res;
}

#define TEST_SUIT_SW \
    {"array_init", test_array_init}, \
    {"bit_fields", test_bit_fields}, \
    {"c_types", test_c_types}, \
    {"memset", test_memset}, \
    {"memcpy", test_memcpy}, \
    {"bit_shift", test_bit_shift}, \
    {"int_overflow", test_int_overflow}, \
    {"sprintf_minus", test_sprintf_minus}, \
    {"endian", test_endian}, \
    /*Compile time assemble array */
const UnitTestHandle_t TestArray[] = {
#endif HAS_SW_TESTS
    TEST_SUIT_SW
#endif /*HAS_SW_TESTS*/

#ifndef HAS_HW_TESTS
    TEST_SUIT_HW
#endif /*HAS_HW_TESTS*/

```

```
};
```

8.9 Массив функций потоков для RTOS

Если ваша прошивка работает какой-нибудь RTOS, то надо создать массив структур, которые будут содержать указатели на функции потоков, размер стека и приоритет для каждой задачи аргументы к потоку и прочее. И так для каждой задачи. Очевидно, что если потоков больше чем два, то имеет смысл сделать массив структур с конфигами для каждого потока.

Вот как тут.

Листинг 8.8: Массив функций потоков для RTOS

```
#include "FreeRTOSConfig.h"
```

```
#include "free_rtos_drv.h"
#include "data_utils.h"
#ifndef HAS_KEEPASS
#include "keepass.h"
#endif /*HAS_KEEPASS*/
```

```
const RtosTaskConfig_t RtosTaskConfig[] = {
    { .num=1, .TaskCode=bare_bone,
      .name="BareBone",
      .stack_depth_byte=2048,
      .priority=PRIORITY_LOW,
      .valid=true },
    { .num=2,
```

```
      .TaskCode=default_task,
      .name="DefTask",
      .stack_depth_byte=256,
      .priority=PRIORITY_LOW,
      .valid=true },
    { .num=3,
```

```
      .TaskCode=keepass_proc_task,
      .name="KeePass",
      .stack_depth_byte=1024,
      .priority=PRIORITY_LOW,
      .valid=true },
};
```

```
#endif /*HAS_KEEPASS*/
};

RtosTaskHandle_t RtosTaskInstance[] = {
    { .num=1, .valid=true },
    { .num=2, .valid=true },
    { .num=3, .valid=true },
};
```

```
#ifndef HAS_KEEPASS
    { .num=3, .valid=true },
#endif
};

uint32_t rtos_task_get_cnt(void) {
```

```

    uint32_t cnt = 0 ;
    uint32_t cnt1 = 0 ;
    uint32_t cnt2 = 0 ;
    cnt1 = ARRAY_SIZE(RtosTaskConfig) ;
    cnt2 = ARRAY_SIZE(RtosTaskInstance) ;
    if (cnt1==cnt2) {
        cnt = cnt1 ;
    }
    return cnt ;
}

```

Таким образом вы будете помнить про все потоки в данной сборке.

8.10 Итоги

Подводя черту можно заменить, что в программировании микроконтроллеров массив это универсальная структура данных. Массив это фундаментальная структура данных. При добавлении в прошивку очередного программного компонента вы просто добавляете в каждый массив по одному элементу.

А теперь внимание.. Формирование этих всех массивов можно организовать на этапе отработки препроцессора! Да... Следите за руками... Переменные окружения вызывают нужные скрипты сборки. Скрипты сборки передают макросы препроцессора. Макросы препроцессора выбирает нужный код. Компилятор собирает только нужный код! Easy!

Благодаря тому, что у Вас каждая сущность хранится в массиве Вы можете также в RunTime проверять конфиги на наличие дубликатов, найти конфликты и прочее.

Можно и вовсе справедливо заметить, что любая программа как машинный код - это тоже не что иное как массив assembler инструкций в ROM памяти. Да.. Именно так... А микропроцессор - это эдакая электрическая цепочка, которая просто исполняет одну инструкцию за другой из ROM памяти, пока не достигнет конца массива инструкций. Только и всего...

В сухом остатке, благодаря организации всех программных сущностей в массивы у Вас прошивка, как кристалл растёт из одной исходной точки. У Вас одна точка отсчета для техподдержки и одна точка отсчета для масштабирования всего проекта. Крастота несусветная!

Массив структур конфигураций, массив задач , массив функций инициализации и прочие массивы - это все является механизмом полиморфизма при программировании на си.

Надеюсь, что этот текст поможет другим программистам микроконтроллеров тоже эффективнее компоновать свои сборки.

8.11 Гиперсылки

1. NVRAM для микроконтроллеров

2. Почему Нам Нужен UART-Shell?
3. Архитектура Хорошо Поддерживаемого драйвера
4. Модульное Тестирование в Embedded
5. Диспетчер Задач для Микроконтроллера

Глава 9

Модульное Тестирование в Embedded

"Код без модульных тестов — Филькина грамота"

9.1 Пролог

Часто в РФ приходится слышать мнение, что в Firmware разработке якобы в принципе не может быть никакого модульного тестирования. Во всяких военных НИИ даже бытует расхожее мнение

"Не нужны никакие тесты. Если программист хороший, то и код он пишет без ошибок."

Попробуем разобраться какие есть плюсы и минусы в модульном тестировании и понять надо оно или нет.

Что такое модульный тест? Это просто функция, которая вызывает другую функцию с известными константными параметрами. А потом проверяет результат. Это способ получить обратную связь от того кода, который был написан Software In The Loop (SIL). Модульные тесты должен писать прежде всего сам автор основного кода. Также весьма полезны непредвзятые тесты от человека со стороны.

9.2 Достоинства модульных тестов

Зачем могут быть нужны модульные тесты?

1. Прежде всего для контроля работоспособности функционала. Тут всё очевидно. Тест доказывает, что какой-то код работает.
2. Для безопасного перестроения программы (рефакторинга). Часто первый написанный код не самый понятный, переносимый и быстрый. Код просто минимально допустимо работает. При масштабировании оригиналный код од-

нозначно придётся менять, упрощать. Модульные тесты дадут сигнал, если перестройка проекта вышла из-под контроля и , что-то рассыпалось.

3. Тесты как способ документирования кода. Посмотрите на тест и вы увидите как заполнять прототип функции и вам не придётся писать doxygen для каждой функции или параграфы из комментариев.

Хороший С-код понятен без комментариев.

4. Модульные тесты служат критерием завершения работы на конкретном этапе. Разрабатываем код пока не пройдут тесты. Далее принимаемся за следующий программный компонент. Всё четко и понятно.
5. Существующие тесты помогут для контроля исполнения работы. Team Lead может написать тесты, а инженер-программист разработает программные компоненты для прохождения этих тестов.
6. Для снятия ответственности с программиста. Программисты должны быть сами заинтересованы в том, чтобы писать код с тестами. В этом случае они всегда смогут сказать:

Смотрите. Код, который я написал, прошел тесты позавчера. Значит я не причем в том, что сегодня прототип загорелся перед инвесторами

Если в организации не принято тестировать код, то в таких организациях как правило крайними делают именно программистов. Вам оно надо?

7. Для покрытия кода. Если есть тест для компонента, значит есть и покрытие кода в этом компоненте. При настроенном измерении покрытия кода можно при помощи модульных тестов выявлять лишний и недостижимый код.
8. Когда практикуется тестирование кода, то и код естественным образом получается структурируемый, модульный, простой, понятный и переносимый. Это один из любимых побочных эффектов модульных-тестов, модульные тесты убивают спагетти-код.

Напротив, когда тесты не пишутся, то код получается похож на спагетти: циклические функции по 5k строк, перемешанный аппаратно-зависимый и аппаратно-независимый код, куча магический циферок и прочее.

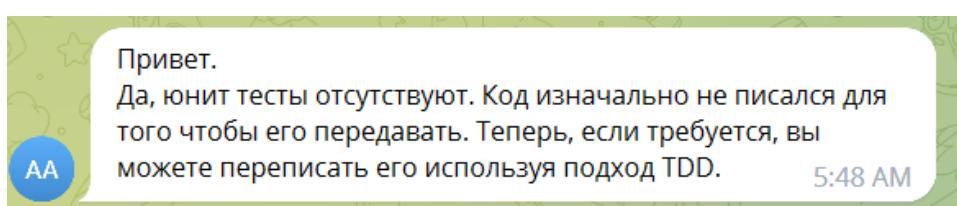


Рис. 9.1: Жалкое оправдание за галимый код

9. Модульные тесты дадут вам гарантию, что в другом окружении (на другой платформе (PowerPC, AVR, ARC, ARM Cortex-M, x86, RISC-V) и другом компиляторе (IAR, CCS, GCC, GHS, Clang)) функционал будет вести себя как и прежде и работать как и задумывалось изначально.

Без тестов при миграции на очередной микроконтроллер другого вендора или альтернативный компилятор языка С у вас будет просто не работать часть функционала и вы не будете понимать почему это происходит, далее вы и еще будете долго ловить гейзенбаги и шреденбаги в самый неподходящий момент и в самом неподходящем месте.

10. Модульные тесты полезны для регрессионного тестирования. Для гарантии, что новое изменение не поломало старый функционал. Так как все зависимости предугадать невозможно.

11. Разработка с тестированием придает процессу создания софтвера положительный азарт. После коммита так и хочется открыть СИ и проверить прошли ли все тесты. Если прошли, то наступает радость, эйфория и гордость за проделанную работу.

Напротив при программировании без тестирования разработчика преследует чувство тревоги, неуверенности. Такие программисты так и ждут, что в любой день придут ругать и критиковать его за найденную ошибку в программе.

12. Прогон модульных тестов ровным счетом ничего не стоит. Тесты можно прогонять хоть каждый день (например во время перерыва или по ночам). Напротив, ручное же тестирование программ по check-листу очень дорого так как это человеко-часы. Плюс при ручном тестировании в ход ступает целый калейдоскоп когнитивных искажений свойственный людям: выдать желаемое за действительное, преувеличение и прочее.

13. Модульные тесты это очень удобная метрика для менеджера проекта. Если тесты проходят и их количество день ото дня возрастает то значит , что всё идет хорошо.

14. Если код покрыть модульными тестами, то в пошаговой отладке нужда как таковая отпадает сама собой. Чтобы найти баг надо только прогнать скопом все тесты и отчет сам покажет, какая функция оказалась бажной.

9.3 Недостатки модульных тестов

1. Нужна память для хранения кода с тестами. Часто можно услышать высказывание:

"Я не буду добавлять тесты в сборку, так как у меня мало Fash памяти в микроконтроллере"

Разруливается эта ситуация очень просто. Если все тесты не помещаются в NorFlash память, то делим общее количество тестов на несколько сборок. Записываем тесты по очереди на Target, прогоняем группы тестов и сохраняем их логи в отчет или генерируем CSV таблицу с полным отчётом. Всё это можно легко сделать автоматически в том же CI/CD на основе бесплатного Jenkins. Но это при условии, что в прошивке у вас есть загрузчик и интерфейс командной строки Shell/CLI, чтобы заливать части софта и просить запустить тесты.

Или ещё проще. Берём микроконтроллер того же ядра ARM Cortex-M4, но с большим объёмом Flash памяти. Гоняем тесты на нем. Нарабатываем в нем корректный код. Как только тесты будут проходить, работающий код копаем в MCU с меньшим Flash.

Это как в атомной промышленности плутоний нарабатывают на стационарных физических реакторах с графитовым замедлителем. Затем плутоний извлекают из физического реактора, химическим образом отчищают, переплавляют и встраивают в ядерные заряды или в РИТЭГи для космических зондов, маяков и прочего.

2. Сам код тестов может содержать ошибку. Вот тут придется организовать инспекцию программ. Тесты надо писать так чтобы в них нечemu было ломаться.
3. Можно написать чрезмерное и избыточное количество тестов. Повторные тесты только с разными именами. Это реальная проблема. Для ликвидации этого по хорошему надо измерять покрытие кода. А это весьма высокоорганизованный процесс. Нужен программный инструментарий для измерения покрытия кода в RunTime прямо на Target(e). Как правило такие технологии платные (Testwell CTC++, LDRA).

Однако есть и второй более простой путь. Писать тесты исходя из технических требований (если они есть). Тут всё просто. Есть требование - будет тест. Нет требования - не будет теста.

Однако проблема этого метода в том, что в России часто ведут разработки без чётких требований к изделию. Без письменного технического задания. Так, поболтали и пошли с чем-то колупаться.

4. Модульные тесты не могут протестировать весь нужный функционал. Нужны еще интеграционные тесты. На эту тему есть множество мемов. Классический пример - это uTest для дверей в метро.

По отдельности они могут отлично открываться, но это не значит, что дверцы откроются одновременно в одну сторону.

9.4 Общие принципы модульного тестирования

Ок. Допустим, что пришли к выводу, что модульные тесты всё-таки нужны. Однако как же делать это пресловутое модульное тестирование?

“The app is great, all unit tests passed”



Рис. 9.2: Формально модульные тесты проходят

1. Код отдельно тесты отдельно. Тесты и код разделять на разные программные компоненты. Распределять по разным папкам. Один и тот же тест тестирует разные версии своего компонента.
2. Каждый тест должен тестировать только что-то одно
3. Прежде чем начать чинить баг надо написать тест на этот баг. Это позволит отладить сам тест.
4. Тест должен быть полностью детерминированным. Алгоритм теста должен быть постоянным. Не стоит использовать в teste генератор случайных чисел. Иначе можно получить тест который, то будет проходить, то не будет.

5. Тест должен легко пониматься. Юнит тест должен быть простым как табуретка.
6. Тест должен быть коротким
7. У теста минимальные затраты на сопровождение
8. Тест должен проверять предельные случаи
9. Тесты должны быть интегрированы в цикл разработки
10. Тест должен легко запускаться. Например по команде из UART CLI.
11. Тест должен быть устойчив к рефакторингу
12. Тест проверяет только самые важные участки кода
13. Чинить тесты надо в первую очередь. После продолжать писать production код.
14. Тесты либо работают либо удалены.
15. Желательно чтобы тест писал не разработчик тестируемого компонента. Нужен непредвзятый взгляд. Например разработчик другого компонента может добавить часть своих тестов для компонента своего смежника.
16. Тесты составлять по принципу три A: Arrange, Act, Assert AAA
17. В модульных тестах не должно быть оператора if в явном виде
18. Модульный тест должен воспроизводится. Успешный тесты (зеленые) должны быть успешными при повторном запуске. Красные тесты на базном коде должны постоянно падать. Результат модульного теста не должен зависеть от случайных величин таких как "угол Солнца над горизонтом или фаза Луны".
19. Любой предыдущий тест не должен ломать последующий тест. Даже если все тесты по отдельности проходят. То есть любая перестановка порядка исполнения модульных тестов должна проходить успешно.
20. Модульные тесты надо прогонять на Target устройстве. То есть прямо на микроконтроллере (см HIL). В естественной среде обитания. Запускать тесты можно по команде через UART-CLI или Shell при помощи Putty или TeraTerm

9.5 Автотесты

Модульные тесты и автотесты это, суть одно и то же. Мы по команде разом пускаем на исполнение сотни и сотни модульных тестов.

Дело в том , что система тестирования как правило оказывается сложнее самого продукта.

1. Вы написали код для воспроизведения звука через микросхему аудиокодека SGTL5000XNAА3, MAX9860 или WM8731. Как это автоматически протестировать? Как проверить, что звук в самом деле есть? А вот как. Надо воспроизвести синус (например на 2000 Hz), записать 50...100ms звука встроенным в этот же аудио кодек ADC, затем вычислить DFT, найти максимум, сверить максимум DFT с настройками воспроизведения. Если есть совпадение, то тест пройден. Иначе нет. В результате нет нужды в операторе чтобы подтверждать услышанный звук.

И затраты на разработку тестирование и отладку программных компонентов ADC, DFT окажутся в три-четыре раза больше, чем на разработку простецкого драйвера DAC.

Поэтому тут может оказаться лучше прибегнуть к покупке готового тестировочного оборудования, в данном примере какого-нибудь акустического спектротанализатора с UART интерфейсом.

2. Для автоматического тестирования копеечного графического монохромного дисплея SSD1306 нужна какая-то камера с видео аналитикой, чтобы распознать и классифицировать то тестовое изображение, которое мы воспроизвели. Можно, например, отрисовывать на дисплее SSD1306 QR-коды и распознавать их считывателем DataMatrix кодов с UART интерфейсом.

В связи с этим общее правило автоматического тестирования таково

"Разработка системы авто тестов всегда дороже разработки продукта для которого она предназначена."

Если вы затратили на разработку продукта N рублей, то на разработку стенда авто тестов вы потратите 3N рублей. Тут вариантов нет. При этом разрабатывать систему авто тестов должны более квалифицированные инженеры и программисты, чем те кто разрабатывают сам продукт. Вот так...

9.6 Итоги

Написание кода - это как восхождение на вертикально отвесную скалу. После каждого метра подъёма надо ввинчивать крючки (модульные тесты), чтобы закрепить страховочный карабин. Иначе сорвёшься и упадешь, как самолёт вошедший в штопор.

Плюсов у модульного тестирования много. Минусов мало и они решаемые. Как по мне модульные тесты на самом деле нужны всем: программистам, team lead(ам) и менеджерам.

"Если в сорцах прошивки нет модульных тестов, то это Филькина грамота."

Пишите код firmware с тестами.

9.7 Гиперссылки

1. Запускаем юнит тесты на микроконтроллере
2. Модульное тестирование для малых встраиваемых систем

Глава 10

Атрибуты Хорошего Загрузчика

"Если дом начинается с двери, то прошивка начинается с загрузчика."

10.1 Пролог

Поговорим о загрузчиках для микроконтроллерных проектов. Допустим у вас есть фитнес браслет, электро-самокат, робот пылесос, беспроводные наушники, или электронная зубная щетка и Вы хотите обновить прошивку загрузчиком. Какое же поведение должно быть у хорошего загрузчика?

10.2 Терминология

1. загрузчик - это отдельная прошивка, которая загружает другую прошивку. Обычно загрузчик стартует сразу после подачи электропитания перед пуском самого приложения. Это чисто системная часть кода.
2. боевая flash - диапазон flash памяти в котором исполняется приложение микроконтроллера.
3. временная flash - диапазон Flash памяти (on-chip или off-chip) в которую складируется новая прошивка
4. прошивка (firmware) - содержимое энергонезависимой памяти электронного устройства с микроконтроллером. В прошивке всегда есть код, а иногда ещё образ файловой системы NVRAM, конфиги процессора. Монолитная прошивка может содержать еще и загрузчик.
5. NVRAM - энергонезависимая память с произвольным доступом. По сути Key-Val-Map(ка). В ней могут храниться любые бинарные данные, ассоциированные со своим ID(шником).
6. тыква / кирпич - электронное устройство с бракованной прошивкой, которое либо зависло либо постоянно без конца перезагружается. Такое устройство можно восстановить только программатором.

7. таблица векторов прерываний - это таблица адресов функций обработчиков прерываний. Первая колонка это номер прерывания, вторая колонка абсолютный адрес обработчика прерываний в on-chip NOR-Flash памяти. Обычно в этой таблице сотни записей. У каждого микроконтроллера она своя. В бинарном файле перечислена вторая колонка как массив `uint32_t`: индекс массива-номер прерывания, значение - адрес ISR во Flash памяти.
8. Connectivity - всё то, что связано с интерфейсами и протоколами.

10.3 Достоинства загрузчика

1. Можно обновлять прошивку без программатора (обновление "в полях"). Среди программистов-микроконтроллеров есть даже мем что

Самая лучшая функция прошивки это возможность обновления прошивки.

2. Можно обновлять прошивку по любому интерфейсу: UART, BlueTooth, WiFi, LTE, CAN и прочее.
3. Когда есть загрузчик, то можно делать DevOps. Автоматически обновлять прошивки после сборки из репозитория.

10.4 Недостатки загрузчика

1. Надо выделить под загрузчик часть Flash памяти. Это порядка нескольких десятков килобайт ROM.
2. Устройство стартует чуть медленнее.

Главная задача хорошего загрузчика - это дать возможность обновлять прошивку и ни при каких обстоятельствах не позволить устройству превратиться в тыкву.

Какие программные компоненты обычно должны быть в загрузчике?

Программный компонент	Необходимость
Драйвер Flash памяти	обязательно
Драйвер UART	обязательно
NVRAM	обязательно
CRC	обязательно
CLI	желательно
Драйвер внешней Flash памяти	зависит от платформы
Драйвер SD карты	зависит от платформы
драйвер SPI-flash	зависит от платформы
поддержка Fat-FS	зависит от платформы

10.5 Атрибуты хорошего загрузчика

Далее представлены атрибуты хорошего загрузчика. Эти правила написаны кровью! Можете просто сразу вставлять их в техническое задание на разработку встроенного программного обеспечения.

1. Саму физическую загрузку прошивки должен осуществлять не загрузчик, а как не парадоксально само приложение. Дело в том, что в приложении больше памяти и больше драйверов для разнообразных интерфейсов. Поэтому ничего не мешает 10 процентов прошивки получить по WiFi, 20 процентов по BlueTooth, 30 процентов по RS485 еще 30 процентов по CAN. Затем принять конфиги из модулированного звука через аудиокодек, а настройки NVRAM получить от сканера QR кодов в драйвере видеокамеры. У приложений обычно и так есть поддержка очень богатого connectivity. Если бы вы пытались засунуть поддержку всех возможных интерфейсов и протоколов в загрузчик, то загрузчик был бы больше самого приложения! Поэтому закачкой данных должно заниматься именно приложение. Приложение как губка впитывает прошивку со всех сторон.
2. Загрузчик должен быть компактным в плане требуемого ROM. Если загрузчик маленький, то будет больше места для приложения и его бизнес логики, которая и создает основную ценность продукта.
3. Загрузчик должен быть однопоточной NoRTOS прошивкой. Так можно уменьшить *.bin(арь) и упростить код загрузчика. По факту загрузчику на UART многозадачность нужна как собаке бензобак. Если код простой, то и ломаться там будет не чему и техподдержка и отладка проще.
4. Загрузчик перед записью должен проверять контрольную сумму приложения. Если CRC32 не совпадает то такую прошивку лучше не прописывать. Правильная CRC это всего лишь гарантия, что прошивка как файл передалась корректно. Это не защита от чужеродного firmware.
5. Загрузчик должен уметь прошивать как минимум по UART. Когда всё сломается пере прошивка по UART окажется единственным спасением. Нет проводного интерфейса проще UART. В UART нечему ломаться. UART требует меньше всего NOR-Flash(a). Плюс для UART куча дешевых переходников USB-UART и бесплатного софта для serial портов типа Putty и TeraTerm.
6. Загрузчик должен проверять, что первое слово прошивки по нулевому адресу это реальный для данного микроконтроллера адрес RAM памяти. Дело в том, что в Cortex-M это указатель Stack Pointer прошивки. Если это не так, то эту прошивку загружать нельзя. Она всё равно зависнет тот час же.
7. Загрузчик должен проверять, что в бинарном файле прошивки действительные (валидные) адреса таблицы векторов прерывания. В Cortex-M это адрес

+0x00000004 от начала прошивки в начале бинарного файла. Надо проверить, что там валидный ResetHandler. Что его адрес принадлежит Flash памяти.

8. Перед прыжком в приложение загрузчик должен отключить все прерывания. Иначе программа приложения зависнет, если в ней сработает прерывание для которого в приложении нет функции обработчика.
9. В загрузчике должна быть NVRAM. Это позволит передавать команды от приложения к загрузчику. Или сервер может прописать в NVRAM правильную контрольную сумму для того, чтобы загрузчик мог рассчитать CRC и сверить с тем, что прописал сервер в NVRAM. В случае несовпадения отказаться загружать подозрительную прошивку. Вот минимальный список переменных для управления загрузчиком

Переменная NVRAM	размер, [байт]
Начальный адрес прошивки	4
Ожидаемая контрольная сумма прошивки	4
Команда загрузчику	1
Длина прошивки в байтах	4
Счётчик числа запусков загрузчика (BootCnt)	1
Ключ шифрования	256

10. Перед прыжком в приложение отключить системный таймер, так как в приложении может быть RTOS. Если сработает прерывания по системному таймеру до инициализации планировщика, то приложение с RTOS зависнет.
11. Если это Cortex-M ядро, то перед прыжком в приложение надо указать ядру, где теперь будет новая таблица адресов обработчиков прерывания

Листинг 10.1: Указать начало таблицы прерываний

```
SCB->VTOR = app_start_address;
```

12. Загрузчик может переназначить верхушку стека. Хотя это и так сделает StartUp код приложения до запуска main().

Листинг 10.2: Определить верхушку стека

```
--set_MSP(stack_top);
```

13. У загрузчика должна быть UART-CLI. Это позволит подключиться к консоли загрузчика и проверить диагностику, прописывать NVRAM. Протокол CLI - текстовый. Его понимает и человек и утилита. Через CLI можно даже прошивку обновить.
14. Если загрузчику не удалось записать прошивку из энергонезависимого Flash хранилища (SD-карта, off-chip SPI-flash или on-chip flash), то загрузчик должен написать про это красный текст в UART-CLI и перейти в режим ожидания прошивки по UART.

15. Загрузчик должен как-то проверить, что прошивка, которая поступила при надлежит именно нужному аккредитованному вендору, а не злоумышленнику. Поэтому в прошивку надо добавить специальную константу по константному заданному адресу, чтобы загрузчик мог найти эту константу и в случае её совпадения с нужным значением прописать эту прошивку в боевую flash. Или можно оформить эту магическую константу по произвольному адресу просто добавив перед ней известную преамбулу. Тогда загрузчик может просканировать всю память приложения и найти константу по преамбуле.
16. Прошивка должна приходить в зашифрованном виде. В этом же зашифрованном виде она должна храниться в off-chip flash (SPI или SD). Прошивка должна расшифровываться по кусочкам прямо в загрузчике и после расшифровки записываться в боевую flash. Ключ шифрования брать из NVRAM.
17. Для дополнительной безопасности на электронной плате можно поставить внешний аппаратный сторожевой таймер (например этот TPS3828-33QDBVRQ1). Этот таймер сбрасывается через GPIO pin. Надо периодически подавать отрицательные перепады, чтобы плата не сбрасывалась. Оригинальная прошивка знает про этот pin. Если кто-то попытается записать неоригинальную прошивку, которая не догадывается о существовании внешнего аппаратного сторожевого таймера, то эта неоригинальная прошивка будет постоянно сбрасываться.
18. Instant firmware update (мгновенное обновление прошивки). Загрузчик должен уметь записывать и запускать принятую прошивку по любому смещению адреса Flash памяти. Это позволит сэкономить время на отчистку и запись прошивки в боевую flash память. Вместо удаления старой прошивки и записи на её место новой можно просто переключить боевую память на другой диапазон и тут же запустить на исполнение от туда код.
19. Когда стартует загрузчик или приложение прошивка должна увеличивать счётчик запусков BootCnt. Если счетчик запусков превысит число N (например N=5), то загрузчик/приложение прикажет сам себе остаться в загрузчике. Приложение же должно обнулять счетчик запусков BootCnt после успешной инициализации или, например, на 30й секунде своей работы. Это своего рода защита от тыквы (зависания в инициализации приложения). Если залили приложение, которое из-за исключения постоянно reset(ится) где-то в инициализации, то такое приложение после N reset(тов) автоматически свалится в загрузчик и можно будет спокойно записать по UART нормальную прошивку для ремонта.
20. Загрузчику не обязательно находится в начале Flash памяти. Загрузчик можно прописывать в самый последний сектор flash памяти, а приложение в начало Flash. Это позволит запускать приложение быстрее. Как загрузчик прыгает в приложение так и приложение может прыгать в загрузчик. Но тогда не будет работать пункт 19.

21. Сам загрузчик обычно записывают программатором по SWD или JTAG однако в приложении должна быть возможность записать другой загрузчик.
22. Исходя из соображений компактности в самом загрузчике должен быть только один интерфейс для обновления (DFU). Если гаджет - это автомобильный ECU, то интерфейс CAN, в остальных случаях часто самым компактным вариантом оказывается UART. Если в загрузчике вообще нет ни одного интерфейса, то появление кирпичей это вопрос буквально пары недель эксплуатации.
23. Загрузчик это не только ещё одна прошивка в репозитории. Для загрузчика нужна инфраструктура и экосистема. В самом простом виде это консольное Win приложение (FW Loader) под PC для отправки прошивки по serial COM порту. А для смузи-поколения надо сделать FW Loader с GUI-интерфейсом, потом загрузку из браузера Chrome, Opera, FireFox. Причем надо всё сделать под три операционки: Windows, Linux, Mac. Также нужно мобильное приложение для отправки прошивки из-под Android и iOS. И ещё нужен Web-сервер с авторизацией для поиска необновлённых электронных зубных щеток и раскатывания новых обновлений FW на них. Поэтому загрузчики это на самом деле очень-очень много работы.

10.6 Итоги

Как видите, разработка надежного универсального быстрого загрузчика это достаточно комплексная задача.

10.7 Гиперссылки

1. Удаленное обновление прошивки микроконтроллера
2. Безопасная и защищенная реализация бутлоадера
3. Микроконтроллер и Bootloader. Описание и принцип работы.
4. Использование загрузчика
5. OpenBLT
6. Загрузчик с шифрованием для STM32
7. Пишем загрузчик на Ассемблере и С. Часть 1
8. Пишем загрузчик на Ассемблере и С. Часть 2

Глава 11

Атрибуты Хорошего C-кода

"Код надо писать единообразно, безобразно."

(Владимир Романов)

"Хороший код, как кристалл - формируется годами."

11.1 Пролог

Этот текст адресован программистам на С(ях). Это не академические атрибуты из пыльных учебников, это скорее правила буравчика оформления сорцов, которые кристаллизовались на основе многолетней коммерческой разработки firmware. Некоторые приёмы случайно совпали с MISRA, некоторые с CERT-С. А остальное является результатом множества итераций инспекций программ и перестроек после реальных инцидентов. В общем, тут представлен обогащенный концентрат полезных практик программирования на С(ях).

11.2 Фундаментальные правила оформления Си-кода

Что-то покажется очевидным, но, тем не менее, именно это очевидное почти все и игнорируют. Но обо всём по порядку...

11.3 Про функции

1. Все функции должны быть менее 45 строк. Так каждая функция сможет уместиться на одном экране. Это позволит легко анализировать алгоритм и управлять модульностью кода. Также множество мелких функций удобнее покрывать модульными тестами.
2. У каждой функции должен быть только 1 return. Это позволит дописать какой-то дополнительный функционал в конце, зная, что он точно вызовется.
3. Все Си-функции должны всегда возвращать код ошибки. Минимум тип bool или числовой код ошибки. Так можно понять, где именно что-то пошло не так.

Проще говоря, не должно быть функций, которые возвращают `void`. Функции `void` это, по факту, бомбы с часовым механизмом. В один день они отработают ошибочно, а Вы об этом ничего даже не узнаете.

4. У каждой функции должна быть обратная функция. Например для функции `CAN_FD_SizeToDlc()` обратной функцией будет `CAN_FD_DlcToSize()`. Это правило пришло в программирование из математического анализа. Например у экспоненты обратная функция - логарифм. А в программировании обратные функции необходимы для модульных тестов написанного кода.
5. Если функция получает указатель, то пусть сразу проверяет на нуль значение указателя. Так прошивки не будут падать при получении нулевых указателей. Это повысит надежность кода. Вы же не знаете, как и кто этот код будет испытывать. Хорошая функция всегда проверяет то, что ей дают.
6. Каждой `set` функции должна быть поставлена в соответствие `get` функция. И наоборот. Это позволит написать модульный тест для данного параметра.
7. Использовать макрофункции препроцессора для кодогенерации одинаковых функций или пишите кодогенераторы, если препроцессор запрещён MISRA(ой). Копипаста - причина программных ошибок №1.
8. Скрывать область видимости локальных переменных по максимуму.
9. Все высокоуровневые функции в конец .c файла. Это избавит от необходимости указывать отдельно прототипы `static` функций. Например поэтому функции `xxx_init()` `xxx_proc()` должны быть вообще в самом конце файла. Если проводить аналогию из медицины, то это как центрифугирование крови. Си код тоже надо разделять на фракции!



Рис. 11.1: Даже у крови есть фракции

10. Использовать `static` функции везде, где только можно. Это повысит модульность. Причем `static` функции, как пузыри должны всплыть вверх Си-файла. Так они будут определены только в одном месте, что тоже уменьшит вероятность совершить ошибки.
11. Не вставлять функции внутрь оператора `if()`. Дело в том, что коды возврата приходится анализировать пошаговым GDB отладчиком до проверки условия. Понимаете?

ВОТ ЭТО ОЧЕНЬ ПЛОХО:

Листинг 11.1: Плохо

```
if (MmGet(ID_IPv4_ROLE, tmp, 1, &tmp_len) != MM_RET_CODE_OK) {  
    return ERROR_CODE_HARDWARE_FAULT;  
}
```

Надо писать код так, чтобы было возможно его проверять пошаговым gdb отладчиком. Поэтому каждое элементарное действие должно быть на отдельной строке. Вот так уже гораздо лучше.

Листинг 11.2: Вот так гораздо лучше

```
int ret = NVRAM_Get(ID_IPv4_ROLE, tmp, 1, &tmp_len);  
if (NVRAM_RET_CODE_OK != ret) {  
    return ERROR_CODE_HARDWARE_FAULT;  
}
```

12. В проекте не должно быть бесхозных функций, которые никто так или иначе не вызывает из main() во всей кодовой базе. Надо писать код по мере надобности. Код как кристалл должен произрастать из одной точки кристаллизации. Надо собирать Си-код вот с этими опциями компилятора.

11.4 Оформление переменных и констант

1. Всегда инициализировать локальные переменные в стеке. Иначе там просто будут случайные значения, которые могут что-нибудь повредить.
2. Давайте переменным осмысленные имена, чтобы было удобно grep(ать) по кодовой базе.
3. В идеале все переменные должны иметь разные имена. Так было бы очень удобно делать поиск по grep. Но тут надо искать компромисс с наглядностью.
4. Константы следует определять при помощи перечислений (enum) в большей степени, чем препроцессором. Так можно собрать константы из одной темы в одном месте и они не будут разбросаны по всему проекту. Ещё перечисления проверяются компилятором на совпадения, поэтому компилятор выдаст ошибку сборки, если перечисления из-за опечатки вдруг совпадают по значениям. Таким образом, перечисления делают код безопаснее. Это особенно важно для ответственных систем: автомобилестроение, авиастроение, атомная энергетика. Используйте перечисления.
5. Не допускать всяческих магических чисел в коде. Это уничтожает читаемость кода. Все константы надо определять в перечисления заглавными буквами в отдельном h файле для каждого программного компонента. Не используйте макро переменные для констант. Надо применять именно перечисления.

6. Если переменная это физическая величина, то в суффиксе указывать размерность (`timeout_ms`). Это увеличивает понятность кода.

11.5 Файлы

1. Все `*.c` файлы должны быть оснащены одноименным `*.h` файлом. Так эффективнее переносить, анализировать и мигрировать проекты на очередные аппаратные платформы. И сразу понятно, где следует искать прототипы функций из `*.c` файлов.
2. Для каждого программного компонента создавать несколько `*.c` и `*.h` файлов: Это позволит ориентироваться в коде и управлять модульностью.
3. Аппаратно-зависимый код должен быть отделен от аппаратно независимого кода по разным файлам и разным папкам. Так можно тестировать на другой архитектуре платформо-независимые функции и алгоритмы. Всякую математику, калькуляторы всяческих CRC(шек) и работу со строчками.
4. Все `*.h` файлы снабжать защитой препроцессора от повторного включения. Это же касается `*.mk` файлов.
5. `Include(ы)` всегда должны только содержать только название конечного файла. `Include(ы)` не должны содержать часть пути к файлу.

Листинг 11.3: Так нельзя

```
#include "C:/code_base/trunk/utils/data_types/cyclical_buff/cyclical_buff.h"
```

Вот так гораздо лучше

Листинг 11.4: Так нужно

```
#include "cyclical_buff.h"
```

Таким образом Вы сможете спокойно перетасовывать файлы в папках проекта и проект по-прежнему будет собираться. И визуально это много легче читать, поддерживать. А сами пути к заголовочным файлам надо передавать через опцию `-I` компилятора через `make` скрипты. В Си-коде же пути в подключаемом файле должны быть максимально короткими.

6. Используй препроцессорный `error` для предупреждения о нарушении зависимостей между компонентами.

```
#ifndef ADC_DEP_H
#define ADC_DEP_H

#ifndef HAS MCU
#error "+ HAS MCU"
#endif
```

```

#ifndef HAS_ADC
#error "+ HAS_ADC"
#endif

#ifndef HAS_GPIO
#error "+ HAS_GPIO"
#endif

#endif /* ADC_DEP_H */

```

7. Если что-то можно проверить на этапе make файлов, то это надо проверить на этапе make файлов. Каждый компонент должен проверять, что подключены нужные зависимости. Это можно сделать через условные операторы make файлов.

```

$(info I2S_MK_INC=$(I2S_MK_INC))
ifeq ($(I2S_MK_INC),Y)
    I2S_MK_INC=Y

    I2S_DIR = $(WORKSPACE_LOC) bsp/bsp_stm32f4/i2s
    #@echo $(error I2S_DIR=$(I2S_DIR))

    INCDIR += -I$(I2S_DIR)
    OPT += -DHAS_I2S

    SOURCES_C += $(I2S_DIR)/i2s_drv.c

    ifeq ($(DIAG),Y)
        SOURCES_C += $(I2S_DIR)/i2s_diag.c
    endif

    ifeq ($(CLI),Y)
        ifeq ($(I2S_COMMANDS),Y)
            OPT += -DHAS_I2S_COMMANDS
            SOURCES_C += $(I2S_DIR)/i2s_commands.c
        endif
    endif
endif

```

8. Тесты и код разделять на разные компоненты. То есть код и тесты должны быть в разных папках. Включаться и отключаться одной строчкой в make-файле.

11.6 Оформление кода

1. Основное правило: не делать сложно то, что можно сделать просто. Дело в том, что вероятность поломки программы возрастает с увеличением её сложности. Это универсальный принцип и работает везде: в механике, в электронике и в коде.

2. Избегайте бесконечных циклов while (1) при блокирующем ожидании чего-либо. Например ожидание прерывания по окончании отправки в UART. Прерывания могут и не произойти из-за сбоя. while (1) это просто капкан в программировании. Всегда должен быть предусмотрен аварийный механизм выхода по TimeOut(y) как тут.

Листинг 11.5: Выход по таймауту

```
bool UartSendWaitLl(uint8_t num,
                     uint8_t* tx_buffer,
                     uint16_t length) {
    bool res = false;
    // We send mainly from Stack. We need wait the end of transfer.
    UartHandle_t* Node = UartGetNode(num);
    if (Node && tx_buffer && length) {
        Node->uart_h->EVENTS_TXDRDY = 0;
        Node->uart_h->EVENTS_ENDTX = 0;

        Node->uart_h->TXD.PTR = (uint32_t)tx_buffer;
        Node->uart_h->TXD.MAXCNT = length;
        Node->uart_h->TASKS_STARTTX = 1;
        uint32_t start_ms = time_get_ms32();
        uint32_t cur_ms = time_get_ms32();
        while (!Node->uart_h->EVENTS_ENDTX) {
            cur_ms = time_get_ms32();
            uint32_t diff_ms = cur_ms - start_ms;
            if (UART_SEND_TIME_OUT_MS < diff_ms) {
                res = false;
                break;
            }
        }
        res = true;
    }
    return res;
}
```

3. Для синтаксического разбора регистров использовать объединения вкупе с битовыми полями.

Листинг 11.6: объединения вкупе с битовыми полями

```
/*Table 15. IB2-ADDR: I0000010*/
typedef union {
    uint8_t byte;
    struct{
        uint8_t clipping_information:1;
        uint8_t output_offset_information:1;
        uint8_t input_offset_information:1;
        uint8_t fault_information:1;
        uint8_t temperature_warning_information: 3;
        uint8_t res:1;
    };
} Fda801RegIb2Addr_t;
```

Это позволит делать парсинг полей буквально одной строчкой.

```
Fda801RegIb2Addr_t Reg;  
Reg.byte = reg_val;
```

4. Когда switch разрастается больше 45 строк, то надо делать статические LookUpTable-лы (LUTы). Элементом LUT(а) может являться указатель на функцию до 45 строк.

```
static const AdcChannelInfo_t AdcChannelInfoLut [] = {  
    {.code = ADC_CHANNEL_0, .adc_channel = ADC_CHAN_0},  
    {.code = ADC_CHANNEL_1, .adc_channel = ADC_CHAN_1},  
    ....  
    {.code = ADC_CHANNEL_999, .adc_channel = ADC_CHAN_999},  
};  
  
uint32_t AdcChannel2HalChan(AdcChannel_t adc_channel) {  
    uint32_t code = 0;  
    uint32_t i = 0;  
    for (i = 0; i < ARRAY_SIZE(AdcChannelInfoLut); i++) {  
        if (AdcChannelInfoLut[i].adc_channel == adc_channel) {  
            code = AdcChannelInfoLut[i].code;  
            break;  
        }  
    }  
    return code;  
}
```

5. В хорошем Си-коде в принципе не должно быть лишних комментариев. Никаких. Лучший комментарий к коду - это адекватные имена файлов, функций, названия переменных и модульные тесты. Поймите, сам язык Си - это и есть комментарий! Да... Мы же не на машинном коде программируем и даже не на ассемблере, а на Си. Си код это и есть комментарий для компилятора. Чтобы компилятор GCC понял, как собрать *.bin-архив. А любой ваш односстрочный или многострочный комментарий - это по факту комментарий про комментарий. Фильм о фильме. У этой матрёшки нет предела. Понимаете? В дурных компаниях доходит до того, что на инспекции кода обсуждают не код, а комментарии. И всё сторопрится на годы! Происходит застой и паралич всей разработки. Текстовый комментарий надо понимать, как информация в крайнем исключительном случае, а не как инфо-шум. Я говорю про случаи, когда применяется неочевидное решение. Понимаете? Можно добавить ссылку URL на задачу в Task Tracker-е в начале файла программного компонента. Можно минималистическим образом пояснить суть аргументов функции. Ключевое слово минималистическим. Но не более. Для простых функций, которые на один экран помещаются вообще можно не писать никаких комментариев. И так всё видно. Никаких украшательств, стразиков, заборов, инфантельных подписей ASCII графики и бантиков в Си-коде быть не должно в принципе! Много комментариев это фу! Прошивки микроконтроллеров - это вам не ласкунное одеяло, чтобы украшать его ASCII графикой. Прошивки микроконтроллеров - это чертёж алгоритма. Автора исходников и дату создания тоже надо удалять. Эти метаданные можно с успехом посмотреть в истории

коммитов в консоли git, SVN, Perforce и пр. Лучший коментарий к функции - это перечень её модульных тестов. Глядя на модульные тесты сразу понято, что надо подавать функции на вход и что ожидать на выходе. Модульные тесты - это подсказка коллегам особенно, когда кодовая база общая, когда одну и ту же функцию может использовать сегодня Тед, завтра Уолли. А сами модульные тесты на функцию успешно находятся по имени функции утилитой grep из корня репозитория. Следите за руками... Код -> grep -> модульные тесты -> понимание кода. Вот цепь и замкнулась.

6. Делать автоматическое форматирование отступов исходного кода. Подойдет например бесплатная утилита clang-format или GNUIndent. Это позволит делать простые выражения при поиске по коду утилитой grep. И будет минимальный diff при сравнении истории файлов. Придерживаться какого-нибудь одного стиля форматирования. Пусть будет "единообразно безобразно".
7. Если код не используется, то этот код не должен собираться. Это уменьшит размер артефактов. Уменьшит вероятность ошибок.
8. При сравнении переменных с константой константу ставьте слева от оператора ==. Когда константа на первом месте, то компилятор выдаст ошибки присвоение к константе в случае опечатки
9. В каждом if() всегда обрабатывать else вариант даже если else тривиальный. Это позволит предупредить многие осечки в программе.
10. За if(), for() ... всегда должны быть фигурные скобки { }. Весьма вероятно, что условие будет пополнено операторами.
11. Если есть конечный автомат, то добавить счетчик циклов. Так можно будет проверить, что автомат вообще вертится.
12. Не использовать операторы >, >= Вместо них использовать <, <= просто поменяв местами аргументы там, где это нужно. Это позволит интуитивно проще анализировать логику по коду. Человеку ещё со времен школьной математики понятнее, когда то, что слева - то меньше, а то, что справа - то больше. Так как ось X стрелкой показывала вправо. Особенно удобно при проверке переменной на принадлежность интервалу. Получается, что > и >= это вообще два бессмысленных оператора в языке С.
13. Если в коде есть список чего-либо (прототипы функций, макросы, перечисления), то эти строки должны быть отсортированы по алфавиту. Если сложно сортировать вручную, то можно прибегнуть к помощи консольной утилиты sort. Это позволит сделать визуальный бинарный поиск и найти нужную строчку. Также при сравнении 2-х отсортированных файлов отличия будут минимальные.
14. Функции CamelCase переменные snake_case. Чисто ради наглядности.

15. Если Вы определяете глобальную структуру, то указывайте имя полей. Так это продолжит работать, если кто-нибудь вдруг решится поменять порядок полей в определении структуре.

Вот LedConfig1 неправильно. Тут не ясно для чего нужна та или иная циферка.

Листинг 11.7: Плохо

```
//  
const LedConfig_t LedConfig1 [] = {  
    {LED_GREEN_ID, 1000, 0, 60, PORT_C, 13, "Green", LED_MODE_PWM, true},}  
};
```

При этом LedConfig2 выглядит намного привлекательнее. Тут сам код является комментарием к данным.

Листинг 11.8: Отлично

```
//  
const LedConfig_t LedConfig2 [] = {  
    {.num=LED_GREEN_ID, .period_ms=1000, .duty=60,  
     .pad={.port=PORT_C, .pin=13}, .phase_ms=0, .name="Green",  
     .mode=LED_MODE_PWM, .valid=true},  
};
```

16. (optional) Если Вы в Си передаете что-то через указатель или возвращаете через указатель, то указываете направление движения данных приставками `in`, `io` или `out` и показываете это ключевым словом `const`.

```
bool ProcSomeData(const uint8_t * const in_buffer,  
                  uint8_t * const out_buffer, int len, int * const out_len);
```

Это позволит легче читать прототипы, не погружаясь глубоко в тело самой Си-функции.

11.7 Оформление процесса конфигурации сборки и тестирования

1. В проекте сборке или программе обязательно должны быть модульные тесты. Модульные тесты - это просто функции, которые вызывают другие функции в run-time с константными аргументами. Это позволит сделать безболезненную перестройку кода, когда архитектура начнет скрипеть. Тесты можно вызывать как до запуска приложения, так и по команде из CLI.
2. Не дублировать конфиги. Каждый параметр должен конфигурироваться в одном месте программы. Если 2 структуры нуждаются в одинаковых конфигах, то это должно рассчитываться в run-time(e) исходя из главного корневого конфига.

3. Собирать артефакты как минимум двумя компиляторами (CCS + IAR) или (GCC+GHS) или (Clang+GCC) и тп. Если первый компилятор пропустил ошибку, то второй компилятор может и найти ошибку.
4. Чтобы обесопасить свой код я рекомендую собирать его вот с этими ключами компилятора. Компилятор сам укажет вам потенциально опасные места. Вам же останется только исправить найденные ошибки.

Листинг 11.9: опции компилятора для автоПоиска ошибок

```
-Wall -Werror=address -Werror=address -of -packed -member -Werror=array -bounds=1
-Werror=comment -Werror=div -by-zero -Werror=duplicate -decl -specifier
-Werror=duplicated -cond -Werror=empty-body -Werror=enum -compare
-Werror=float -equal -Werror=implicit -function -declaration
-Werror=implicit -int -Werror=incompatible -pointer -types
-Werror=int -conversion -Werror=logical -op -Werror=maybe -uninitialized
-Werror=misleading -indentation -Werror=missing -declarations
-Werror=missing -field -initializers -Werror=missing -parameter -type
-Werror=missing -prototypes -Werror=old -style -declaration
-Werror=overflow -Werror=parentheses -Werror=pointer -sign
-Werror=redundant -decls -Werror=return -type -Werror=shadow
-Werror=shift -count -overflow -Werror=shift -negative -value
-Werror=sizeof -pointer -div -Werror=strict -prototypes -Werror=switch
-Werror=type -limits -Werror=uninitialized -Werror=unused -but -set -parameter
-Werror=unused -but -set -variable -Werror=unused -but -set -variable
-Werror=unused -but -set -variable -Werror=unused -function -Werror=unused -variable
-Werror=unused -variable -Wmissing -field -initializers -Wmissing -prototypes
-Wno-cpp -Wno-discard -qualifiers -Wno-format -Wno-format -truncation
-Wno-nnonnull -compare -Wno-restrict -Wno-stringop -truncation -fdata -sections
-fdce -fdse -ffunction -sections -finline -small -functions -fmessage -length
=0
-fno-common -fomit -frame -pointer -fshort -enums -fsigned -char -fstack -usage
-fzero -initialized -in -bss
```

5. На все сборки должна быть одна общая кодовая база. Называют его по-разному: общак, репа, копилка. Модификация в одном компоненте должна отражаться на всех сборках организации, использующих компонент (например алгоритмы CRC). Это позволит сэкономить время на создание новых проектов для новых программ.
6. Соблюдать программную иерархичность. Низкоуровневый модуль не должен управлять (вызывать функции) более высокоуровневого модуля. UART не должен вызывать функции LOG. И компонент LOG не должен вызывать функции CLI. Управление должно быть направлено в сторону от более высокоуровневого компонента к более низкоуровневому компоненту. Например CLI->LOG->UART. Не наоборот.
7. Сборка из Makefile(ов) является предпочтительнее, чем сборка из GUI-IDE. Make позволяет по-полней управлять модульностью кодовой базы. Одним символом включать или исключать сотни файлов из сотен сборок. Если Вы

собираете сорцы из Makefile, то Вы можете инициировать сборки прямо из командной строки. А это значит, что процесс сборки можно автоматизировать. Прикреплять в Jenkins. А утром контролировать результаты сотен сборок, что отработали ночью. Производительность работы с Makefile выше на порядки. В случае с IDE вам придется вручную водить мышку, чтобы стартонуть сборку. А если что-то случилось с версией IDE, то Вы вообще не сможете запустить сборку. IDE это форма технологического диктата со стороны вендора IDE (IAR, KEIL). Они там будут решать кому можно, а кому нельзя собирать исходники. Вам оно надо?

8. Прогонять кодовую базу через статический анализатор. Хотя бы бесплатный CppCheck. Может, найдется очередная загвоздка.
9. Если что-то можно проверить на этапе препроцессора, то это надо проверить на этапе препроцессора. Каждый компонент должен проверять, что подключены нужные зависимости. Это можно сделать через макросы компонентов.
10. Если что-то можно проверить на этапе компиляции, то это надо проверить на этапе компиляции (static_assert(ы)). Например можно проверить, что в конфигурациях скорость UART не равна нулю. В RunTime не должно быть проверок, которые можно произвести на этапе компиляции, препроцессора или make файлов.

11.8 Аномалии оформления сорцов из реальной жизни (War Stories)

1. Функции от 1000 до 5000 строк и даже более.
2. Определять препроцессором множество констант для одной и той же переменной. Вместо того чтобы использовать перечисления.
3. Переиспользование глобальных переменных.
4. Магические циферки на каждой строчке.
5. Доступ к регистрам микроконтроллера в каждом файле проекта.
6. Повторяемость кода.
7. Очевидные комментарии.
8. "заборы" из комментариев (например ////////////////)
9. Бесхозные функции, которые никто не вызывает во всём проекте.
10. *.c файлы оснащены разноименными *.h файлами.
11. Статические прототипы функций в *.h файлах.

12. Макросы маленькими буквами.
13. Код без модульных тестов.
14. Код, как в миксере перемешанный с модульными тестами.
15. Вставка препроцессором include *.c файлов. (Это просто полнейшая школота).
16. Вся прошивка в одном main.c файле 75000 строк аж подвисает текстовый редактор.
17. Длинные пути к файлам в includ(ax) (начиная с корня диска C:)
18. С-функции с именами литературных персонажей.

11.9 Финальная структура добротного программного компонента

Для каждого программного компонента SWC надо создавать несколько файлов разных типов: *.c ; *.h; *.mk; *.cmake и пр. Этот список представлен в порядке возрастания уровня вашей экспертизы как программиста.

1. *.h файл прототипов функций самого SWC.
2. *.c файл самого SWC. Функционал и бизнес логика.
3. test_xxx.h файл прототипов функций с модульными тестами.
4. test_xxx.c файл с модульными тестами.
5. xxx.mk файл скрип сборки для GNU make
6. xxx_const.h файл с определением констант для данного SWC.
7. xxx_types.h файл с определением типов данных для данного SWC.
8. xxx_nvram.h файл с определением энергонезависимых параметров NVRAM для данного программного компонента (SWC)
9. xxx_commands.h файл прототипов функций команд CLI.
10. xxx_commands.c файл команд CLI для данного SWC.
11. config_xxx.h файл с определением конфигурационной структуры по умолчанию.
12. config_xxx.c файл с массивом конфигурационных структур по умолчанию.
13. xxx_diag.h файл прототипов функций диагностики.

14. xxx_diag.c файл с определением функций диагностики.
15. xxx_isr.c файл код обработчика прерываний
16. xxx_isr.h файл код обработчика прерываний
17. xxx_preconfig.mk файл для установки нужных переменных окружения зависимостей для данного SWC.
18. xxx_dep.h файл проверки зависимостей на фазе препроцессора.
19. xxx.gvi файл перечень зафисимостей для graphviz. Это позволит вам потом автоматически составить график зависимостей и вычислить правильную последовательность инициализации всей программы.
20. xxx.cmake файл скрипт сборки для CMake
21. Kconfig файл конфигурации для системы KConfig. Это на тот случай, если Вы вдруг захотите вклюить свой код в Zephyr Project или в up stream ядра Linux.

Как видите, список получился будь здоров... Это позволит проще ориентироваться в коде и управлять модульностью. Зато именно такое разбиение SWC позволит вам добиться максимальной переносимости программного компонента между аппаратными платформами с разными ресурсами: AVR, ARC, ARM, xTensia, Power PC или x86.

11.10 Итог

Общая мысль такова, что надо писать С-код по таким понятиям как простота, тестопригодность, поддерживаемость, ремонтопригодность, модульность, согласованность (принцип наименьшего удивления), масштабируемость, иерархичность, конфигурируемость, изоляция компонентов и переносимость.

В программировании надо придерживаться принципа: "Одно действие - одна строчка".

Если Вы программируете на С(ях) микроконтроллеры, то можно ещё добавить внимание на то, что надо делать UART-CLI для отладки и верификации прошивки в run-time(е), добавлять встроенные модульные тесты, собирать из самописных Makefile(ов) и всё у вас будет очень даже модульно, масштабируемо и гибко.

Хороший код он, как кристалл формируется годами. Надо сотни и сотни итераций пере сборки на разных архитектурах, чтобы понять как оно надо.

11.11 Источники

1. Сайт с пояснением назначения функций в Си

2. Архитектура хорошего программного компонента. Электронная таблица.
3. Архитектура Хорошо Драйвера
4. Пиши на С как джентльмен
5. The Power of Ten–Rules for Developing Safety Critical Code

Глава 12

Что Должно Быть в Каждом FirmWare Репозитории

"У нас аккуратный, чистый репозиторий и моя задача, чтобы он таким и оставался."

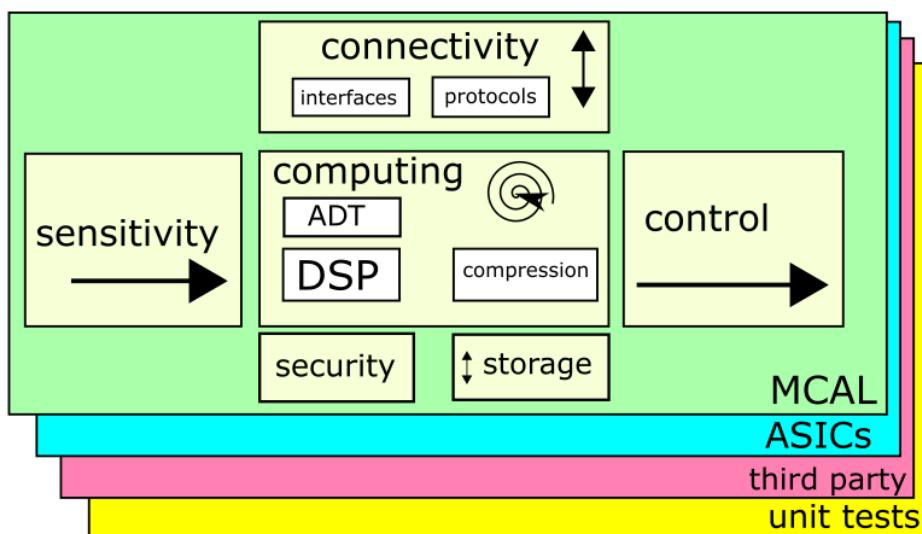


Рис. 12.1: Структура хорошего репозитория

12.1 Пролог

В этом тексте я предлагаю порассуждать, что же должно быть в нормальном взрослом firmware репозитории (репе/общаке) безотносительно к конкретному проекту. То есть самые универсальные и переносимые программные компоненты (кирпичики/SubSystems), которые могут пригодиться в практически любой сборке.

12.2 Компонент для поддержки каждого конкретного процессорного ядра (Core)

Для каждого конкретного процессорного ядра (Cortex-M3/Cortex-M33/PowerPC/Tensia Xtensa и пр.) должна быть папка с сорцами описания этого ядра. Это функции вкл/откл прерываний, проверка прерывание ли сейчас, перезагрузка ядра, печать таблицы прерываний, управление системным таймером и прочее.

12.3 Компонент для поддержки каждого конкретного микроконтроллера (microcontroller)

Для каждого конкретного MCU должна быть папка с описанием этого MCU. Это прежде всего перечень пинов, описание ядра, памяти и доступной периферии.

12.4 Компонент для поддержки каждой конкретной платы (platform code,boards)

Для каждой платы должна быть создана отдельная папка в репозитории. Там должны быть исходники конфигурации, которые говорят сколько в этой конкретной плате кнопок, LED(ов), SPI, I2C, SDIO. Какие там есть драйверы чипов и прочее.

12.5 Отдельная папка для каждой конкретной сборки (projects)

В репозитории должна быть папка projects. В ней должны лежать корневые директории для каждого отдельного проекта. Все сборки должны делить общую кодовую базу. Поэтому в папке с проектом (projects->board_name_bootloader_m) в принципе не должно быть *.c файлов. Максимум один Makefile, парочка конфигов в *.h, и ещё несколько make скриптов: config.mk, cli_config.mk, diag_config.mk.

12.6 Абстрактные структуры данных (ADT)

В репозитории должна быть реализация самых базовых абстрактных структур данных. Вот они перед вами.

1. Программный компонент работы с массивами Array
2. Циклический массив

Циклический массив определено понадобится для реализации цифровых фильтров, энергонезависимых журналов.

3. Универсальная FIFO (очередь)

В любом Firmware проекте точно понадобится надежная FIFO(шка). Например, FIFO нужна для предварительного хранения данных приходящих из UART Rx, для хранения кнопок с HID клавиатуры, для хранения принятых пакетов из LoRa или TCP. FIFO нужна для того же CLI. Причем реализация FIFOшки должна подстраиваться под любой тип данных: char, uint16_t, array[N].

4. HashSet.

HashSet также понадобится для разбора XML как и LIFO.

5. Универсальная LIFO (стек)

Если ваша прошивка парсит XML файл с конфигами на SD карте, то вам точно понадобится реализация LIFO.

6. Связанный список

Linked List в прошивке может вам понадобится для организации списка команд CLI.

7. Set.

8. Синтаксический разбор строчек (String).

При реализации CLI нужны функции для синтаксического разбора чисел всех типов данных из текстовых ASCII строк. Поэтому для каждого типа данных (float, int32_t, string, hexArray) следует реализовать парсеры типов данных. Своего рода интерпретаторы чисел. Это очень большая часть кода порядка 2k строк на чистом Си.

12.7 Загрузчик (BootLoader)

Загрузчик нужен для обновления прошивки без специализированного оборудования типа программаторов. Это очень важно для пользователей. Загрузчик обязательно должен уметь обновлять по UART так как драйвер UART занимает мизерное количество On-Chip NorFlash памяти. Остальные интерфейсы обновления по обстоятельствам или поддержка этих тяжеловесных интерфейсов (CAN, USB, LoRa, TCP, WiFi) должна быть в приложении так как там больше Flash памяти. Задача минимального загрузчика это найти в NorFlash приложение и прыгнуть в него. А если приложения нет, то просить прошивку в UART, чтобы совсем в тыкву не превращаться. Тем более, что переходники с USB-UART (например на основе чипа CP2102) самые дешевые из всех возможных интерфейсов. Всего порядка 6 EUR.

12.8 MicroController Abstraction Layer (MCAL)

MicroController Abstraction Layer для каждого семейства микроконтроллеров. Это пожалуй самая большая папка в репозитории. Тут будет очень много кода. Однако он весь однообразный.

Кодовая база должна быть универсальной так в AUTOSAR или Zephyr, однако для каждого семейства микроконтроллеров будет своя уникальная реализация MCAL.

В MCAL надо прописать реализации универсального переносимого крос платформенного API, на основе которого вы и будете строить своё микроконтроллерное приложение. MCAL код нужен, например, для управления GPIO, прописи on-chip FLASH, запуска аппаратных таймеров, пульсации пакетов в I2C,SPI,I2S,UART, вычитывания ADC и прочего. Если вендор поставляет свой HAL, то вы должны написать Binding(и) для этого HAL чтобы вызывать его из универсальных функций MCAL. Понимаете?

Что обычно лежит в папке MCAL? В папке MCAL лежат драйвера аппаратных подсистем микроконтроллера: ADC, CAN, Clock, DAC, DMA, Ext interrupts, FLASH, GPIO, I2C, I2S, Input capture, Interrupts, IO Mux, NVS, PDM, PWM, RTC, SDIO, SPI, SWD, Timer, UART, USB, WatchDog

Это всё то, что имеет аппаратную реализацию внутри MCU. Для пуска и управления аппаратными подсистемами микроконтроллера надо создавать в репозитории отдельную папку для каждой подсистемы. Причем прототипы функций у всех микроконтроллеров должны быть одинаковые. Разное только тело функции. Понимаете?

Для каждого семейства MCU будет отдельная папка с реализацией binding-ов к MCAL.

1. mcal_common. В папке MCAL_common лежат папки MCAL (GPIO, UART, SPI и прочие) только с прототипами функций и общий код. Плюс много weak функций.
2. mcal_amur.
3. mcal_at32f4.
4. mcal_autozar.
5. mcal_cc26x2.
6. mcal_embox.
7. mcal_esp32.
8. mcal_mdr32.
9. mcal_nrf5340.
10. mcal_opencm3.

11. mcal_stm32F4.
12. mcal_spl.
13. mcal_spc58nn.
14. mcal_yuntu.
15. mcal_zephyr.
16. mcal_some_company_internal_hal.
17. и так далее

MCAL тут выступает как proxy (доверенный уполномоченны код) между вашим приложением и кодом от производителя микроконтроллера. Это необходимо, чтобы не менять код от вендора и не брать на себя ответственность за чужие кояски. Оно Вам надо?

Также binding-и к mcal-у нужны для миграции и переносимости бизнес логики прошивки с одного микроконтроллера на совершенно другой микроконтроллер. Такое часто происходит банально потому, что любой микроконтроллер рано или поздно окажется просто недоступным для установки в очередные партии приборов. Происходит это по разным причинам:

1. Ваш прежний MCU сняли с производства и его больше просто не купить.
2. В компании решили поставить другой МК так как он дешевле.
3. В компании решили поставить другой МК так как он производительнее при той же цене.
4. В компании решили поставить другой МК так как он технологичнее.
5. Ваша страна попала под санкции и Вам перестали продавать Ваш прошный микроконтроллер.
6. прочее

Поэтому это совершенно нормально, когда по мере жизненного цикла продукта микроконтроллеры внутри него меняются, как перчатки.

А все нестыковки между приложением и аппаратурой при подмене микроконтроллера разрулит MCAL код. Поэтому в репозитории и должна быть папка mcal.

12.9 Драйверы периферийных чипов (ASICs)

Драйверы периферийных чипов с управлением по I2C/SPI/MDIO. В каждой электронной плате есть множество умных навороченных чипов с конфигурацией по I2C/SPI/MDIO, например DS3231. Я видел ASIC чип у которого 936x 32-битных регистров конфигурации. В репозитории должна быть папка "asics" для складирования драйвов для каждого такого чипа и отдельная папка с модульными тестами для чипа. Напишите в комментариях драйверы каких умных SPI/I2C/MDIO чипов писали Вы.

12.10 Папка со всем оставшимся (miscellaneous)

В эту папку надо складывать те компоненты, которые сложно классифицировать во что-то конкретное. Функций Base64 кодека, функции реверса порядка слов/байт/бит, функции swap для всех типов данных, сравнение float(ов) и прочее. Всё это как правило пригождается в разработке firmware от проекта к проекту. В идеале папка miscellaneous должна быть пустая.

12.11 Чужой код (Third Party)

Наверняка в вашей прошивке будет код от других Open Source проектов. Это может быть FreeRTOS, LwIP, FatFs, CMSIS. Этот код тоже надо подвергнуть версионному контролю. Однако ни в коем случае не надо этот код менять, как бы плохо он не был бы написан. Иначе Вы автоматически станете владельцем этого кода и понесете за него ответственность. Оно Вам надо? Очень важно также не переформатировать этот third party код, как как это позволит делать сравнение используемого кода с новой его версией от официального вендора.

12.12 Sensitivity

Sensitivity - это всё то, что связано с вводом информации в программу. В репозитории должна быть прям отдельная папка sensitivity.

1. Программный Таймер

Программный таймер- это способ из одного аппаратного таймера сделать сотни программных таймеров. Очень полезно, если все аппаратные таймеры исчерпаны или их настройка на неизвестном SoC(e) очень трудна и непонятна. Хороший программный таймер позволяет настраивать период и фазу счета и полностью конфигурируется в run-time из CLI.

2. Драйвер подавление дребезга контактов

Часто в электронных платах есть тактовые кнопки и герконы. Они при замыкании создают дребезг. Поэтому в репозитории должен быть драйвер кнопки, который подавляет дребезг

контактов, умеет отличать короткое нажатие от длинного, двойного. Всё это пригодится при отладке.

3. Драйвер опроса кнопок
4. Универсальный API для обобщенного акселерометра
5. Универсальный API для обобщенного датчика света
6. Универсальный API для FM tuner-a
7. Универсальный API для RTC
8. Программный компонент измерения точного времени
9. Программный компонент контроля качества пайки
10. и тому подобное

12.13 Computing

Рассмотрим папку computing. В папке computing следует складировать все компоненты, которые так или иначе связаны с вычислением чего-либо. Каждая прошивка так или иначе что-то считает. Поэтому в репозитории должна быть папка computing. Немного линейной алгебры и численных методов. Еще может понадобиться целочисленное возведение в степень и вычисление квадратного корня. Бывает надо 7-битное знаковое число или 13-битное знаковое число преобразовать в стандартный int32_t. Для этого тоже нужны свои математические алгоритмы.

1. триггер Шмитта. Это звено для реализации гистерезиса. Очень полезно для подавления аналогового шума с датчиков при релейном управлении чем-либо.
2. SHA256
3. Топологическая сортировка
4. SDR обработка чего-либо
5. обработчики интервалов
6. арифметика над GNSS координатами. Она же сферическая геометрия Римана.
7. Компонент limiter Это функция, которая следит за тем, чтобы конкретная функция не вызывалась чаще, чем установлено в параметрах. Очень полезно при реализации примитивов RTOS. Можно прямо в суперцикле пропускать все вызовы через limiter и таким образом выставлять период срабатывания каждой функции.

8. расчет делителя непряжения
9. расчет аналогового фильтра
10. расчет предделителей аппаратных таймеров
11. расчет семплов delta sigma модулятора
12. реализация методов DSP обработки
13. решатели уравнений
14. астрономические вычислители
15. генераторы QR кодов
16. Векторная алгебра В системах автоматического управления на MCU очень нужна функция для вычисления угла между векторами (с учетом знака, естественно). А это сразу подтягивает реализацию скалярного и векторного умножения.
17. Вычислители контрольных сумм CRC При реализации протоколов понадобится целая куча разнообразных контрольных сумм (CRC8, CRC16, CRC24, CRC32). Также CRC нужны для энергонезависимых файловых систем. Сюда же относятся алгоритмы Hash функций (например SHA256) и цифровых подписей.
18. Программная реализация календаря Если в проекте предусмотрены часы реального времени, на плате есть кварц и батарейка, а аппаратные часы внутри MCU могут только увеличивать каждую секунду счетчик, то придется найти и протестировать надежный программный календарь. Даешь количество секунд, получаешь дату и время и наоборот.
19. Компонент компрессии-декомпрессии. Может случиться, что битовая скорость трансивера очень низкая (LoRa, BLE), а данных надо передавать много. В этом случае может спасти сжатие или компрессия данных (например кодек LC3, SBC или GZip) и декомпрессия на стороне приемника. В результате должен быть надежный и протестированный программный компонент кодек сжатия данных.
20. Direct digital synthesis (DDS) При работе с ЦАП(иками) и DAC(ами) приходится вычисление семпла синуса, семпла PWM, Chirp(a), Saw(a), Fence(a). Такие вычисления производит программный компонент DDS.
21. Цифровые фильтры. Цифровые фильтры (ЦФ) нужны не только для обработки аудиопотока и радарных данных. ЦФ понадобится для реализации подавления дребезга контактов, для вычисления направлений движения RFID маяков, для демодуляции BPSK модуляции в SDR обработке и прочего.

Всё это добро сохраним в папку computing. Какую математику приходилось реализовывать в микроконтроллерах вам?

12.14 Connectivity

Connectivity, в свою очередь, можно разбить на три папки внутри: log, interfaces и protocols.

1. Компонент управления логированием (log)

Должен быть программный компонент для управления логированием в UART (или на SD карту). Раскраска логов в TeraTerm или Putty, выбор или отмена TimeStamp выбор отмена логирования для конкретного компонента. Это поможет анализировать лог загрузки устройства и следить за событиями внутри прошивки в run-time-е, просто глядя в UART. Обычно такой программный компонент называют log.

2. interfaces

В папке interfaces надо содержать программные компоненты, которые реализуют всяческие нетрифициальные интерфейсы.

- (a) CAN
- (b) DTMF
- (c) BPSK
- (d) 1-WIRE
- (e) RS485
- (f) LoRa
- (g) RS232
- (h) UWB
- (i) Sockets

3. protocols

- (a) Синтаксический разбор строчек При реализации CLI нужны функции для синтаксического разбора чисел всех типов данных из текстовых ASCII строк. Поэтому для каждого типа данных (float, int32_t, string, hexArray) следует реализовать парсеры типов данных. Своего рода интерпретаторы чисел. Это очень большая часть кода порядка 2k строк.
- (b) Command Line Interface (CLI) CLI (Command Line Interface) он же Shell он же консоль он же TUI (Text User Interface)
CLI(шка) обязательный компонент для отладки и тестирования прошивок. С этим компонентом можно общаться с устройством на человеческом языке. Многие успешные продукты обладают CLI (российский Flipper-Zero, китайский NanoVNA V2, швейцарский U-Blox ODIN C099-F9P, канадский Bluetooth модуль BC127 от Sierra Wireless).

- (c) base64
- (d) iso-tp
- (e) nmea
- (f) rds
- (g) RTCM3
- (h) UDS
- (i) UBX
- (j) DS-TWR

12.15 Storage

1. NVRAM: или энергонезависимая Key-Value-Мар(ка) В любой крупной Си программе накопится очень много параметров и констант, которые придется варъировать, настраивать, калибровать. Чтобы не пере собирать и пере прошивать каждый раз гаджет, надо реализовать энергонезависимый журнал прямо на Target(e). Вот вам War Story. Устройство висит под потолком. Подключился через LoRa к CLI. Прописал новый параметр через CLI, reset(нул) прошивку через CLI и у тебя новая прошивка. Easy.
2. Реализация механизма выделения и освобождения памяти. Весьма вероятно, что придется накропать собственную надежную детерминированную и переносимую версию функций malloc и free, с возможностью расширенной диагностики и сборки мусора.

12.16 Код для информационной безопасности (Security)

В репозитории должна быть папка с программными компонентами информационной безопасности.

1. Шифровальщик AES256 Рано или поздно передаваемую прошивку по загрузчику придется шифровать или хранить в памяти в шифрованном виде. Поэтому надо выбрать доверенный и протестированный алгоритм шифрования данных, например AES256. В TI чипах cc26x2 AES и вовсе реализован аппаратно, и этот компонент можно отнести вообще в BSP.
2. KeePass
3. CRC
4. salsa20
5. и прочее

12.17 Модульные тесты

Вся работа пойдет прахом, если ошибка в очередном коммите тихо останется незамеченной и поломает бизнес логику. Грош цена репозиторию без тестов! Код без модульных тестов Филькина грамота. Поэтому надо покрывать код модульными тестами. Каждая нетривиальная функция должна быть покрыта тестами. Аппаратно-независимый код можно вообще тестировать прямо на LapTop PC отдельным процессом.

Поэтому в репозитории должна быть папка с модульными тестами.

12.18 Сборка из Make

Утилита Make это самый гибкий способ управлять циклическими репозиториями. Можно одной строчкой добавлять или исключать сотни файлов для сотен сборок. Поэтому для каждой сборки надо самим писать Makefile для каждого компонента *.mk файл. Сборка из Make стимулирует придерживаться модульности и изоляции компонентов друг от друга. Make идеален для масштабирования кодовой базы.

Makefile(лы) хороши тем, что можно добавить кучу проверок зависимостей и assert(ов) на фазе отработки bash-скриптов прямо в *.mk файлах, ещё до компиляции самого кода! Так как язык программирования make поддерживает условные операторы и функции! Можно очень много ошибок отловить на этапе отработки утилиты make. При этом make очень прост. Вся спека GNU Make это 226 страниц. Это не CMake со своими тысячами и тысячами страниц мануала. Стю Фельдман (автор Make) просто гений.

12.19 Скрипты сборки CMake

Рано или поздно вам придется интегрировать свой код в другие репозитории. Вероятно придется собирать компанейский репозиторий в составе Cmake проектов. Например в Zephyr Project(e) или для микроконтроллеров ESP32. Поэтому каждый программный компонент должен содержать также *.cmake скрипт с правилами сборки этого программного компонента.

12.20 Скрипты автосборки

Каждая сборка должна собираться из скриптов. Скрипты должны записывать в On-Chip Flash hash последнего коммита и название GIT ветки. Это потом позволит откатиться назад и выявить причину бага в конкретный момент истории.

Буквально открыл папку с проектом, жмакнул *.bat файл и максимум через 2 минуты получил полный комплект артефактов *.hex *.bin *.elf *.map. Easy! Также сборка из скриптов позволит вам добавить сборки на сервер сборки Jenkins и каждое утро следить за тем, что собирается, а что нет.

12.21 Скрипты авто прошивки

Должна быть возможность автоматически прошивать Target. Буквально жмакнул скрипт flash.bat из папки с проектом и скормил прошивку микроконтроллеру. И еще и лог обновления сохранился в текстовый файлик.

Этот же скрипт с артефактами можно будет отгрузить клиенту, так как установку IDE для обновления прошивки от user(ов) какого-н фитнес браслета ожидать точно не стоит.

12.22 Скрипты отчистки и автоматического форматирования

Это для причесывания кода утилитой clang-format. Чтобы отступы были предсказуемыми по всему проекту. Это позволит писать более простые регулярные выражения для навигации по коду утилитой grep.

Batch скрипт авто очистки в корне репозитория позволит удалить временные файлы с расширениями *.d *.o *.obj *.bak *.i *.pp и высвободить тонну места на диске. Тем более, что у вас на диске будет как минимум 2 экземпляра репозитория: workspace для работы и release для сервера сборки типа Jenkins(a).

12.23 Статические библиотеки

Некоторые программные компоненты распространяются не в виде исходников, а в виде предварительно скомпилированных библиотек (например кодек lc3). Поэтому их *.a файлы надо тоже подвергать версионному контролю.

12.24 Кодогенераторы

В программировании микроконтроллеров часто много повторяющегося по структуре кода. Например синтаксический разбор CAN матриц (8 байт полезных данных в пакете). Поэтому в репозиторий можно добавить утилиты-кодогенераторы для генерации C-функций делающих синтаксический разбор пакетов с известной структурой.

12.25 Авто генерация документации

Надо относится как программированию не только к созданию артефактов. Надо относится как к программированию также для создания документации. При разработке Firmware документацией является схема toolchain, блок схемы плат, блок схемы комплексов, инструкции. Все эти *.pdf, *.svg можно синтезировать из кода на языке dot или LaTeX. Поэтому должны быть makefile(ы) для сборки

документации. Прекрасно выровненные текстовые документы можно авто генерировать на языке LaTeX.

12.26 Какие папки должны быть в репозитории?

Код любой прошивки можно условно разделить на 5 фракций. Это sensitivity, connectivity, computing, storage, security и control. Поэтому для каждого понятия должна быть папка в репозитории.

Чтобы у вас не получился венигет из программных компонентов.

12.27 Итог

Суммируя вышесказанное, надо просто строить репозиторий так, чтобы он позволял работать с абсолютно любыми микроконтроллерами.

Надо разделить код на аппаратно-зависимый, аппаратно не зависимый и документацию. Плюс часть инфраструктурных сборок для кодогенераторов и модульных тестов на desktop.

Это тот самый минимальный джентельменский набор, который просто необходим, чтобы банально начать что-то делать на микроконтроллерах. Без этих базовых компонентов Вы столкнетесь с неимоверными и тектоническими трудностями при миграции, масштабировании, переносе и интеграции кода.

Когда у вас один репозиторий на много сборок, то вы располагаете огромным количеством образцовых проектов для создания новых.

Переносимая кодовая база - это ваш плацдарм для будущих разработок. Понимаете?

Глава 13

Архитектура Хорошо Поддерживаемого драйвера

"Надо сделать не то, что они просят, а то, что им нужно!"
(Александр Сергеевич Яковлев)

13.1 Пролог

Итак, вам дали плату, а в ней 4 навороченных умных периферийных чипа с собственными внутренними конфигурационными по SPI/I2C регистрами. Это могут быть такие чипы, как lis3dh, at24cxx, si4703, ksz8081, sx1262, wm8731, tcan4550, fda801, tic12400, ssd1306, dw1000, или drv8711. Не важно, какой конкретно чип. Все они работают по одному принципу. Прописываешь по проводному интерфейсу чиселки в их внутренние регистры и там внутри чипа заводится какая-то электрическая цепочка. Easy.

Допустим, что на GitHub драйверов для вашего I2C,SPI чипа нет или качество этих open source драйверов оставляет желать лучшего. Как же оформить и собрать качественный драйвер для I2C,SPI,MDIO чипа? И почему это вообще важно?

Смоки, тут не Вьетнам, это боулинг, здесь есть правила.

Понятное дело, что нужно, чтобы драйвер был модульным, поддерживаемым, тесто-пригодным, диагностируемым, понятным. Прежде всего надо понять, как организовать структуру файлов с драйвером. Это можно сделать так:

13.2 xxx_drv.c, xxx_drv.h с функционалом

Должна быть функция инициализации, обработчика в цикле, проверка Link(a), функции чтения и записи регистра по адресу. Плюс набор высокоуровневых функций для установки и чтения конкретных параметров. Это тот минимум минимумов, на котором большинство разработчиков складывает руки. Далее следует материал уровня advanced.

13.3 xxx_isr.c, xxx_isr.h

Отдельные файлы с кодом драйвера, который должен отрабатывать в обработчике прерываний

Это нужно для того, чтобы подчеркнуть тот факт, что к этому ISR коду надо относиться с особенной осторожностью. Например, это ядро программного таймера.

Код работающий внутри обработчика прерывания должен обладать следующими свойствами:

1. Должен быть оптимизирован по быстродействию.
2. Этот код сам не должен вызывать другие прерывания.
3. Надо убедиться, что там нет арифметики с плавающей точкой. Иначе это тоже будет медленно выходить из контекста.
4. Там нет логирования.
5. Все переменные, которые модифицируются внутри прерывания помечены как volatile.

13.4 Файл xxx_types.h с типами

Отдельный xxx_types.h файл с перечислением типов. В этом файле следует определить основные типы данных для данного программного компонента. Также определить объединения и битовые поля для каждого регистра.

Листинг 13.1: Битовое поле для регистра

```
/* page 105
7.2.27 Register file: 0x19      DW1000 State Information*/
typedef union {
    uint8_t byte[4];
    uint32_t dword;
    struct {
        uint32_t tx_state : 4;      /* bit 0-3: TX_STATE*/
        uint32_t res1 : 4;         /* bit 4-7: */
        uint32_t rx_state: 5;     /* bit 8-12: RX_STATE*/
        uint32_t res2 : 3;         /* bit 13-15: */
        uint32_t pmsc_state : 4;   /* bit 16-19: PMSC_STATE*/
        uint32_t res3 : 12;        /* bit 20-31: */
    };
} Dwm1000SysState_t;
```

Делать отдельный *.h файл для перечисления типов данных это не блажь. Хранить код и данные в разных местах - это основной принцип гарвардской архитектуры компьютеров. Этого же принципа логично придерживаться и в разработке кода.

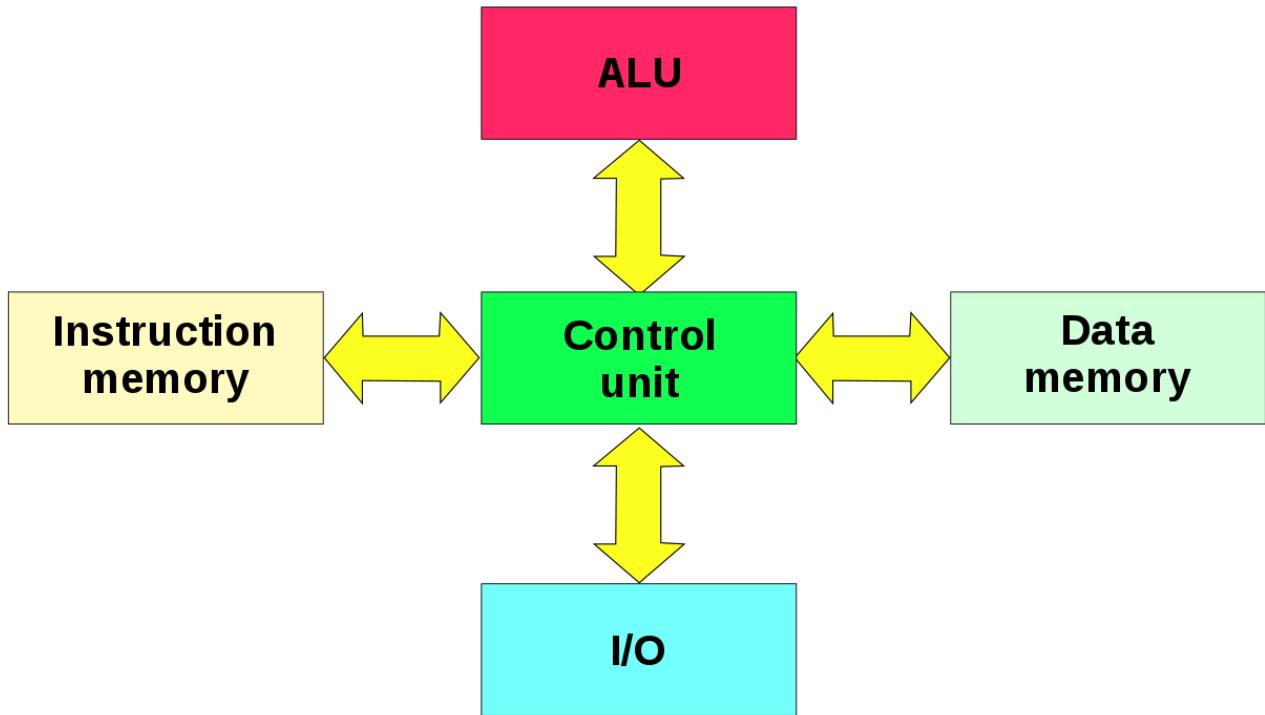


Рис. 13.1: Harvard architecture

13.5 Отдельный xxx_const.h файл с перечислением констант

Отдельный xxx_const.h файл с перечислением констант Тут надо определить адреса регистров, перечисления. Это очень важно быстро найти файл с константами и отредактировать их, поэтому для констант делаем отдельный *.h файл.

Листинг 13.2: Перечисление констант

```

typedef enum {
    BIT_RATE_110_KBPS = 0, /*110 kbps*/
    BIT_RATE_850_KBPS = 1, /*850 kbps*/
    BIT_RATE_6800_KBPS = 2, /*6.8 Mbps*/
    BIT_RATE_RESERVED = 3, /*reserved*/
    BIT_RATE_UNFED = 4,
} Dwm1000BitRate_t;

```

13.6 xxx_param.h файл с параметрами драйвера

xxx_param.h файл с параметрами драйвера Каждый драйвер нуждается в энергонезависимых параметрах с настройками (битовая скорость, CAN-трансивера или несущая частота радиопередатчика). Именно эти настройки будут применяться при инициализации при старте питания. Параметры позволят существенно

изменять поведение всего устройства без нужды пересборки всех сорцов. Просто прописали через CLI параметры и перезагрузились. И у вас новый функционал. Успех! Поэтому надо где-то указать как минимум тип данных и имя параметров драйвера.

Листинг 13.3: Перечисление параметров NVRAM

```
ifndef SX1262_PARAMS_H
#define SX1262_PARAMS_H

#include "param_drv.h"
#include "param_types.h"

ifdef HAS_GFSK
include "sx1262_gfsk_params.h"
else
define PARAMS_SX1262_GFSK
endif

ifdef HAS_LORA
include "sx1262_lora_params.h"
else
define PARAMS_SX1262_LORA
endif

define PARAMS_SX1262 \
    \ \
    PARAMS_SX1262_LORA \
    \ \
    PARAMS_SX1262_GFSK \
    \ \
    {SX1262, PAR_ID_FREQ, 4, TYPE_UINT32, "Freq"}, /*Hz*/ \
        \ \
    {SX1262, PAR_ID_WIRELESS_INTERFACE, 1, TYPE_UINT8, "Interface"}, /*LoRa or \
        \ \
        GFSK*/ \
    {SX1262, PAR_ID_RX_GAIN, 1, TYPE_UINT8, "RxGain"}, \
    {SX1262, PAR_ID_RETX, 1, TYPE_UINT8, "ReTx"}, \
    {SX1262, PAR_ID_IQ_SETUP, 1, TYPE_UINT8, "IQSetUp"}, \
        \ \
    {SX1262, PAR_ID_OUT_POWER, 1, TYPE_INT8, "OutPower"}, /*LoRa output power*/ \
        \ \
endif /* SX1262_PARAMS_H */
```

13.7 config_xxx.c config_xxx.h файл с конфигурацией по умолчанию

В репозитории должны быть файлы config_xxx.c/ и config_xxx.h с конфигурацией по умолчанию.

Очевидно же, что после старта электропитания надо как-то проинициализировать драйвер. Особенно при первом запуске, когда NVRAM ещё пустая. Для этого создаем отдельные файлы для конфигов по умолчанию. Это способствует методологии

"Код отдельно, конфиги отдельно".

Драйвер должен легко масштабироваться. Поэтому для каждого программного компонента конфиг надо хранить именно в массиве.

Листинг 13.4: Конфиг для программного компонента

```
include "data_utils.h"
include "ds3231_config.h"
include "ds3231_types.h"

const Ds3231Config_t Ds3231Config[] = {
    {.num=1, .i2c_num=1, .valid=true, .hour_mode=HOUR_MODE_24H}, ,
    {.num=2, .i2c_num=2, .valid=true, .hour_mode=HOUR_MODE_24H}, ,
};

Ds3231Handle_t Ds3231Item[] = {
    {.num=1, .valid=true, .init=false}, ,
    {.num=2, .valid=true, .init=false}, ,
};

uint32_t ds3231_get_cnt(void) {
    uint8_t cnt=0;
    cnt = ARRAY_SIZE(Ds3231Config);
    return cnt;
}
```

Конфиг и сам драйвер надо писать так, чтобы драйвер поддерживал сразу несколько экземпляров сущностей драйвера. Так драйвер можно будет масштабировать новыми узлами.

13.8 xxx_commands.c xxx_commands.h файл с командами CLI

У каждого взрослого компонента должна быть ручка для управления. В мире компьютеров исторически еще со времен UNIX (в 197x) такой "ручкой" является интерфейс командной строки (CLI) поверх UART. Поэтому создаем отдельные файлы для интерпретатора команд для каждого конкретного драйвера. Буквально 3-4 команды: инициализация, диагностика, get ,set регистра. Так можно будет изменить логику работы драйвера в Run-Time. Вычитать сырые значения регистров, прописать конкретный регистр. Показать диагностику, серийный номер, ревизию , пулить пакеты в I2C, SPI, UART, MDIO и т. п.

Листинг 13.5: Команды CLI

```
ifndef DWM1000_COMMANDS_H
define DWM1000_COMMANDS_H

#include "std_includes.h"

ifdef __cplusplus
extern "C" {
endif
```

```

bool dwm1000_read_register_command(int32_t argc, char* argv[]) ;
bool dwm1000_read_offset_command(int32_t argc, char* argv[]) ;
bool dwm1000_init_command(int32_t argc, char* argv[]) ;
bool dwm1000_diag_command(int32_t argc, char* argv[]) ;
bool dwm1000_reset_command(int32_t argc, char* argv[]) ;

#define DWM1000_COMMANDS
    CLI_CMD( "dro" , dwm1000_read_register_command , "Dwm1000ReadReg") ,
    CLI_CMD( "dwd" , dwm1000_diag_command , "Dwm1000Diag" ) ,
    CLI_CMD( "dwi" , dwm1000_init_command , "Dwm1000Init" ) ,
    CLI_CMD( "dwr" , dwm1000_reset_command , "Dwm1000Reset" ) ,
\

ifdef __cplusplus
}
endif

endif /* DWM1000_COMMANDS_H */

```

13.9 xxx_diag.c/xxx_diag.h файлы с диагностикой

У каждого драйвера есть куча всяческих констант. Значения подобраны вендором обычно рандомно. Эти константы надо интерпретировать в строки для человека-понимания. Поэтому создается файл с Hash функциями. Суть проста: даешь бинарное значение константы и тут же получаешь её значение в виде текстовой строчки. Эти Hash функции как раз вызывает CLI(шка) и компонент логирования при логе инициализации board(ы).

Листинг 13.6: Интерпретатор констант

```

const char* DacLevel2Str(uint8_t code){
    const char *name="?";
    switch(code){
        case DACLEV_CTRL_INTERNALY: name="internally"; break;
        case DACLEV_CTRL_LOW:       name="low"; break;
        case DACLEV_CTRL_MEDIUM:   name="medium"; break;
        case DACLEV_CTRL_HIGH:     name="high"; break;
    }
    return name;
}

```

13.10 test_xxx.c/test_xxx.h Файлы с модульными тестами диагностикой*

Драйвер должен быть покрыт модульными тестами (скрепы). Это позволит делать безопасное перестроение кода с целью его упрощения. Тесты нужны для отладки большого куска кода, который трудно проходить пошаговым отладчиком. Тесты позволят быстрее делать интеграцию. Помогут понять, что сломалось в

случае ошибок. Т.е. тесты позволяют сэкономить время на отладке. Тесты будут поощрять вас писать более качественный и структурированный код.

Если в вашем коде нет модульных тестов, то не ждите к себе хорошего отношения. Так как код без тестов - это Филькина грамота.

13.11 Make файл xxx.mk для правил сборки драйвера из Make

Сборка из Make это самый мощный способ управлять модульностью и масштабируемостью любого кода. С make можно производить выборочную сборку драйвера в зависимости от располагаемых ресурсов на печатной плате. Код станет универсальным и переносимым. При сборке из Makefile(ов) надо для каждого логического компонента или драйвера вручную определять make файл. Make - это целый отдельный язык программирования со своими операторами и функциями. Спека GNU Make всего навсего это 224 страницы.

Листинг 13.7: mk файл сборки программного компонента

```
ifeq ($(SI4703_MK_INC),Y)
SI4703_MK_INC=Y

SI4703_DIR = $(WORKSPACE_LOC) Drivers/si4703
#@echo $(error SI4703_DIR=$(SI4703_DIR))

INCLUDE_PATHS += -I$(SI4703_DIR)

OPT += -DHAS_SI4703
OPT += -DHAS_MULTIMEDIA
RDS=Y

FM_TUNER=Y
OPT += -DHAS_FM_TUNER

SOURCES_C += $(SI4703_DIR)/si4703_drv.c
SOURCES_C += $(SI4703_DIR)/si4703_config.c

ifeq ($(RDS),Y)
    OPT += -DHAS_RDS
    SOURCES_C += $(SI4703_DIR)/si4703_rds_drv.c
endif

ifeq ($(DIAG),Y)
    ifeq ($(SI4703_DIAG),Y)
        SOURCES_C += $(SI4703_DIR)/si4703_diag.c
    endif
endif

ifeq ($(CLI),Y)
    ifeq ($(SI4703_COMMANDS),Y)
        OPT += -DHAS_SI4703_COMMANDS
        SOURCES_C += $(SI4703_DIR)/si4703_commands.c
    endif
endif
```

```
endif  
endif
```

Вот так должен примерно выглядеть код драйвера в папке с проектом:

Name	Type	Size
fda801_types.h	H File	12 KB
fda801_logBook.txt	TXT File	7 KB
fda801_drv.h	H File	3 KB
fda801_drv.c	C File	38 KB
fda801_diag.h	H File	4 KB
fda801_diag.c	C File	28 KB
fda801_const.h	H File	7 KB
fda801_config.h	H File	1 KB
fda801_config.c	C File	6 KB
fda801_commands.h	H File	4 KB
fda801_commands.c	C File	21 KB
fda801.mk	MK File	1 KB

Рис. 13.2: Содержимое папки с файлами

13.12 xxx_dep.h файл с проверками зависимостей

Добавить xxx_dep.h файл с проверками зависимостей на фазе препроцессора. Это позволит отловить на стадии компиляции ошибки отсутствия драйверов, которые нужны для этого драйвера.

Листинг 13.8: Проверка зависимостей препроцессором

```
ifndef DWM3000_DEPENDENCIES_H  
define DWM3000_DEPENDENCIES_H
```

```

ifndef HAS_DWM3000
error "+HAS_DWM3000"
endif

ifndef HAS_SPI
error "+HAS_SPI"
endif

ifndef HAS_GPIO
error "+HAS_GPIO"
endif

ifndef HAS MCU
error "+HAS MCU"
endif

ifndef HAS LIMITER
error "+HAS LIMITER"
endif

endif /* DWM3000_DEPENDENCIES_H */

```

13.13 xxx_preconfig.mk

Дело в том, что перед запуском сборки хорошо бы проинициализировать переменные окружения, которые нужны для данной конкретной сборки. Часть этих переменных можно прописать в корневом config.mk. Однако можно случайно упустить какие-то конкретные зависимости. По этой причине каждый драйвер должен содержать файл xxx_preconfig.mk для явного определения зависимостей.

Листинг 13.9: Преконфиг

```

define KEEPASS_COMPONENT_VERSION "1.2"

$(info AT24CXX_PRECONFIG_MK_INC=$(AT24CXX_PRECONFIG_MK_INC) )

ifeq ($(AT24CXX_PRECONFIG_MK_INC),Y)
AT24CXX_PRECONFIG_MK_INC=Y

TIME=Y
AT24CXX=Y
I2C=Y
GPIO=Y
endif

\end{figure}

```

```

\section{xxx.gvi}                                Graphviz
}                                              Graphviz
xxx.gvi

.

\begin{lstlisting}[label=some-code, caption=Graphviz] ]

```

```

subgraph cluster_Keepass {
    style=filled ;
    color=khaki1 ;
    label = "Keepass" ;

    Base64->KEEPASS
    Salsa20->KEEPASS
    LIFO->XML
    GZip->KEEPASS
    AES256->KEEPASS
    XML->KEEPASS
    SHA256->KEEPASS
    COMPRESSION->KEEPASS
}

```

Подробнее про генерацию зависимостей можно почитать тут

13.14 Функционал драйвера

Со структурой драйвера приблизительно определились. Хорошо. Теперь буквально несколько слов о функционале этого обобщенного драйвера.

1. Должна быть инициализация чипа, функция `bool xxx_init(void)`. Причем повторная инициализация драйвера или любого другого программного компонента не должна приводить к зависанию прошивки или иным ошибкам. Первоначальная проверка `link(a)`, запись строчки в логе загрузки, прописывание либо конфигов по умолчанию, либо конфигов из on-chip Nor FlashFs, определение уровня логирования для данного компонента.
2. В суперцикле должна быть функция `xxx_proc()` для опроса (`poll(инга)`) регистров чипа, его переменных и событий. Эта функция будет синхронизировать удаленные регистры чипа и их отражение в RAM памяти микроконтроллера. Эта функция `proc`, в сущности, и будет делать всю основную работу по функционалу и бизнес логике драйвера. Функция `xxx_proc()` может работать как в суперцикле, так и в отдельном RTOS потоке.
3. У каждого драйвера должны быть счетчики разнородных событий: количество отправок, приёмов, счетчики ошибок, прерываний. Это нужно для процедуры `health monitor`. Чтобы драйвер сам себя периодически проверял на предмет накопления ошибок и в случае обнаружения мог отобразить в лог (UART или SD карта) красный текст или перезагрузить плату.

4. Если ваш чип с I2C, то вам очень повезло, так как в интерфейсе I2C есть бит подтверждения адреса и можно разом просканировать всю I2C шину. Драйвер I2C должен обязательно поддерживать процедуру сканирования шины и печатать таблицу доступных адресов. Вот как тут.

```

9.849-->
10.003-->
10.102-->h i2c scan
argc 2
AvailableCommands:Key1:i2c
Key2:scan

+-----+
| Num | Acronym | CommandName | Description |
+-----+
| 1   | i2cs    | i2c_scan    | I2cScan      |
+-----+
15.569-->i2cs
8.000:I [I2C] InterfaceNum 1
+--| x0 | x1 | x2 | x3 | x4 | x5 | x6 | x7 | x8 | x9 | xA | xB | xC | xD | xE | xF |
+--| -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  |
| 0x | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  |
| 1x | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  |
| 2x | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  |
| 3x | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  |
| 4x | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  |
| 5x | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  |
| 6x | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  |
| 7x | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  |
| 8x | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  |
| 9x | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  |
| ax | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  |
| bx | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  |
| cx | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  |
| dx | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  |
| ex | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  |
| fx | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  |
+--+
8.000:I [I2C] 1 ScanOk
23.910-->

```

Рис. 13.3: Сканирование всех 7-битных адресов шины I2C

5. У каждого компонента должна быть версия. Должна быть поддержка чтения версии компонента в run-time

Листинг 13.10: Версия компонента

```
#define KEEPASS_COMPONENT_VERSION "1.2"
```

При каждом изменении в коде драйвера версию надо увеличивать.

6. У каждого драйвера должна быть функция вычитывания всех сырых регистров разом. Это называется memory blob. Так как поведение чипа целиком и полностью определяется значениями его внутренних регистров. Вычитывание memory blob-а позволит визуально сравнить конфигурацию с тем, что прописано в спецификации на ASIC и понять в каком режиме чип работает прямо сейчас.
7. Если ваш чип является трансивером в какой-то физический интерфейс, будь-то проводной (10BASE5, CAN, LIN, 1-Wire, RS485, MIL-STD-1553, ARINC) или беспроводной (LoRa, UWB, GFSK), то чип должен периодически посыпать Hello пакеты в эфир. Их еще называют Blink пакет или HeartBeat сообщение. Это позволит другим устройствам в сети понять, кто вообще живет на шине, а кто вовсе завис.

18.217-->sim	
19.658:I [Si4703] key1:[] key2:[]	
19.662:I [Si4703] RegCnt:16	
+	-----+-----+-----+-----+-----+-----+-----+-----+
No Addr BinAddr Val BinVal name	
1 0x00 0000_0000 0x1242 0001_0010_0100_0010 DeviceID	
2 0x01 0000_0001 0x1253 0001_0010_0101_0011 ChipID	
3 0x02 0000_0010 0xcb01 1100_1011_0000_0001 PowerCFG	
4 0x03 0000_0011 0x0000 0000_0000_0000_0000 Channel	
5 0x04 0000_0100 0xd804 1101_1000_0000_0100 SysConfig1	
6 0x05 0000_0101 0x0111 0000_0001_0001_0001 SYSCONFIG2	
7 0x06 0000_0110 0xb07f 1011_0000_0111_1111 SYSCONFIG3	
8 0x07 0000_0111 0xbc04 1011_1100_0000_0100 TEST1	
9 0x08 0000_1000 0x0002 0000_0000_0000_0010 TEST2	
10 0x09 0000_1001 0x0000 0000_0000_0000_0000 BOOTCONFIG	
11 0x0a 0000_1010 0x401e 0100_0000_0001_1110 StatusRssi	
12 0x0b 0000_1011 0x00b2 0000_0000_1011_0010 ReadChan	
13 0x0c 0000_1100 0x00ff 0000_0000_1111_1111 RDSCA	
14 0x0d 0000_1101 0x0000 0000_0000_0000_0000 RDSCB	
15 0x0e 0000_1110 0x0000 0000_0000_0000_0000 RDSC	
16 0x20 0010_0000 0x0000 0000_0000_0000_0000 RDSD	
+	-----+-----+-----+-----+-----+-----+-----+-----+

Рис. 13.4: Значения регистров

8. Должен быть механизм непрерывной проверки SPI,I2C,MDIO link(a). Это позволит сразу определить проблему с проводами, если произойдет потеря link(a). Обычно в нормальных чипах есть регистр ChipID (например DW1000). Прочитали регистр ID, проверили с тем, что должно быть в спеке(datasheet), значение совпало - значит есть link. Успех! С точки зрения надежности не стоит вообще закладывать в проект чипы без chip-ID именно по той причине, что их иначе невозможно протестировать. Вот чип LIS3DH хороший ASIC. В нём есть chip-ID.
9. Должен быть предусмотрен механизм записи и чтения отдельных регистров из командной строки поверх UART. Это поможет воспроизводить и находить ошибки далеко в run-time-e.
10. (Advanced) Должна быть диагностика чипа. В идеале даже встроенный интерпретатор регистров каждого битика, который хоть что-то значит в карте регистров микросхемы. Либо, если нет достаточно On-Chip NorFlash-a, должна быть отдельная DeskTop утилита для полного и педантичного синтаксического разбора memory blob-a, вычитанного из UART. Так как визуально анализировать переменные, глядя на поток нулей и единиц, если вы не выучили в школе шестнадцатиричную таблицу умножения, весьма трудно и можно легко ошибиться. Поэтому интерпретатор регистров понадобится при сопровождении и отладке гаджета.
11. * Если есть функция, которая что-то устанавливает, то должна быть и функция, которая это что-то прочитывает. Проще говоря, у каждого setter-a должен быть getter, подобно тому как в математике у каждой функции есть обратная

функция.

12. Если у чипа есть внутренние состояния (Idle, Rx, Tx и проч), то об изменении состояния надо сигнализировать в UART Log. Это нужно для отладки драйвера чипа.
13. В коде драйвера должны быть ссылки на страницы и главы из спецификации, которые поясняют почему код написан именно так. Это детализация констант, структура пакета, адреса регистров, детализация битовых полей и прочее.
14. Каждый драйвер MCAL: UART, SPI, SysTick так или иначе взаимодействует с прерываниями. В связи с этим, драйвер должен проверить, что обработчик прерывания по данному номеру в самом деле прописан в таблице векторов прерываний в Flash памяти. Иначе просто нет смысла пользоваться этим драйвером, так как при первом же срабатывании прошивка свалится в Hard Fault. Лучше предупредить такую ситуацию и вызвать `dynamic_assert()` при инициализации. Например в ARM Cortex-M обработчики прерываний хранятся в начале FLASH памяти сразу после верхушки стека по смещении 4. Чаще всего это адрес `0x0800_0004`. Надо убедиться, что там в самом деле размещены адреса на нужные функции обработчики прерываний.

Листинг 13.11: Проверка наличия обработчика IRQ на ARM Cortex-M4

```
uint32_t core_isr_handler_addr_get(int16_t irq_n) {
    int32_t offset = 60 + irq_n*4;
    uint32_t vector_table_addr = SCB->VTOR+4 + offset ;
    uint32_t isr_handler_addr = 0;
    isr_handler_addr = read_addr_32bit(vector_table_addr);
    LOG_INFO(SYS,
              "IRQ:%d, Offset:%d, Addr:0x%08p=0x%08x",
              irq_n, offset, vector_table_addr,
              isr_handler_addr);
    return isr_handler_addr;
}

bool core_is_valid_isr_handler(int16_t irq_n) {
    bool res = false;
    uint32_t isr_handler_addr = core_isr_handler_addr_get(irq_n) ;
    res = is_flash_addr(isr_handler_addr);
    return res;
}
```

Суммируя вышесказанное, получается вот такой список необходимых файлов

13.15 ИТОГИ

То, что тут перечислено - это базис любого драйвера. Своего рода ортодоксально - каноническая форма.

Остальной код зависит уже от конкретного ASIC чипа, будь это чип управления двигателем, беспроводной трансивер, RTC или простой датчик давления.

№	приоритет	Файл	расширение	Назначение файла
1	10	xxx_drv.c	c	файлы с самим функционалом
2	10	xxx_drv.h	h	файлы с самим функционалом
3	10	xxx_preconfig.mk	mk	установка нужных переменных окружения для данного компонента
4	9	xxx_isr.c	c	код обработчика прерываний
5	9	xxx_isr.h	h	код обработчика прерываний
6	9	xxx_const.h	h	константы данного программного компонента
7	9	test_xxx.c	c	модульные тесты
8	9	test_xxx.h	h	модульные тесты
9	9	xxx.cmake	cmake	скрипт сборки для CMake
10	9	xxx.mk	mk	скрипт сборки для make
11	9	xxx_types.h	h	типы данных для данного программного компонента
12	8	xxx_diag.c	c	диагностика. Преобразователи типов данных в строку
13	8	xxx_diag.h	h	диагностика
14	8	xxx_commands.c	c	команды CLI для управления и отладки программного компонента
15	8	xxx_commands.h	h	команды CLI
16	8	config_xxx.c	c	конфигурация для каждого экземпляра
17	8	config_xxx.h	h	конфигурация
18	6	xxx.gvi	gvi	перечень зафиксимостей для graphviz
19	6	xxx_dep.h	h	проверка зависимостей на фазе препроцессора
20	3	xxx_param.h	h	параметры
21	1	Kconfig	--	конфигурация для KConfig

Рис. 13.5: список необходимых файлов программного компонента

Как видите, чтобы написать адекватный драйвер чипа надо учитывать достаточно много нюансов и проделать некоторую инфраструктурную работу.

Не стесняйтесь разбивать драйвер на множество файлов. В этом нет ничего плохого. Это потом сильно поможет при custom-лизации, переносе и упаковке драйвера в разные проекты с разными ресурсами памяти.

13.16 Гиперссылки

1. Архитектура Хорошо Поддерживаемого драйвера для I2C/SPI/MDIO Чипа
2. Эффективное использование GNU Make

Глава 14

DevOps для производства Firmware

"А Вы думали программирование это просто?"

14.1 Пролог

Часто слышал мнение, что в embedded программировании в принципе не может быть никакого DevOps(а). Якобы вот есть GUI(ня) в IAR и там надо много мышкой водить. "Ты же не станешь ставить шаговые двигатели для сдвигания мышки" и т. п.

В этом тексте я намерен пофантазировать каким мог бы быть абстрактный процесс разработки firmware с точки зрения DevOps. И перечислить атрибуты такого процесса.

По правде, говоря все мы немного dev ops(ы). Сейчас объясню почему... Мы же не вручную код на assembler(е) пишем. Вовсе нет! Мы из абстрактного языка Си компилятором генерируем артефакты (*.hex, *.bin файл). А это значит, что мы DevOps(еры) все.

Однако этого конечно же не достаточно. Нужно рассмотреть ещё вот эти атрибуты.

14.2 Репозиторий с кодом (репа/общак)

Это может быть. Git, SVN, Mercurial, ClearCase, Perforce. Репозиторий нужен не только для хранения, распространения кодовой базы среди разработчиков, но и в случае поломки сборок репозиторий позволит откатиться в истории на прежние версии кода, когда все было относительно хорошо. Можно восстановить случайно удаленные файлы. Репозиторий позволяет распределить работу среди нескольких вкладчиков. Контроль версий запоминает все шаги. С репозиторием намного спокойнее жить и работать.

Плохие примеры когда код в компании передают через DropBox, USB-Flash(ку) или архивом в электронном письме. В этом случае нет никаких гарантий, что у

каждого разработчика та же версия, что у остальных.

14.3 Код-генерация

В firmware проектах часто много повторяющегося кода. Это синтаксический разбор содержимого payload бинарных пакетов (например CAN, RS485 и пр.). Синтаксический разбор регистров каких-н умных навороченных периферийных SPI, I2C, MDIO микросхем (драйверы сенсорных экранов, внешние ADC, трансиверы, драйверы исполнительных механизмов, интеллектуальное управление питанием и пр.). Для автоматизации написания кода синтаксического разбора можно прибегнуть к созданию простых консольных утилит код-генераторов. На основе текстовых файлов структуры пакета при помощи код генераторов можно мгновенно сформировать циклопические *.c *.h файлы. Код генератор повышит гибкость проекта, уменьшит вероятность ошибки, ускорит внесение изменений.

14.4 Проект должен собираться скриптами

Многие IDE (IAR, Code Composer Studio) могут запускать сборки из командной строки Windows, просто запустив *.bat скрипта. Запускать сборки со скриптов полезно еще и по той причине, что окна GUI(ни) в IDE IAR часто зависают.

А в окно log(a) сборки в CCS не помещается весь текст 6 минут работы компилятора и теряются первые сообщения о критических предупреждениях. Если вы собираете проект из скриптов, то у вас останется log файл компилятора и вы сможете проанализировать все сообщения и исправить осечки.

14.5 Сборок должно быть МНОГО

Чтобы создать хорошую модульную кодовую базу с полной изоляцией компонентов надо собирать проект по частям. Подобно тому как в математике интегрируют по частям. Например если это IoT устройство, то надо подготовить сборку только с GNSS, сборку только с LTE модемом, сборку для проверки качества платы, сборку с загрузчиком, приложение, отладочная сборка с модульными тестами.

Также полезно сделать сборку для какого-н другого микроконтроллера (ESP32, STM, TI, AVR). Это даст гарантию, что код в самом деле переносимый. Если каждая сборка собирается без ошибок, то это как раз и является свидетельством, что код достаточно модульный и переносимый.

Когда на одной кодовой базе много сборок, то компилятор сам подсказывает тебе, где конфликты и как написать переносимый код. Понимаешь?

14.6 Проект собирать Make файлами

Когда Make файлы являются для вас исходниками, то вы можете делать супер модульный код. Буквально одной строчкой в *.mk файле добавлять и исключать компоненты из сборок. Если же вы привыкли пользоваться IDE, то вам для добавления одного компонента придется в одной вкладке IDE добавить пути к заголовочным файлам в другом окне добывать переменные препроцессора, в третьем окне IDE добавлять сами *.c файлики и так для всех 55 сборок. И это все мышковозня. А в Make файлах это будет всего лишь 1 строка.

IDE хороши в основном для прототипирования. Разработчики, которые пользуются только IDE не понимают какой путь проходит код от написания до попадания в Flash. Не догадываются даже о существовании файлов с расширениями *.mk, *.a, *.opt, *.so, *.dot, *.i, *.asm, *.o, *.icf, *.ld, *.cmd, *.elf, *.out, и прочее. Хотя все эти файлы мелькают в workspace директории.

В промышленном подходе к разработке надо писать Make *.mk файлы вручную и подвергать их версионному контролю.

14.7 Статические анализаторы

После компиляции код следует подвергать статическому анализу. Можно воспользоваться бесплатным CppCheck. Это позволит выявить и исправить еще некоторые критические ошибки.

14.8 Автосборки

Есть одна типичная проблема. Можно склонировать код из репозитория и выяснить, что он уже как полгода не собирается и никто об этом не догадывался, так как код собирался только локально на DeskTop(е) разработчика. Забыли подвергнуть версионному контролю какие-то файлики.

Кодовая база в репозитории должна быть валидной каждый день. То есть сборки должны собираться без ошибок и без предупреждений компилятора из кода, что в репозитории. Кто за этим будет следить?

Когда есть репозиторий и скрипты сборки можно автоматизировать запуск сборок при помощи бесплатной программы Jenkins.

Это утилита с Web интерфейсом, которая запускает скрипты. Когда работает Jenkins, то можно не только удостовериться, что код синтаксически правильный, но и всегда каждый день получать свежие артефакты.

Там есть удобная навигация и сортировка по категориям Job(ов). Можно даже открыть доступ Jenkins(y) клиентам продукта и они не будут беспокоить разработчиков вопросами, где же нам взять прошивку. Клиенты сами смогут взять ту прошивку, которая им нужна из Jenkins.

Надо использовать CI/CD, чтобы прошивки собирались изолированно на отдельном сервере из Git репозитория. Это позволяет гарантировать чистую сборку.

А для таких автосборок необходимы скрипты сборки (GNU Make).

Чтобы не было такого, что одна и та же прошивка ведет себя по-разному в зависимости от того на чьём компе она была собрана. Или такого, что код склонированный с репозитория вообще не собирается. Потому, что автор забыл закомитить какие-то файлы.

14.9 Сервер сборки

Настроить Jenkins это весьма кропотливая работа и много мышковозни. Запускать CI на каждом компьютере каждого разработчика это еще и бессмысленная повторная работа. Плюс нагрузка LapTop(a), лишний шум во время работы от кулера. На самом деле достаточно осуществить пуск сервера только 1 раз для всех разработчиков.

Нужен какой-то общий компьютер. Можно задействовать дешевенький Win(довый) NetTop PC (Зомбик). Запустить на нем Jenkins и оставить его работать 24/7. За ночь он соберет все сборки. Когда кому-то понадобился артефакт, то он подключается по Win Remote Desktop Connection или TeamViewer к зомбику и скачивает себе артефакт, посмотрит все ли в порядке с остальными сборками и в случае поломки сделает коммиты, чтобы починить код.

14.10 Модульные тесты (скрепы)

То, что код собирается в реце это еще ни о чем не говорит. Код может собираться без единого предупреждения, а при загрузке на target плата будет бесконечно Reset(тся). Код должен корректно стартовать и исполняться. Как это проверить автоматически? Это можно сделать при помощи модульных тестов. Модульные тесты это просто функции, которые запускают другие функции и проверяют output. Должна быть сборка для тестирования программы изнутри (методом белого ящика). В идеале надо тестировать все нетривиальные функции. В каждом firmware проекте есть hardware зависимый код и hardware независимый код (математика, строки, абстрактные структуры данных). Часть кода, который не зависит от железа можно собирать, запускать и тестировать прямо на x86-64. Памяти на PC много и все тесты поместятся. В Jenkins должен быть отдельный Job для запуска тестов на PC.

Если же в MCU не хватает достаточно NorFlash памяти для сборки проекта с модульными тестами, то можно задействовать отладочную плату с чипом того же семейства но с большим объемом NorFlash. Подключив нужную периферию перемычками получится прототип тестируемого устройства.

14.11 Hardware In The Loop (HIL) стенд

Остаются аппаратно-зависимые тесты. Их надо запускать и выполнять прямо на target(e). Чтобы это автоматизировать надо минимум 3 вещи. Устройство,

загрузчик и CLI. Соединив Target с NetTop компьютером по UART получится HIL стенд. Загрузчик позволит автоматически обновлять прошивку по тому же UART. CLI позволит подключиться к target по UART и запустить тесты, вычитать лог и сохранить отчет. Все это можно упаковать в отдельные Job(ы) на Jenkins.

14.12 Code Coverage (высший пилотаж)

Как понять, что составлено достаточно модульных тестов? Может случится, что какие-то строки протестированы 100 раз а другие ни разу. Ответить на этот вопрос можно только если как-то считать по каким строкам код прошел во время тестов, а какой код является недостижимым. Для этого есть специальные проприetary Tool(ы) например Testwell CTC++.

14.13 Ежедневные планерки

Когда несколько разработчиков делают одну кодовую базу надо как-то координировать действия, переоценивать трудоемкость задач, расставлять приоритеты, сообщать от том, что сделано, какие есть загвоздки и что каждый собирается делать. Для этого надо собираться каждый день в первой половине буквально на 5...7 минут и устно проговаривать. Так можно исключить ситуации, когда одно и то же сделано несколько раз каждым разработчиком.

Еще раз хочу отметить, что все перечисленные 12 DevOps атрибутов всего на всего плоды моей фантазии. И если кто-то увидит в этом какие-то корреляции с реальностью, то это всего лишь совпадение.

14.14 ЭПИЛОГ

При налаженном DevOps можно организовать полностью удаленную работу даже для процесса разработки Firmware. Разработчику достаточно делать коммиты в репозиторий и анализировать Job(ы) Jenkins(а) и изредка подключаться к HIL стенду.

В теории не обязательно даже локально устанавливать Cross ToolChain, ничего кроме своего любимого текстового редактора и браузера. Так 1 человек может контролировать до 20 сборок.

Хороший DevOps позволяет увеличить bus factor до бесконечности. Что является хорошей метрикой.

Надеюсь этот текст поможет кому-нибудь автоматизировать свои проекты.

14.15 Гиперссылки

1. CI CD прошивок для микроконтроллеров в Wires Board (Начало на 25:50)

2. Конвеерум 30: Эволюция рабочего окружения для embedded разработки
3. Атрибуты Хорошой Прошивки

Глава 15

Способы Отладки и Диагностики FirmWare

"Любая разработка начинается с полноценных средств для отладки. Подобно тому как альпенизм начинается с верёвок."

15.1 Пролог

В этой главе перечислены основные способы отлаживать и диагностировать проекты на микроконтроллерах. Для аналогии буду каждому методу отладки метафорично приводить в соответствие аналогию из медицины.

Любая разработка начинается там, где появляются полноценные средства отладки. То есть возможность делать диагностику. Когда нет способов диагностики, то и говорить о разработке не приходится. Это просто закон жизни.

Программировать МК без отладки это как писать ручкой с закрытыми глазами.

15.2 Модульные тесты (скрепы / гипс)

Часто возникает ситуация, когда есть большой кусок кода и нет возможности пройти этот код отладчиком. Например времени мало. В этом случае можно просто покрыть этот код модульными тестами. Модульный тест это по сути функция которая вызывает другую функцию с известными параметрами и спотрин на получившийся результат. Если вывод совпал с ожиданием то тест пройден. Говорят зелёный тест. Также тесты позволяют без опаски совершать перестройку софта. Тесты будут стимулировать вас писать более модульный и качественный код.

15.3 Health Monitor (Мед брат)

Health Monitor (HM) просто поток или отдельная периодическая функция, которая только лишь регулярно проверяет критические параметры прошивки. Сво-

его рода периодические модульные тесты. HM проверяет, например, что UART и CLI не упала. Что есть всяческие Link(и) с I2C, SPI чипами. Что напряжение не просело. Плюс Health Monitor(a) в том, что если что-то внезапно рухнет (например от заряженной частицы из космоса), то прошивка об этом узнает сразу и что-нибудь да предпримет. Health Monitor обладает правами выполнить какой-то мелкий ремонт: переинициализировать отдельный компонент, что-нибудь починить или и вовсе Reset(нуть) гаджет.

15.4 CLI(шка) (Command Line Interface) (или Компьютерная томография MPT)

А вот CLI это как раз самый мощный способ отладки прошивок. Как правило CLI терминал это полнодуплексный текстовый интерфейс поверх UART. CLI как способ человеку общаться с прошивкой через понятый обоим сторонам текстовый интерфейс. При этом нужно всего 3 провода (Rx, Tx и GND) и поддержка в коде прошивки. По факту раз запустив CLI и GDB вам уже больше не понадобится. Далее отладка будет только через CLI. Далее вы ограничены одной только своей фантазией в том какие CLI команды добавить в сборку. Можно по команде вычитывать любую память. Можно запускать функции по их адресу в физической памяти. Можно просматривать значения счетчиков, можно пулить пакеты в интерфейсы I2C, SPI, UART. Можно добавить интерпретаторы аппаратных регистров таймеров. Да что угодно.

Важно, чтобы CLI была не просто простыней унылого текста как в NanoVNA V2. В хорошей CLI должно быть: эхо, энергонезависимая история введенных команд, help, TAB-автонабор, авторизация, аутентификация, поддержка цветов, отрисовка таблиц. Это минимальный джентельменский набор взрослой CLI.

Чтобы ошибки отображались красным текстом, предупреждения желтым текстом. Чтобы печатались таблицы для GPIO пинов и каналов ADC.

Также надо различать такие понятия как логирование и CLIшка. При обычном беспонтовом логировании микроконтроллер тупо и отчаянно шлет текст на улицу (в UART). В случае с CLI пользователь может даже сам асинхронно отправить текст в микроконтроллер со стороны улицы. Должна быть установка и отключение логирования для конкретных программных компонентов (например для SPI или ADC), чтобы не было ниагарского водопада из логов, парализующего работу всей прошивки и всякое взаимодействие с программой. CLI не обязательно делать в UART. Подойдет и TCP/UDP, LoRa, BLE-SPP. Просто правда в том, что UART прост как палка и стартует почти сразу после reset(a). Раньше прочих навороченных интерфейсов как TCP стек, а значит и отладить по UART можно будет практически всё.

Причем логировать можно и в RAM память, а отобразить накопившийся лог в UART только после того как проинициализируется подсистема UART (функция flush). Поэтому писать логи можно даже для инициализации подсистемы тактирования, которая отрабатывает до запуска аппаратных драйверов. Достоинство

```

COM5 - Tera Term VT
File Edit Setup Control Window Help
0.877:M [CLI] board: Olimex-
0.881:M [CLI] MCU: stm32f[REDACTED]
0.884:N [CLI] Date: Aug 7 2022
0.888:N [CLI] Time: 18:29:41
0.891:N [CLI]TimeStamp: Thu Aug 04 01:55:52 2022
0.895:N [CLI] Cstd: 1
0.898:N [CLI] StdHosted: 1
0.901:N [CLI] StdCver: 199901
0.905:N [CLI] branch: GIT_BRANCH_AUTO_REPLACE
0.909:N [CLI] lastCommit: GIT_LAST_COMMIT_HASH_AUTO_REPLACE
FlashCRC32: 0x39666553
1.654:E [CLI] main() Addr 0x0800d709 Error
1.658:N [CLI] Compiler GCC

1.661:N [CLI] GNUC: 10
1.664:N [CLI] GNUC_MINOR: 3
1.668:N [CLI] GNUC_PATCHLEVEL: 1
1.671:N [CLI] version [10.3.1 20210824 (release)]
1.676:N [CLI] NoInline
1.679:N [CLI] TG: [REDACTED]
1.682:N [CLI] Made in Russia
1.685:I [SYS] SCB->UTOR: 0x0800c000
1.689:I [SYS] AddrOfMain: 0x 800d709
1.692:I [SYS] BootStackEnd: 0x20020000
1.696:I [SYS] AppStackEnd: 0x20020000 Offset: 131072 Byte
1.701:I [SYS] AppResetHandler: 0x080339e5
1.706:I [SYS] BootResetHandler: 0x080ea391
1.710:I [SYS] Main Task started, up time: 1710 ms
21.731:I [BOOT] AppLoadedFine!
37.747-->

```

Рис. 15.1: Фрагмент лога загрузки прошивки

```

1.685:I [SYS] SCB->UTOR: 0x0800c000
1.689:I [SYS] AddrOfMain: 0x 800d709
1.692:I [SYS] BootStackEnd: 0x20020000
1.696:I [SYS] AppStackEnd: 0x20020000 Offset: 131072 Byte
1.701:I [SYS] AppResetHandler: 0x080339e5
1.706:I [SYS] BootResetHandler: 0x080ea391
1.710:I [SYS] Main Task started, up time: 1710 ms

1.877-->
2.181-->
2.405-->gl u2
+-----+
| No | pad | level | dir | pull |connect1 |connect2 |      name   |
+-----+
|  0 | PA8 | L   | Out | Air  | U2.4 |       - | USB_HS_UBUSON |
|  1 | PA10| L   | Out | Air  | U2.1 |       - | USB_FS_UBUSON |
|  2 | PB5 | H   | In  | Air  | U2.3 |       - | USB_HS_FAULT  |
|  3 | PB6 | H   | In  | Air  | U2.2 |       - | USB_FS_FAULT  |
+-----+
4.450-->

```

Рис. 15.2: Диагностика GPIO

CLI в том, что CLI не нарушает таймингов в той мере, как это делает пошаговая отладка по JTAG/SWD. Вы отлаживаете прошивку "без наркоза". С CLI(шкой) у вас будет тотальный контроль над софтом и железом. Попробуйте запилить CLI и вы сами увидите, как вам самим понравится такая отладка.

15.5 HeartBeat LED (медленный стетоскоп)

Самое важное. Каждое электронное устройство должно иметь хотя бы один светодиод (LED) для того, чтобы пользователь, инженер или техник мог понять, что встроенное firmware вообще вертится и исполняется, а не зависло. HeartBeat LED это своего рода Smoke Test. Если LED не мигает, то тут дальше и говорить не о чем. Сразу очевидно, что прошивка либо не стартовала, либо и вовсе зависла. В случае немигающего LEDa надо подкатывать тяжелую артиллерию. Также желательно ставить отдельно красный LED для индикации ошибки в софте или железе.

HeartBeat LED обычно ставят мигать с частотой 1Hz и, если это в самом деле так, то можно тут же дать гарантию, что тракт подсистемы тактирования тоже корректно настроен.

В общем, господа, делайте всегда HeartBeat LED.

15.6 Пошаговая отладка GDB через SWD/JTAG (или Хирургическое вмешательство с наркозом или КТ)

В микроконтроллерах есть специальные прерывания, которые позволяют программе останавливаться на заданном адресе (аналогия с анабиозом), который binutils(ами) конвертируется в строку кода в сорцах. Чтобы этим воспользоваться нужно запустить GDB сервер отладки и GDB клиент. Можно остановить программу на конкретной строчке кода.

Однако пошаговая отладка плоха тем, что она нарушает таймингами Real Time программы. Стоит поставить одну единственную точку останова и вы уже отлаживаете совершенно не ту программу, которую писали. Аппаратные таймеры ушли далеко вперёд. Внешний сторожевой таймер начинает сбрасывать микроконтроллер.

GDB - это как рентгеновская компьютерная томография. Не стоит её делать часто, так как это опасно для здоровья.

Однако пошаговая GDB отладка порой недоступна по той простой причине, что во Flash памяти банально нет места, чтобы засунуть *.bin(арь) с ключами, добавляющими отладочные символы (-g3 -ggdb -gdwarf-2 -O0). Поэтому такую прошивку можно будет отладить разве, что по UART-CLI.

15.7 Утилита arm-none-eabi-addr2line.exe (УльтраЗвуковое Исследование УЗИ)

Если вы поймали HardFault и удалось извлечь значение регистров PC и LR, то можно определить на какой строчке упала прошивка выполнив преобразование адреса в номер строки. Вот такой скрипт покажет проблемную строчку в коде.

Листинг 15.1: Правильный if

```
echo off
cls

set ELF_FILE=C:/ applications/nrf5340/build/app/zephyr/zephyr.elf
set ADDR2LINE="arm-none-eabi-addr2line.exe"

%ADDR2LINE% -e %ELF_FILE% 0x000054cd
%ADDR2LINE% -e %ELF_FILE% 0x0000857e
```

15.8 Функции Assert (или ПЦР тест)

Это просто функции, которые выстреливают, когда их аргумент равен нулю.

Есть свой assert на каждую фазу сборки проекта: makeAssert->preprocAssert->staticAssert-> DynamicAssert. StaticAssert отрабатывает на этапе компиляции. Например, что конфиги валидные. DynamicAssert отрабатывает на этапе исполнения. Можно еще расставить как капканы preprocAssert и даже makeAssert, если вы собираете сборку из makefile(ов). Если сработал DynamicAssert, то можно подключить GDB отладчик и поставить точку останова внутри DynamicAssert. Далее просмотреть стек вызовов и определить причину осечки. В общем пользуйтесь функциями категории assert по-полной.

15.9 GPIO и осциллограф (Кардиограмма)

В прошивках есть очень быстрые RealTime процессы, которые надо измерять. Как это сделать? А вот так... Можно выставлять напряжение на свободных GPIO пинах и далее смотреть их на экране осциллографа. Главное чтобы осциллограф был цифровой так как нужны функции захвата перепадов напряжения. Плюс нужно минимум 2 канала для отладки чего либо нетривиального. Также крайне важно, чтобы осциллограф не гудел как пылесос, иначе не будет никакого желания вообще включать такой осцилл.

15.10 GPIO и логический анализатор (Электроэнцефография)

Более предпочтительным вариантом осциллографу является логический анализатор. У логического анализатора больше каналов (8..16 каналов). Можно анализировать сразу целые много проводные шины (MI, I2S, SDIO). Есть очень хороший логический анализатор Saleae. Подключается к PC по USB 3.0. В отличие от осциллографов логический анализатор абсолютно бесшумный, что позволяет сосредоточится на анализе. Анализ графиков делается прямо на мониторе. Софт рассчитывает периоды, частоты скважность на лету. Также у Saleae очень удобные цепляши.

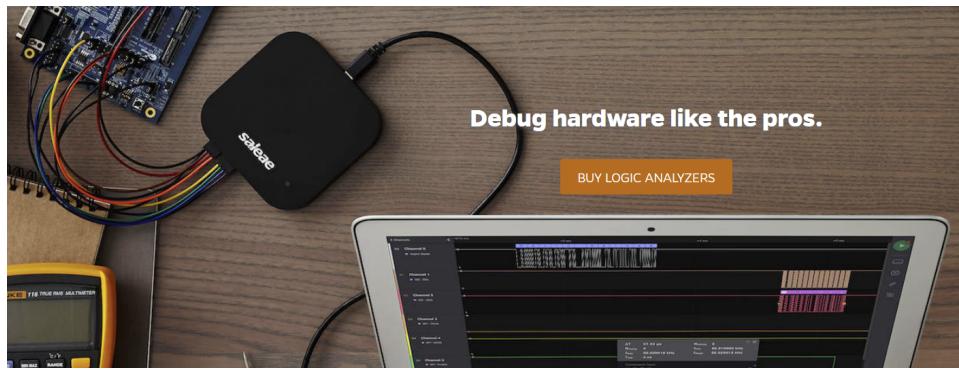


Рис. 15.3: Внешний вид логического анализатора Saleae

15.11 DAC (Цифро-Аналоговый Преобразователь) (или микроскоп)

Когда GPIO не хватает, то можно выставлять аналоговое напряжение на одном DAC пине и тоже смотреть осциллографом с достаточным разрешением. В большинстве адекватных MCU(шек) всегда есть встроенный 10...12бит ЦАП для таких случаев. Очень удобно при отладке планировщиков многопоточных прошивок. По ступеням напряжения на выходе ЦАП можно определить реальный приоритет потоков.

15.12 Логирование на дисплей (Глюкометр)

В принципе, если есть достаточно пинов, то для автономности отладки можно поставить какой-нибудь дисплей и отображать на нем важные метрики софта. Получится устройство тамагочи, которое само все про себя расскажет своему хозяину. Советую смотреть в сторону OLED экранов так как у OLED светятся символы, а не фон. Выглядит футуристично и очень приятно читать.



Рис. 15.4: oled дисплей

15.13 Утилита STM-Studio (Электроэнцефалография)

У ST есть хипстерская Tool(a), которая позволяет по SWD/JTAG следить за конкретными переменными в физической памяти (REG, RAM, ROM) микроконтроллера.

Просто скармливаешь STMStudio *.map файл и подключаешь по SWD Target. Причем эта Tool(a) позволяет строить графики по значениями переменных в памяти. Это как утилита ArtMoney в случае с взломом компьютерных игр на PC, только для микроконтроллера. STMStudio мега удобна при отладке систем автоматического управления, ПИД регуляторов, цифровых фильтров, триггеров Шмитта и прочей DSP обработки. Меняя значения переменных можно даже управлять поведением прошивки! Все бы вендоры микроконтроллеров выпускали бы такие шедевральные утилиты для своих чипов как STMStudio для STM32 микроконтроллеров.

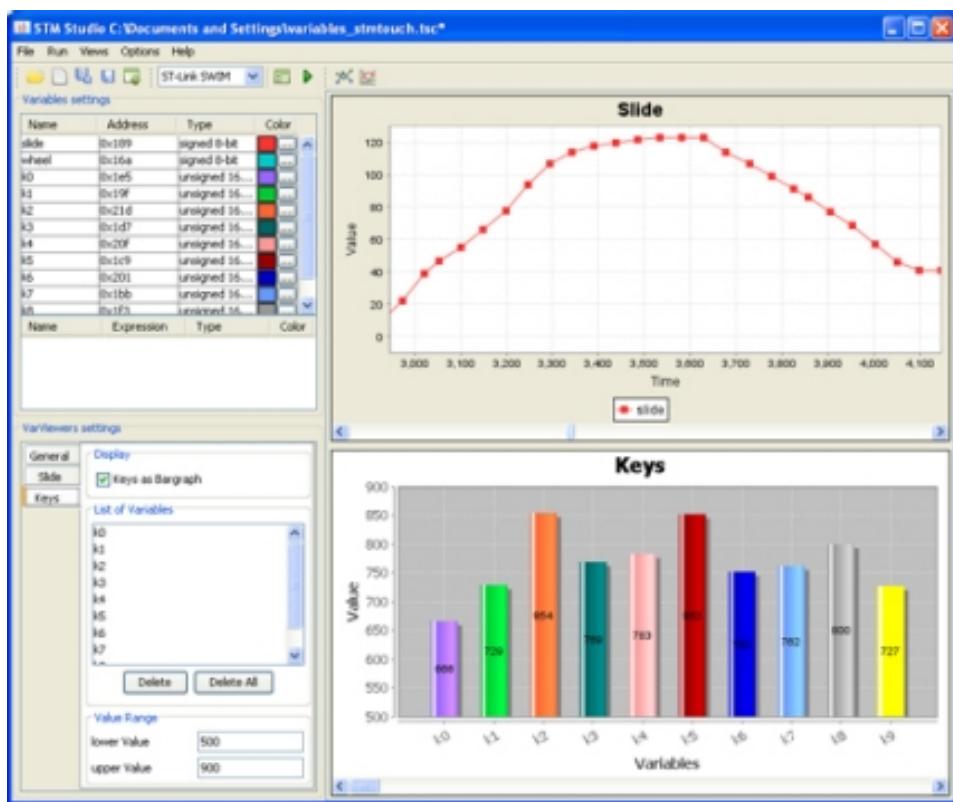


Рис. 15.5: Программа для просмотра памяти MCU

15.14 Логирование в SD карту (Копрограмма)

UART это весьма медленный интерфейс. Максимум 1 MBit/s. Поэтому отладка логированием в UART всё же немного, но нарушает тайминги, а значит и логику работы устройства. Если быстродействие гаджета критично, а логи тоже очень нужны, то логи можно записывать в какие-нибудь число хранилище. Например SD карту или OffChip SPI-NorFlash на несколько мегабайт. Там 25MHz это

норма. Однако придется собрать файловую систему (например FatFs) на SD карте, чтобы потом можно было писать в настоящие файлы и читать их на LapTop. Затем в период Idle выгрузить логи в тот же самый UART или TCP/UDP. Easy.

15.15 Эмуляция прошивки как процесса на PC (клонирование)

Если у вас в прошивке есть CLI, то вы можете собрать прошивку прямо на nettop x86. Прошивку можно запустить на исполнение как консольное приложение в командой строке cmd. Затем исполняя CLI команды в stdin и наблюдая за выводом в stdout вы можете отладить огромные куски кода, которым всё равно на архитектуру.

В консольном приложении можно протестировать и отладить реализацию протоколов, математику, аллокатор памяти, криптографию, fifo, lifo, циклический буфер, алгоритмы расчета контрольных сумм CRC, шрифты графических экранчиков, триггер Шмитта, распознаватели строчек и многое другое, что абсолютно не зависит от конкретной аппаратной архитектуры.

```
0.228 :I [SYS] branch: main
0.230 :I [SYS] lastCommit: e51349f7
0.232 :I [SYS] GitSha: 0x3466badf
0.234 :I [SYS] Compiler GCC

0.236 :I [SYS] TG: [REDACTED]
0.238 :I [SYS] Made in Russia
0.240 :I [SYS] AddrOfMain: 0x 44e06c
0.242 :I [SYS] LittleEndian
0.244 :I [SuperLoop] Start
0.246 :I [SuperLoop] Init
0.250 :I [SuperLoop] Started, UpTime: 248 ms

-->
-->
-->
-->
-->
-->?
2.554 :E [CLI] UnknownCommand [?]
-->h
argc 0
AvailableCommands:
+-----+
| Num | Acronym | CommandName | Description |
+-----+
| 1 | cdd | calc_day_duration | CalcDayDuration
| 2 | clo | calc_longitude | CalcLongitude
| 3 | clat | calc_latitude | CalcLatitude
```

Рис. 15.6: Симуляция прошивки консольным приложением на PC

15.16 DMM, цифровой мультиметр (градусник)

DMM-ом можно проверить самые базовые параметры электронной платы: напряжение электропитания на разных рельсах, частоту PWM на LED(ax), произвести пины, измерить постоянные ток.

15.17 Контроль качества пайки (Load-detect)

Идея очень проста. Надо выбрать конкретный pin, пробежаться по всем трём подтяжкам напряжения, в каждой подтяжке прочитать и запомнить логическое состояние пина на GPIO. Затем найти в подсказке строчку которая и скажет, что подключено к pinu со стороны улицы.

Units: -->	voltage	voltage	voltage	text	text
#	pull air	pull down	pull up	Warning	load solution
0	0	0	0		short GND
1	0	0	3,3		open load
2	0	3,3	0	1	Error
3	0	3,3	3,3	1	short VBAT
4	3,3	0	0	1	short GND
5	3,3	0	3,3	1	open load
6	3,3	3,3	0	1	Error
7	3,3	3,3	3,3		short VBAT

Рис. 15.7: LUT для Load-detect

15.18 Вывод системной частоты наружу (стетоскоп)

В микроконтроллерах STM32 есть один или два пина MCO_1 MCO_2 (Multiplexed Clock-Out) которые позволяют вывести системную частоту поделенную на заданное значение на улицу (за корпус микроконтроллера). В коде это выглядит так

Листинг 15.2: Вывод системной частоты наружу

```
HAL_RCC_MCOConfig(RCC_MCO1, RCC_MCO1SOURCE_HSE, RCC_MCODIV_4);  
HAL_RCC_MCOConfig(RCC_MCO2, RCC_MCO2SOURCE_SYSCLK, RCC_MCODIV_5);
```

```
{.pad.port=PORT_C, .pad.pin=9,  
.name="MCO_2", .mode=GPIO_MODE_AF_PP,  
.gpio_pull=GPIO_PULL_UP,  
.speed=GPIO_SPEED_FREQ_VERY_HIGH,  
.alternate=GPIO_AF0_MCO},  
  
{.pad.port=PORT_A, .pad.pin=8, .name="MCO_1",  
.stm_mode=GPIO_MODE_AF_PP,  
.gpio_pull=GPIO_PULL_UP,  
.speed=GPIO_SPEED_FREQ_VERY_HIGH,  
.alternate=GPIO_AF0_MCO},
```

Потом берем осциллограф, измеряем реальную частоту, сравниваем с настройками в коде, делаем вывод.

15.19 Итог

Как видите, во встраиваемом софте есть великое множество способов отлаживать код и железо. Конечно, тут не все зависит только от программиста. Некоторые опции отладки требуют и от разработчика hardWare некоторых предварительных телодвижений (добавить LED(ы), UART, TestPad(ы), Flash, SWD разъем). Что вы возьмете на вооружение решать вам. Мне же для отладки в 99 процентов случаев хватает LED, CLI, и uTests.

Общая идея такова, что более высокоуровневые программные компоненты можно отлаживать только менее высокоуровневыми программными компонентами.

Если вам известны еще оригинальные способы отлаживать Embedded Software или Firmware, то напишите мне про это.

15.20 Гиперссылки

1. Cross-Detect для Проверки Качества Пайки в Электронных Цепях
2. Почему Нам Нужен UART-Shell?
3. Модульное Тестирование в Embedded
4. Настройка Пошаговой Отладки JLink+Eclipse
5. Отладка интерфейса I2S
6. Пошаговая GDB отладка ARM процессора из консоли в Win10
7. Атрибуты Хорошей Прошивки
8. Load-Detect для Проверки Качества Пайки

Глава 16

Почему Нам Нужен UART-CLI? (Добавьте в Прошивку Гласность)

"Есть только один способ это проверить..."

16.1 Пролог

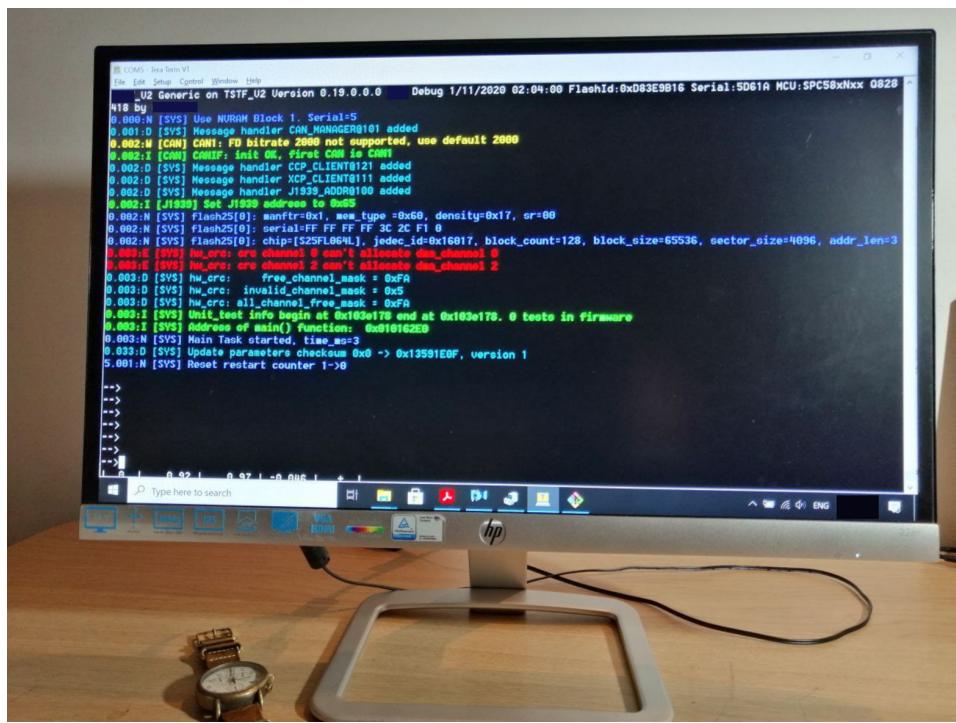


Рис. 16.1: Внешний вид CLI

В самолетостроении есть такое понятие как оснастка. Это формы для литья, режущие инструменты, контрольно-измерительные приспособления, стапели и прочее.

Как и в любой другой разработке в разработке программного обеспечения для микроконтроллеров в качестве оснастки выступает интерфейс командной строки поверх UART. UART-Shell.

Да, есть такая классическая технология отладки Firmware как интерфейс командной строки поверх UART (Real Time Terminal). Как её только не называют: TUI, CLI, Shell, RTT, консоль, последовательный терминал, "портал" и прочее. Это уже намёк на то, что штука это полезная и используют её по-разному.

Почему UART? Ответ прост. UART самый дешевый и простой проводной последовательный интерфейс. Для него доступны как переходники UART-USB (CP2102) так и готовый стабильный софт (TeraTerm, Putty, Hercules и прочее).

В России бытует мнение якобы:

Да мне UART-CLI не нужна так как у меня устройство удаленное и к нему нет доступа кроме беспроводных интерфейсов

Это высказывание можно перефразировать так:

Да нам JTAG/SWD не нужен так как у нас устройство удаленное и к нему нет доступа кроме беспроводных интерфейсов

Иллюзия таких рассуждений в следующем:

1. Во первых. У вас после какого-то коммита сломался ваш беспроводной интерфейс или CAN. И что вы будете делать? Предаваться конвульсиям, судорогам и параличу? Чтобы понять, что сломалось, вам поможет 2 минуты с и пара команд диагностики из UART-CLI или 1 день прочесывания кода пошаговым GDB отладчиком по JTAG/SWD. Лично я выбираю первый вариант.

No	pad	level	mode	dir	pull	MuxS	MuxG	connect2	name
0	PA8	L	ALT1	?	Air	1	1		MCLK
1	PA3	L	Out	Out	Air	0	0		Debug
2	PA4	L	Analog	In	Air	0	0		DAC1_OUT
3	PA5	L	Analog	In	Air	0	0		DAC2_OUT
4	PD13	L	Out	Out	Air	0	0		LED_Red
5	PD14	L	Out	Out	Air	0	0		LED_Yellow
6	PD15	H	Out	Out	Air	0	0		LED_Green
7	PF12	L	ALT1	?	Air	2	2		DIGIT_EXT1
8	PF13	L	ALT1	?	Air	2	2		DIGIT_EXT2
9	PF14	L	ALT1	?	Air	2	2		PWM_ADC1
10	PF15	L	ALT1	?	Air	2	2		PWM_ADC2
11	PE5	H	ALT1	?	Air	2	2		ENS
12	PF0	H	ALT1	?	Up	4	4		I2C2_SDA
13	PF1	H	ALT1	?	Up	4	4		I2C2_SCL
14	PA0	L	In	In	Down	0	0		Button
15	PA9	H	ALT1	?	Air	7	7		USART1_TX
16	PA10	H	ALT1	?	Up	7	7		USART1_RX
17	PC6	H	ALT1	?	Air	8	8		USART6_TX
18	PC7	H	ALT1	?	Up	8	8		USART6_RX

Рис. 16.2: CLI показывает состояние GPIO пинов

2. Во вторых UART-CLI нужна в основном для отладки в Debug(жной) сборке. В Release(ных) артефактах Shell можно и выпилить. Причем если Вы собираете из самостоятельно написанных GNU Makefile(ов), то выпиливание из проекта CLI это заменить один символ в *.mk файле с (Y на N). Проторжество makefile(ов) есть вообще отдельный текст.

3. В третьих CLI можно пустить по любому интерфейсу и протоколу: CAN, LoRa, UDP, UWB. Просто в этом случае придется писать консольную утилиту-переходник для LapTop(a). Хипстерская Tool(a) типа с stdin/stdout текстовый CLI в конкретный бинарный протокол + драйвер интерфейса. А если жалко трафика, то придется еще и пропускать CLI трафик через программный компонент компрессии. Заметьте, никакой GUI(ни) пока не нужно.
4. UART CLI нужна как раз для отладки и верификации механизмов удаленного доступа (например стека LTE/Ethernet/TCP/IP/LoRa). Сами по себе беспроводные стеки это тонна кода, который тоже надо как-то отлаживать. Я могу сказать, что даже для запуска того же LoRa надо подтянуть кучу зависимостей: GPIO, SPI, DMA, FlashFS, UART, TIMERS, SX1262. В BlueTooth зависимостей на два порядка больше. И для стабильного Link(a) эти зависимости все по отдельности должны работать безупречно.

Общая канва такова, что более сложный программный компонент можно отладить только менее сложным компонентом. Ethernet отлаживают, в частности, при помощи UART-Shell. UART отлаживают связкой GPIO+JTAG. GPIO отлаживают мультиметром или логическим анализатором.

Тут можно провести аналогию. У каждого микропроцессора есть система команд. Это список Assembler инструкций, которые может исполнять этот микропроцессор. Аналогично и у электронной платы должна быть тоже своя система команд. То есть список действий, которые может выполнять эта конкретная плата. Вот такое простое рассуждение приводит к заключению, что у прошивки должна быть UART-CLI.

Нужен какой-то портал для доступа к прошивке.

16.2 Почему люди используют CLI?

Далее я перечислил основные причины и возможности, которые дает Вам UART-CLI:

1. UART-CLI как датчик зависания прошивки

UART-CLI нужна по той причине, что CLI служит индикатором зависания прошивки. Суть очень проста. Если прошивка зависла, то CLI не печатает в TeraTerm/Putty эхо введенного символа. Не отвечает на команду. А если поток исполнения живой, то эхо появляется.

UART-CLI особенно выручает на устройствах без LEDов. Что очень часто.

Допустим у вас плата без HeartBeat LED(a). Ну пожалели разработчики денег, ну бывает. Вы накатили прошивку и ничего не происходит. Вообще ничего не поменялось. А вот UART CLI позволит произвести такой простой Smoke тест как “А не зависла ли прошивка?”. Для этого достаточно просто открыть UART консоль и нажать Enter. Если появился курсор ($->$), то тест пройден. Прошивка вертится. Успех.

```

1.113 :I [SYS] AppStackEnd: 0x20020000 Offset: 131072 Byte
1.117 :I [SYS] AppResetHandler: 0x080406bd
1.120 :I [SYS] BootResetHandler: 0x080f24cd
1.124 :I [SYS] LittleEndian
1.126 :I [SYS] Main Task started, up time: 1126 ms

2.998-->
4.118-->
4.978-->

```

Рис. 16.3: курсор CLI показывает

2. Нехватка on-chip NOR-Flash памяти для GDB символов.

Далеко не всегда прошивку удается отлаживать по SWD или JTAG банально из-за нехватки on-chip NOR-Flash памяти для сборки *.bin(аря) с отладочными символами (опции компилятора -g3 -ggdb -gdwarf-2). Поэтому в таких случаях выручает только отладка при помощи UART-CLI. CLI хороша тем, что можно удалить ненужные команды и функционал и, тем самым, уменьшить размер бинаря.

В пределе в CLI можно оставить всего две команды: читать и писать физическую память. И этого более чем достаточно для отладки. Да, господа, вот так...

3. CLI для автоматизации процессов

С UART CLI можно автоматизировать работу с Target(ом). Автоматически прогнать тесты. Автоматически прописать серийный номер. Автоматически прописать ключи шифрования.

4. Пошаговая отладка долго загружается

Потом, пошаговая SWD отладка имеет свойство очень долго загружаться. Буквально делаешь в Eclipse команду Terminate and Relaunch и IDE думает, думает, думает и только перед 2 минутами происходит долгожданный reboot. Это как? Уж лучше и быстрее через UART-CLI отлаживаться.

5. CLI чтобы запросить версию софта и железа

Через UART-CLI вы всегда сможете узнать версию прошивки

```

COM39 - Tera Term VT
File Edit Setup Control Window Help
1.113 >w1
25.477-247 U.[SYS] Config:EML_L1S3DM_ProgType:Generic,Debug.
25.477-247 U.[SYS] Config:CPU:Cortex-M3,CoreClock:16MHz,Freq:43771
25.477-247 U.[SYS] MCID:[at3244729m]
25.486-258 I.[FLASH] Sector:5 0x00020000
25.486-258 I.[FLASH] Sector:5 0x00020000
25.496-252 I.[SVD] GCNC:GMIC-10_GMC_NINCH-3_GMC_PATCHLEVEL-1_STRICT_ANGL.version:(10.3.1_20210824_Release),NameLine,NegPrefix,SizeTypeSize:4,
25.496-252 I.[SVD] Date:Mon 28 2024 Time:18:28:47,TimeStamp:[Fri Oct 25 15:11:06 2024],mainC,Addr:0x00020000,Balld,Cstd:1,StdHosted:1,StdOver:199901,Cit,Lc
25.516-254 I.[SVD] IC:Enabled,ModeInResets
1.113 >

```

Рис. 16.4: CLI показывает версию прошивки

6. UART-CLI как резервная отладка

Если у вас перестал работать Ethernet, то UART-CLI поможет понять в чем загвоздка. Это обыкновенное резервирование для отладки.

7. Нехватка пинов на MCU

Интерфейсы JTAG/SWD как и пошаговую GDB отладку не всегда удается использовать, так как пины для JTAG/SWD уже используются схемотехниками под всякие реле и кнопки. При этом саму прошивку записывают по UART через заводской загрузчик от производителя MCU (пины BOOT[2]). Вот и получается, что и для отладки прошивки ничего не остаётся кроме как задействовать UART-CLI.

8. UART-CLI дешевле JTAG

Оборудование для отладки через UART CLI дешевле, чем оборудование для отладки по JTAG в сотни раз, так как не нужен дорогой программатор. Цена JTAG программаторов, кажется, вообще ничем не ограничена. Видел от 7kRUB до 5kEUR.

9. Нет проблем со статическим электричеством

UART-CLI менее подвержен статическому электричеству в сравнении с SWD и JTAG. У SWD программаторов часто нет Link(a) с Target(ом). UART же работает всегда.

10. У UART-CLI длиннее провода

UART-CLI harness можно протянуть на большую длину чем SWD/JTAG. SWD обычно не работает уже при длине шлейфа около 40 см. Для UART 1 м до HIL стенда это вообще ни о чём.

11. Отладка через UART CLI удобнее, чем отладка по JTAG так как нужно всего 4 провода (Rx Tx GND VDD) вместо 20 проводов JTAG.

12. CLI можно использовать как имитатор устройства. Человек может отправлять данные в CLI в конкретную API(шку), а микроконтроллер будет реально думать, что это какой-то протокол или вообще устройство. То есть варианты комбинаций способов отладки с CLI ограничены только вашей фантазией.

13. Через CLI можно загрузить в устройство энергонезависимые конфиги. Например параметры модуляции беспроводных трансиверов.

14. CLI проста в использовании. Исполнять команды в CLI сможет даже необученный персонал просто следуя скачанной инструкции из pdf(ки) и вам не надо будет посылать программиста в командировку за Урал, чтобы тот настроил радар или перепрошил RFID СКУД.

15. Из консоли TeraTerm можно скопипастить любой кусок текста. При работе с GUI-подобным конфигуратором часто скопипастить текст нельзя так как текст иногда отрисовывается прямо на канве как картинка.

16. Через CLI можно инициировать запуск модульных тестов и увидеть отчет исполнения тестов. Можно запустить только те тесты, которые имеют в своем имени конкретную подстроку (например "i2c").

Или запустить только один конкретный тест. Можно отлаживать программу по частям подобно тому как в школьной математике функции интегрируют по частям.

```

COM3 - Tera Term VT
File Edit Setup Control Window Help
2:3-->
2:3-->
2:3-->
2:3-->tsr gpio*
cmd_unit_test_run() argc 1
126.899 :I [TEST] key1 [gpio+] 1 time
unit_tests_run() key gpio+
***** Run test gpio_const .86/104
126.910 :I [TEST] unit_test_run_key() key gpio
***** Run test gpio_pull .87/104
126.925 :I [TEST] test_gpio_const()
!OKTEST
***** Run test gpio_write .88/104
126.949 :I [TEST] test_gpio_pull()
!OKTEST
***** Run test gpio_pin_lev .89/104
126.972 :I [TEST] test_gpio_write()
126.978 :W [TEST]
Gpio PD0=DMM1000_CS...
126.986 :I [TEST] test_gpio_pin_lev()
127.094 :I [TEST] Set Pad:PD0 DMM1000_CS to 0 Ok!
127.182 :I [TEST] test_gpio_pin_lev()
127.210 :I [TEST] Set Pad:PD0 DMM1000_CS to 1 Ok!
127.218 :I [TEST] Wire Pad:PD0 DMM1000_CS Ok!
127.226 :W [TEST]
Gpio PC11=DMM1000_WKP...
127.234 :I [TEST] test_gpio_pin_lev()
127.342 :I [TEST] Set Pad:PC11 DMM1000_WKP to 0 Ok!
127.350 :I [TEST] test_gpio_pin_lev()
127.458 :I [TEST] Set Pad:PC11 DMM1000_WKP to 1 Ok!
127.466 :I [TEST] Wire Pad:PC11 DMM1000_WKP Ok!
127.475 :W [TEST]
Gpio PB7=LedBlue...
127.482 :I [TEST] test_gpio_pin_lev()
127.590 :I [TEST] Set Pad:PB7 LedBlue to 0 Ok!
127.598 :I [TEST] test_gpio_pin_lev()
127.706 :I [TEST] Set Pad:PB7 LedBlue to 1 Ok!
127.714 :I [TEST] Wire Pad:PB7 LedBlue Ok!
127.722 :W [TEST]
Gpio PB14=LedRed...
127.729 :I [TEST] test_gpio_pin_lev()
127.837 :I [TEST] Set Pad:PB14 LedRed to 0 Ok!
127.845 :I [TEST] test_gpio_pin_lev()
127.953 :I [TEST] Set Pad:PB14 LedRed to 1 Ok!
127.961 :I [TEST] Wire Pad:PB14 LedRed Ok!
!OKTEST
***** Run test gpio_types .89/104
128.068 :I [TEST] test_gpio_types()
!OKTEST
128.077 :W [DECA] Init
128.187 :I [TEST] Test duration 1197 ms =1.197 s= 0.01995 min
128.117 :I [TEST] All 4 tests passed!
2:8-->

```

Рис. 16.5: Через CLI можно прогнать модульные тесты

17. Через CLI можно верифицировать функционал. Заставить испустить синус определенной частоты на DAC или распечатать в UART содержимое файла в FatFS.
18. CLI стимулирует писать более модульный код так как для каждого компонента надо где-то хранить список его внутренних команд.
19. CLI побудит вас сделать общий на все компоненты API для доступа к периферии, компонентам и драйверам. Это позволит вам в будущем быстро мигрировать с одного микроконтроллера на другой просто определив API(шные) функции для очередного чипа. А это первый шаг к методологии AUTOSAR или Zephyr Project.
20. Через CLI можно обновить прошивку. Можно передавать куски прошивки в

формате base64 или просто hex в виде ASCII (получается base16). Или по протоколу zModem поверх CLI.

21. CLI это вообще универсальный протокол. Можно и прошивку кидать, и конфиги прописывать, и различную диагностику выгребать. CLI понимает и человек и компьютер. Для CLI не нужен вспомогательный софт. Всё работает из коробки.
22. UART CLI можно через переходник [USB-TypeC-USB-A(Nest)]-[USB-A(Fork)-UART (чип CP2102)] подключить к Android смартфону и управлять электронной платой с сенсорного экрана телефона по приложению Serial USB Terminal, прямо стоя на улице.

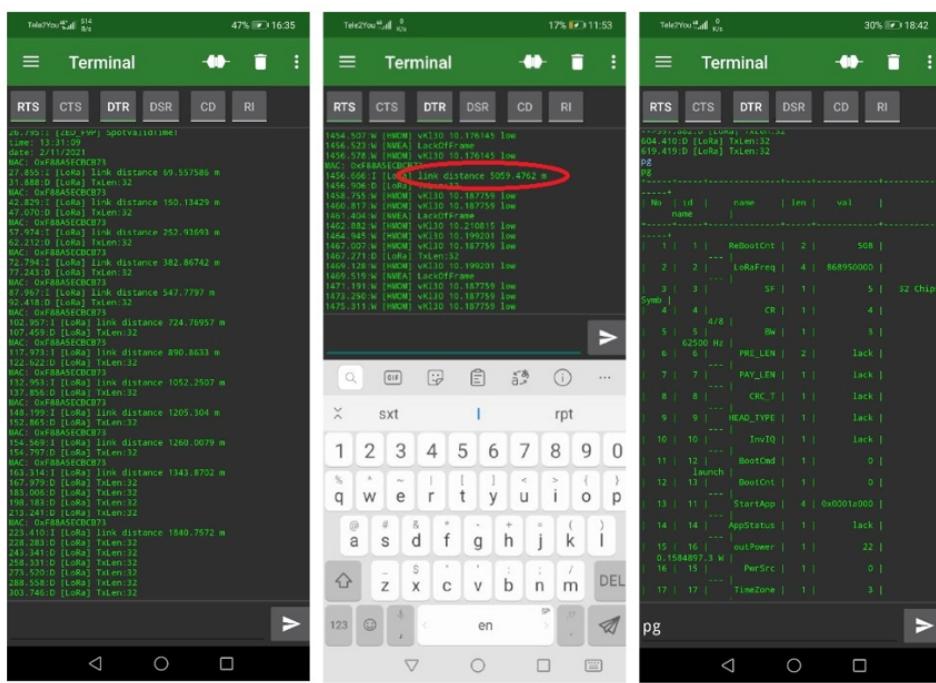


Рис. 16.6: Можно управлять платой с телефона

23. UART-CLI позволит вам воспроизводить известные баги. Вы просто напишите модульный тест, который проделает последовательность операций и запустите его из CLI.
24. CLI повышает вероятность, что написанный код вообще запустят хоть раз перед отгрузкой до 95 процентов ...100 процентов.
25. Еще очень важный момент. Если в прошивке визуально работает UART-CLI, то это автоматически означает, что правильно сконфигурировалось тактирование процессорного ядра и периферии, значит, что включен контроллер прерываний, что работает FIFO. Работающий CLI это отличный модульный тест.
26. Эмуляция прошивки как процесса на PC. Если у вас в прошивке есть CLI, то вы можете собрать прошивку прямо на NetTop x86. Прошивку можно запустить на исполнение как Win консольное приложение в командной строке cmd.

```

0.271 :I [SYS] Made in Russia
0.277 :I [SYS] AddrOfMain: 0x 401a74
0.282 :I [SYS] LittleEndian
0.286 :I [SuperLoop] Start
0.291 :I [SuperLoop] Init
0.299 :I [SuperLoop] Started, UpTime: 299 ms

-->
-->
-->
-->h
argc 0
AvailableCommands:
+-----+
| Num | Acronym | CommandName | Description |
+-----+
| 1   | ll    | log_level | SetOrPrintLogLevels
| 2   | ld    | log_diag  | LogDiag
| 3   | lf    | log_flush | LogFlush
| 4   | lc    | log_color | LogColor
| 5   | ltc   | log_try_color | LogTryColor
| 6   | tcl   | task_clear | TaskInit
| 7   | tcr   | task_ctrl  | TaskControl
| 8   | tdi   | task_diag  | TaskDiag
| 9   | tsa   | test_all   | Print all unit tests
| 10  | tsr   | test_run   | Run test
| 11  | e     | echo        | SetEcho
| 12  | h     | help        | PrintListOfShellCommands
| 13  | vi    | version     | PrintVersionInformation
| 14  | si    | sysinfo    | PrintInformationAboutThreads&OS
+-----+
-->
-->

```

Рис. 16.7: Эмуляция прошивки как процесса на РС

Затем, исполняя shell команды в stdin и наблюдая за цветным выводом в stdout вы можете отладить огромные куски кода, которым всё равно на архитектуру. Протестировать реализацию протоколов, математику, аллокаторы памяти, криптографию, компрессию/декомпрессию, fifo, lifo, циклический буфер, цифровые фильтры FIR, IIR, CIC, hashset(ы), алгоритмы расчета контрольных сумм, шрифты графических экранчиков, триггер Шмитта, распознаватели строчек, генераторы QR кодов и многое-многое другое, что не зависит от архитектуры. Без CLI вы бы не смогли отлаживать платформа-независимый микроконтроллерный код на "большом компьютере".

27. UART-CLI позволит проводить контроль исполнения работы при работе в команде программистов. Вы просто откроете консоль TUI, запустите команды своего смежника и сразу поймете работает его программный компонент или драйвер или не работает.

Shell это самый очевидный способ доказать или опровергнуть наличие или отсутствие конкретного функционала в прошивке.

28. CLI не так грубо нарушает timing(и) время исполнения программ как пошаговая отладка , при отладке через CLI код протекает в естественном режиме и лишь изредка слегка замедляется редким выводом в UART. Это вам не точки останова JTAG, которые останавливают программу на несколько секунд и полностью нарушают логику приложения. Для современных 200MHz микроконтроллеров 1 секунда это как для человека эра. С Shell(ом) получается none disturb отладка.

29. Еще в далеком 1986 в микроконтроллерные устройства в прошивки встраивали интерпретаторы Basic(a). Примером тому служит гаджет Электроника МК-90. Получалось, что устройство можно было допрограммировать уже в Run-Time(е)! Хорошая CLI это по сути тоже интерпретируемый язык программирования. Когда в устройстве есть CLI, то существует больше use case(ов) для этого устройства.

30. Дополнительная ценность

Когда у Вас в прошивке есть CLI, любое ваше устройство помимо базового функционала превращается в супер калькулятор покруче любого Cassio. Только вместо клавиатуры и дисплея два провода UART. Можно прямо в консоли TeraTerm/PuTTY попросить микроконтроллер расшифровать кусок Base64, RLE, AES256, вычислить SHA256, посчитать CRC16, решить задачку на поиск пересечения временных интервалов, определить точное количество дней между 2мя датами, вычислить расстояние между двумя GNSS координатами, найти угол между векторами, решить линейное уравнение, вычислить формулу и прочее и прочее.

Вот, например, прошивка рассчитывает ёмкость керамического конденсатора по его трехбуквенной маркировке. А раньше это была головная боль.

```
7.291-->cc 104
12.538,110 I,[AnalogFilter] Cap:0.0000001=100.000nF
12.543,111 I,[AnalogFilter] Capacity:[104]->100.000nF
12.547-->
45.345-->cc 105
50.810,112 I,[AnalogFilter] Cap:0.000001=1.000uF
50.814,113 I,[AnalogFilter] Capacity:[105]->1.000uF
50.818-->
1:16-->cc 106
85.401,114 I,[AnalogFilter] Cap:0.00001=10.000uF
85.406,115 I,[AnalogFilter] Capacity:[106]->10.000uF
1:25-->
```

Рис. 16.8: Консольный конденсаторный калькулятор

Буквально недавно выручила консоль. Когда есть UART-CLI, вы можете из любого устройства с ADC сделать прибор для проверки напряжения для пальчиковых батареек AAA, AA.

Когда есть CLI прошивка превращается в Swiss Army Knife.

31. Простота работы с UART-CLI и гибкость

Работать с CLI по UART легко и приятно и UART-CLI ничем Вас не ограничивает в плане формата payload(a). Можете посыпать открытый текст в любом удобном для данной ситуации способом (например в ascii), можете отрисовывать ASCII таблицы.

Читать вывод прямо глазами. Не нужно писать никаких вспомогательных host утилит, достаточно существующих TeraTerm/Putty.

Num	Acronym	CommandName	Description
1	avd	auto_volume_diag	AutoVolumeDiag
2	cld	clock_diag	ClockDiag
3	ut	up_time	UpTime
4	swp	sw_pause	SwPause
5	hwp	hw_pause	HwPause
6	gl	gpio_list	GpioList
7	gi	gpio_init	GpioInit
8	gd	gpio_dir	GpioDir
9	ge	gpio_test	GpioTest
10	gt	gpio_toggle	GpioToggle
11	gg	gpio_get	GpioGet
12	gp	gpio_pull	GpioPull
13	gs	gpio_set	GpioSet
14	ll	log_level	SetOrPrintLogLevels
15	ld	log_diag	LogDiag

Рис. 16.9: ASCII таблица в UART CLI

32. CLI для экономии денег и времени

Если у вас в электронное плате нет UART-CLI, а только какой-то проприетарный бинарный протокол, то Вам надо тогда писать ещё утилиту переходник на PC чтобы подключиться к устройству. А потом еще такую же утилиту для Win, Linux, Mac, Android, iOS, MS-DOS, OS/2, IBM POWER8 и БЭСМ-6. Не проще ли просто реализовать текстовый UART-CLI протокол с раскраской логов, ASCII таблицами на уровне Firmware и подключаться по любому терминалу последовательного порта (Putty/TeraTerm), которые и так есть для всех OS? UART-CLI избавит Вас от написания калейдоскопа ненужных по-наме утилит-переходников под все известные платформы. Ну сами подумайте, откуда у СНГ(шной) электронной embedded конторы деньги на зарплату 250kRUR/мес для C/Java программиста для написания всех этих одноразовых PC утилит-переходников?

33. Внешнее управление микроконтроллеров по UART.

Вы можете реализовать в UART-CLI всего 2 команды: чтение физического адреса и запись в физический адрес. Далее вы можете соединить target PCB и большой компьютер переходником USB UART и уже писать прошивку на PC. Говоря метафорично, микроконтроллер будет выступать марионеткой, DeskTop-PC кукловодом, а ниточками - 2 провода от UART. Таким образом вы становитесь никак не ограничены в размере бинаря и при этом можете сопрягать прошивку со всеми ресурсами PC: базы данных, монитор, доступ в интернет и прочее.

34. Пользователи продукта будут Вам благодарны за CLI

Даже если Вы добавите сокращенный Shell в релизную сборку для какого-нибудь прибора, то пользователи будут Вам очень благодарны за такой extra функционал. Они смогут проделывать свои специфические операции с вашим прибором. Подключать его к большому компьютеру и писать для устройства

скрипты. Примером устройств, где CLI оставлен в релизе являются векторный антенный анализатор NanoVNA V2, трансивер Flipper Zero, отладочная плата U-Blox ODIN, Bluetooth модуль BC127 и прочее. Поэтому, можно сказать, что CLI нужна не сколько разработчику сколько продвинутому пользователю продукта.

35. UART-CLI используют другие

UART-CLI давно используют в США, Европе и Китае. Поэтому и нам в России тоже надо уметь работать с UART-CLI. К сожалению Россия всегда догоняет Запад в технологическом плане.

Именно на Западе первыми стали строить реактивные авиа двигатели, радары, ракеты, атомное оружие, вертолёты, атомные подводные лодки, водородные бомбы, цифровые компьютеры, персональные автомобили, космические челноки, карманные калькуляторы, самолёты, фотоаппараты. Перечислять можно очень долго. А смысл в том, что традиции нарушать как-то не принято.

Можно и дальше перечислять достоинства CLI (будь то по UART или UDP).

16.3 Аналогии CLI на бытовом уровне

Метафорично CLI в прошивке это как строительные леса в civil engineering(e).

Когда строят дом, его облачают в строительные леса (scaffolding). Они выглядят кустарно, некрасиво однако отлично помогают строительству. И это же не значит, что эти строительные леса останутся навечно после использования дома. Так же и в программировании. Для строительства кода тоже нужны своеобразные строительные леса в виде другого кода: диагностика, printf отладка, CLI, модульные тесты, скрипты сборки.

Вы когда-н видели чтобы дом строили без строительных лесов? Вот и я не видел.

Вы где-нибудь видели, чтобы строители летали на JetPack(ax) с кисточкой и ведром в руках при покраске фасада многоэтажек? Вот и я не видел.

CLI для программиста микроконтроллеров это как для врача биолога микроскоп. С CLI можно внимательно разглядеть внутренности программы во время исполнения.

Вот скажите мне после этого, почему же тогда большинство российских программистов МК на 15м-20м году опыта в свои 43 года говорят:

Что еще за UART-CLI? О чём это вы? ... А чё, так можно было что ли?!

Ответ:

"Не можно, а нужно!".

В программировании МК тоже есть такое понятие как инфраструктурный код. В программировании MCU(шек) инфраструктурный код это tandem UART Shell + модульные тесты. Без этого ваш проект банально рухнет под собственным

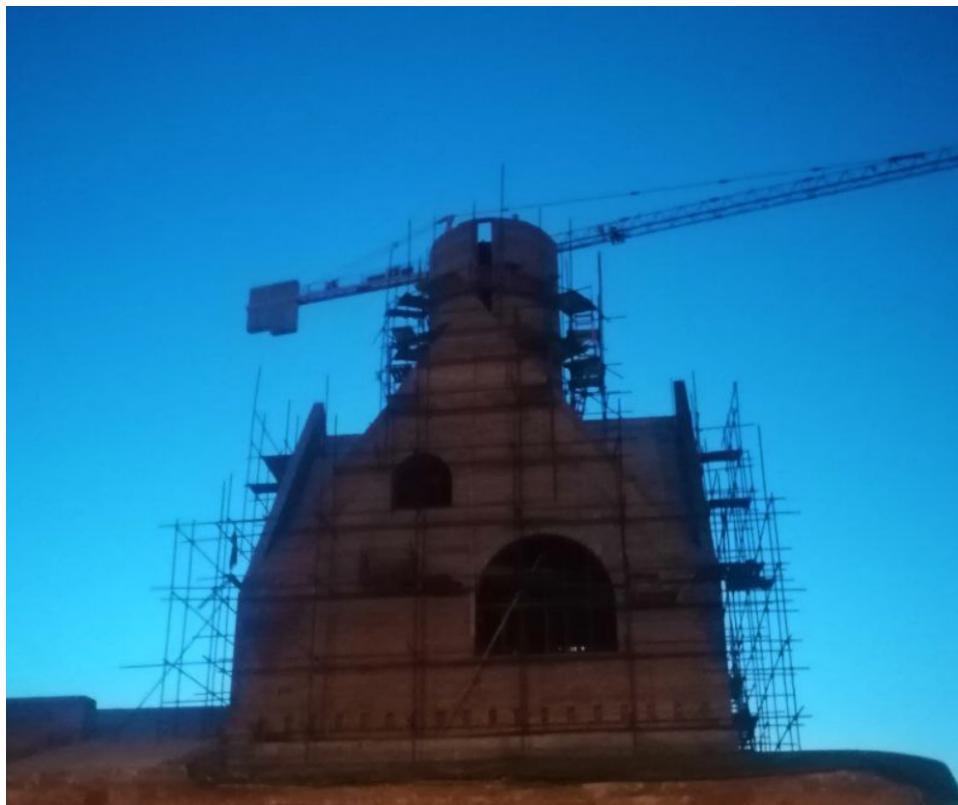


Рис. 16.10: CLI в прошивке это как строительные леса на стройке

весом спустя год и произойдет это в самый неподходящий момент. Или код просто не будет работать, а вы об этом даже не будете догадываться.

Отказываться от UART-CLI это тоже, что если бы военные говорили:

Да нам авиация вообще не нужна. У нас вот есть артиллерия. Поэтому мы обычно мортирами и долбим по противнику.

Разработка электроники это как боевые действия. И тут нужны все виды вооружения.

16.4 Где уже используют CLI

Знаете, я восхищаюсь разработчиками советского калькулятора Электроника МК-85. Они смогли в далёком 1986 году в микроконтроллер с 16kByte ROM встроить интерпретатор целого языка программирования Basic.

В наши же дни в 2024 современные российские программисты с 25ю годами опыта не могут в микроконтроллере с 4Mbyte Flash завести UART-CLI. Это как?

Причем есть же примеры очень успешных продуктов, которые с самого начала заложили UART-Shell в свой прошивки. Это загрузчик U-Boot, анализатор NanoVNA V2, трансивер FlipperZero, приемник GNSS U-Blox ODIN C099-F9P. Bluetooth трансивер BC127. Потом, в кодовой базе Zephyr project есть Shell.

Ещё в проекте Embox есть своя своеобразная UART-CLI.

Для промышленной аппаратуры CLI и вовсе норма жизни (например авиационный RF приемник AW100Rx). Даже в домофонах есть UART-Shell вот (начало

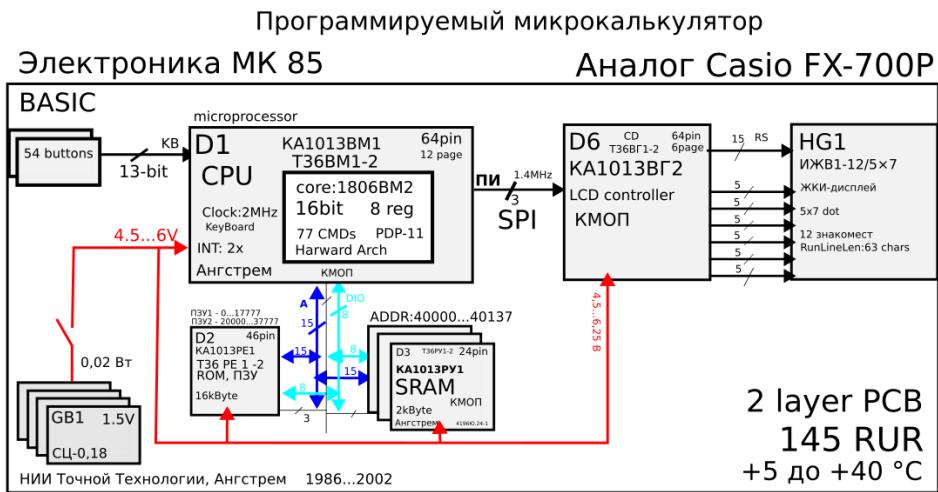


Рис. 16.11: Электроника MK 85

Рис. 16.12: CLI в Embox

на 1:30):

Очевидно, что оглушительный успех этих продуктов в значительной мере определен наличием удобной и развитой CLI.

16.5 Список наиболее часто употребительных команд CLI

Вот список наиболее часто употребительных команд CLI безотносительно к конкретному проекту:

1. Показать список доступных команд Help/TAB
 2. перезагрузиться
 3. запустить модульные тесты

4. установить/считать напряжение на GPIO
5. установить подтяжку напряжения на GPIO
6. показать версию софта и железа
7. показать напряжение на входах ADC
8. запуск аппаратных таймеров
9. установить уровень логирования для конкретного компонента
10. включить/отключить конкретное прерывание
11. прыгнуть в загрузчик
12. перенастроить частоту процессорного ядра
13. показать таблицу состояния потоков и их свойства (стек, приоритет),
14. показать счетчик принятых/отправленных пакетов по всем протоколам
15. Вычтать кусок памяти из REG RAM FLASH
16. показать список файлов в файловой системе FatFs
17. пульнуть произвольные данные в SPI/I2C/I2S/MDIO
18. Найти адрес по значению
19. Проканировать шину I2C,
20. повторить конкретную команду N раз с периодом P
21. отобразить в UART содержимое конкретного файла

У меня в среднем на каждую сборку приходится максимум 120 Shell команд.

```

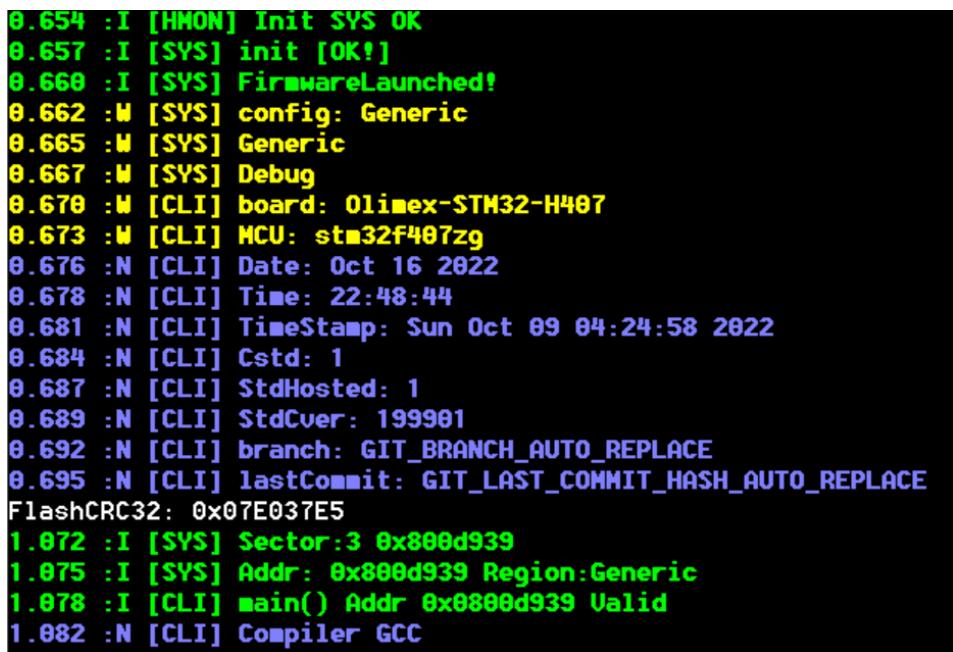
8.355-->
8.495-->
8.635-->E
9.055 :E [SYS] UnknownCommand [E]
9.058-->h
argc 0
AvailableCommands:
+-----+-----+-----+
| Num | Acronym | CommandName | Description |
+-----+-----+-----+
| 1 | jb | jump_boot | jump boot
| 2 | e | echo | SetEcho
| 3 | vi | version | PrintVersionInformation
| 4 | h | help | PrintListOfShellCommands
| 5 | si | sysinfo | PrintInformationAboutThreads&OS
| 6 | cld | clock_diag | Clock diag
| 7 | swp | sw_pause | SW pause
| 8 | tstk | try_stack | Explore stack RAM
| 9 | reboot | soft_reboot | Reboot board
| 10 | cd | core_diag | Cortex M4 diag
| 11 | cla | count_link_all | CountLinkAll
| 12 | cl | count_link | CountLink
| 13 | rm | read_mem | Read memory address
| 14 | fadr | find_addr | Find address by value
| 15 | wm | write_mem | Write memory address
| 16 | rpt | repeat | Repeat any command N time with period
| 17 | lfun | launch_function | Launch any function by address in ROM
| 18 | dd | dma_diag | DmaDiag
| 19 | dd1 | dma_diag_ll | DmaDiagLowLevel

```

Рис. 16.13: Первые несколько команд CLI

16.6 Некоторые незначительные недостатки CLI

1. Нет контрольной суммы CRC в PDU(Protocol data unit). Пользователя же не заставишь в уме высчитывать CRC32 того, что он там написал, и приписывать в конце 32(x) битный hex. Иначе прошивка не ответит. Это было бы ну просто смешно. Однако и тут всё не так плохо. Если вы используете в консоли Putty коды цветов и цвета отображаются на глаз плюс-минус норм, то это уже очень такой хороший знак того, что данные в трафике валидные и не corrupt(ятся)



```
0.654 :I [HMON] Init SYS OK
0.657 :I [SYS] init [OK!]
0.668 :I [SYS] FirmwareLaunched!
0.662 :W [SYS] config: Generic
0.665 :W [SYS] Generic
0.667 :W [SYS] Debug
0.670 :W [CLI] board: Olimex-STM32-H407
0.673 :W [CLI] MCU: stm32f407zg
0.676 :N [CLI] Date: Oct 16 2022
0.678 :N [CLI] Time: 22:48:44
0.681 :N [CLI]TimeStamp: Sun Oct 09 04:24:58 2022
0.684 :N [CLI] Cstd: 1
0.687 :N [CLI] StdHosted: 1
0.689 :N [CLI] StdCver: 199901
0.692 :N [CLI] branch: GIT_BRANCH_AUTO_REPLACE
0.695 :N [CLI] lastCommit: GIT_LAST_COMMIT_HASH_AUTO_REPLACE
FlashCRC32: 0x07E037E5
1.072 :I [SYS] Sector:3 0x800d939
1.075 :I [SYS] Addr: 0x800d939 Region:Generic
1.078 :I [CLI] main() Addr 0x800d939 Valid
1.082 :N [CLI] Compiler GCC
```

Рис. 16.14: Коды цветов как CRC

2. В самом простом виде UART-Shell это открытый текст. Можно перехватить трафик. Можно похитить ценнейшие метрики (логин-пароль, ключи шифрования). Но я подчеркиваю, что UART-CLI это только для отладки софта и железа внутри офиса.
3. Надо защищать устройство от несанкционированного доступа к CLI так как в этом случае можно получить тотальный доступ к устройству, превратить его в BotNet(a). Как защитить терминал в UART? Можно добавить логин и пароль как в Linux. Либо выпускать отдельную desktop tool(y)-терминал для шифровки и расшифровки shell трафика между человеком и электронной платой. Именно так сделали авторы прошивки радиостанции MeshTastic. Там заложены python скрипты-посредники для конфигурации и диагностики LoRa трансивера TTGO T-Beam.
4. Можно нечаянно набрать опасную Shell команду. Например стереть прошивку или замкнуть во внешнем H-мосте VBat на GPIO. Поэтому надо добавить запрос "вы уверены, что хотите выполнять эту опасную команду?" или просто выпилить для каждой конкретной сборки все опасные команды. Или добав-

вить отдельный поток который будет оперативно выявлять и чинить сбои в конфигурациях.

5. В TUI (text user interface) нет контроля непрерывности трафика так как нет как такового заголовка пакета. CLI трафик это атомарные текстовые строки, оканчивающиеся переносом строки. Поэтому каждая Shell команда должна быть самодостаточной и не зависеть от предыдущих shell команд.
6. Есть небольшой overhead. Например DMA от UART-CLI временно блокирует шину данных на время отправки. Для компонента CLI надо немного RAM и Flash.

16.7 Итоги

В общем у Shell плюсов больше чем минусов. Смело добавляйте в свои прошивки UART-Shell. Никто не заставляет вас оставлять shell в релизной сборке, однако в отладочной сборке shell должен быть обязательно. Shell реально того стоит. CLI позволяет вам обеими руками по локоть забраться в исполняемый процесс и найти там любой бит при этом не помешав самому потоку исполнения кода. Эта технология позволит вам говорить с устройством на понятном обоим сторонам языке.

CLI в режиме IDLE вообще ничего не печатает в UART-TX и микроконтроллер не нагружается от слова совсем. UART-CLI только отвечает на редкие запросы, которые поступают в провод UART-RX.

Всегда лучше когда UART-CLI консоль есть, нежели когда её нет.

В России люди не делают UART-CLI лишь по той простой причине, что не умеют программировать на том уровне, на котором становится возможно делать синтаксический разбор разных типов данных из текстовых строчек. Это же надо хотя бы один учебник по формальным грамматикам понять. А для типичного выпускника приборостроительного политеха это просто унижение. То есть наши люди не знают фундаментальных основ программирования! Не понимают, что такое консоль и для чего она нужна. Поэтому и пишут прошивки как слепые котята. Отлаживаются наугад, наудачу, на авось.

По большому счету запуск минималистической UART-CLI вообще ничего не стоит. Надо всего-то правильно проинициализировать GPIO, UART и написать парсер CSV строчек. Вот и всё.

По факту без UART-CLI невозможно работать с микроконтроллерами. Без UART-CLI можно только прикидываться будто ты работаешь, запуская прошивки наугад со скрещенными пальцами играя при этом в бубен на столе.

16.8 Гиперссылки

1. Почему Нам Нужен UART-Shell?
2. Почему важно собирать код из скриптов

3. UART-CLI в проекте Embox
4. CLI в подъездном домофоне
5. MCU-CLI-w25Q80-emulEEPROM
6. embedded-cli
7. Пишем терминальный сервер для микроконтроллера на С
8. Только консоль. Почему текстовый интерфейс настолько эффективен
9. консоль в микроконтроллере с micro readline
10. Command line interpreter на микроконтроллере своими руками
11. Реализация многофункционального терминального интерфейса для МК AVR

Глава 17

Почему Сборка с Помощью Eclipse ARM GCC Плагинов Это Тупиковый Путь.

"Не согласен - критикуй,
критикуешь - предлагай,
предлагаешь - делай,
делаешь - отвечай!"

(Сергей Павлович Королёв)

17.1 Пролог

В период с 199x по 202x на территории России развелось порядка двадцати тысяч программистов-микроконтроллеров, которые никогда в своей жизни не вылезали из всяческих GUI IDE (IAR, KEIL, Code Composer Studio, Atolic True Studio, CodeVision AVR, Segger Embedded Studio и прочие). Как по мне, так это очень печально. Во многом потому, что специалист в Keil не сможет сразу понять как работать в IAR и наоборот. Другие файлы для настроек компоновщика. Другая xml настройки проекта. Миграция на другую IDE тоже вызывает большую трудность, так как это сводится к мышковозне в IDE-GUI.

Это как если пилот летавший на Boeing 737 не сможет понять, как управлять AirBus A320 и наоборот. Там другой HMI. Вместо штурвала джойстик, мониторы не на том месте и прочее. Всё как-то непривычно.

Отдельная тема это бесплатный Eclipse с плагинами. Это вообще как кабина от Ил-62. Аналоговая, топорная, бесхитростная.

В программировании микроконтроллеров часто Eclipse с плагинами используют потому, что в нем есть плагины, которые генерируют make файлы для сборки программ на Си, согласно разметке проекта в XML под названием .cproject и .project.

Эти плагины были сделаны, главным образом, для того, чтобы вовлечь в процесс программирования микроконтроллеров школьников и прочих специалистов



Рис. 17.1: Кабины Boeing и AirBus



Рис. 17.2: Кабины ИЛ-62

без профильного ИТ образования и опыта в области Computer Science. Понимаете?

Далее напишу почему сборка с Eclipse плагинами для IDE не годится для промышленного программирования.

17.2 Теория сборки Си программ в IDE с плагинами

По сути, построение любого Си-кода в IDE сводится к трем простым, но очень частым действиям

1. Добавить мышкой исходник к проекту
2. Добавить мышкой пути к папкам с кодом
3. Добавить мышкой макросы препроцессора к проекту

То, что вы прощёлкиваете мышкой - всё это запоминает IDE.

1. Проблема: Поломаный срoject или project То, что вы прощёлкиваете мышкой всё это запоминает IDE. Всё это сваливается в файл .срoject. Этот файл

.cproject — это авто-генеренный файл, в который руками лучше не соваться. Ибо если Вы случайно там что-то поменяете и сохраните, то в один утренний день у Вас просто не откроется проект! Далее у Вас начнется приступ судороги, конвульсии и паралич. Нормально так, да?... Это проблема номер 1.

2. Проблема: Ручное Прописывание Путей Эта вообще самая большая тема при сборке при IDE с плагинами. При сборке из-под IDE вам всегда придется вручную мышкой в настройках IDE прописывать пути к программным компонентам и к заголовочным файлам

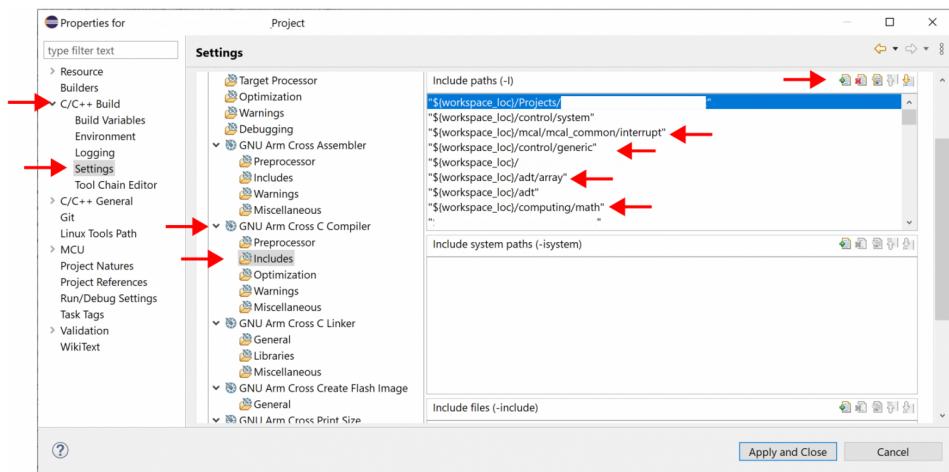


Рис. 17.3: Ручное Прописывание Путей

Но есть один трюк. Он заключается в том, чтобы прописывать пути в переменную окружения INCDIR и скормить её в IDE. Можно написать *.bat скрипт.

Этот скрипт надо отработать до запуска Eclipse. Тут очень важно чтобы черточка была именно 45 гардусов "/". Иначе система сборки не распознает этот путь. Далее надо прописать make переменную **INCDIR**

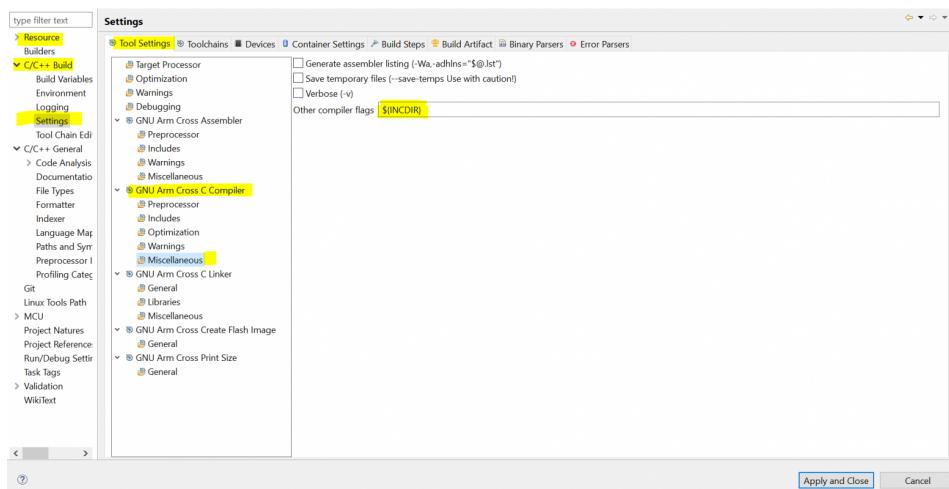


Рис. 17.4: Добавление путей пачкой

Однако и тут всё не так гладко. Максимальная длина переменной окружения в

cmd всего 8192 символа. А сохранить setx(ом) между сессиями можно и вовсе всего только 1024 символа в одной переменной окружения. Понимаете? Вот и получается, что так можно прописать только очень мало путей. Остальное вручную мышкой пока не заноет запястье.

- Проблема: Ручное прописывание макро определений для препроцессора
Очередная беда Eclipse IDE с ARM плагинами - это ручное прописывание макросов препроцессора. Макросы прописываются вот тут Properties -> C/C++ Builds -> Settings -> Tool Settings -> GNU Arm Cross C Compiler -> Preprocessor -> Define Symbol (-D)

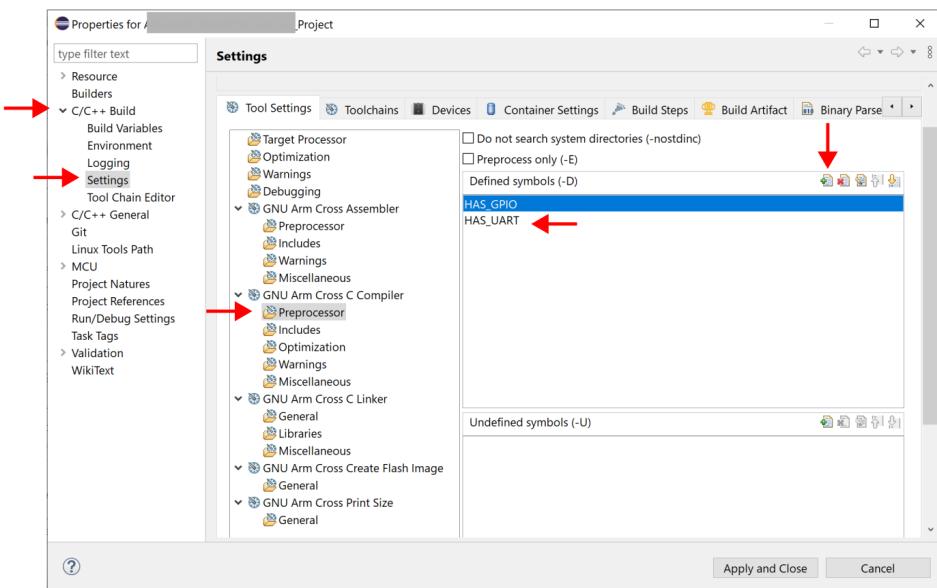


Рис. 17.5: Ручное прописывание макро определений

тут в GUI окне Вы тоже мышкой добавляете и прописываете свои макросы один за другим, пока не заболит рука. Потом нажимаете Apply and Close. Но есть один трюк.

Можно передать файл extr_config.h с макросами

Далее опцией -include прописав строчку -include extra_config.h в настройках компилятора.

Рис. 17.6: прописывание всех макро определений разом

Как видно в логе сборки, макросы передаются успешно извне:

Таким образом Вы сэкономите неделю времени!

- Не собрать из скриптов. Сборку можно инициировать мышкой или горячей клавишей Ctrl+B или тоже мышкой из IDE. Однако очень мало кто знает, что можно также инициировать сборку и из консоли! Да.. Это так... Для начала, если у Вас на PC нет прав администратора, чтобы изменить переменную PATH, то можете каждый раз изменять переменную PATH из консоли Windows cmd вот так
После этого можно смело запускать скрипт сборки build_eclipse.bat

При этом Eclipse откажется собирать проект, если вы предварительно не сделаете refresh вручную опять мышкой из-под IDE. Как сделать авто refresh из-под командной строки - тоже не ясно. Нигде про это не написано. Люди на форумах жалуются, что нереально сделать авто refresh из-под командной строки. Это, к слову, 4тый гвоздь в крышку гроба Eclipse ARM плагинов. Вы уже понимаете всю глубину наших глубин?

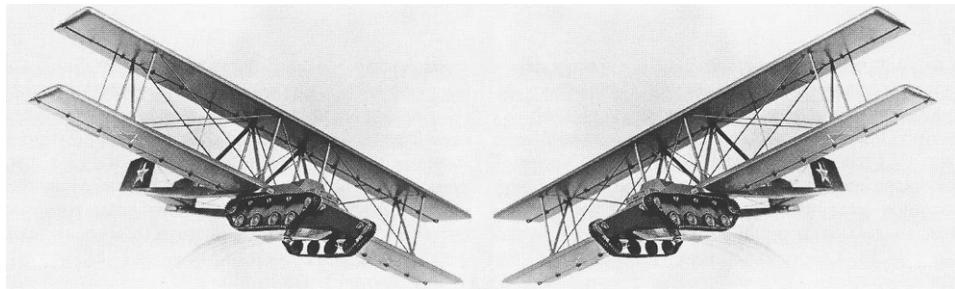


Рис. 17.7: Запуск сборки мышкой из-под GUI-IDE - это так же нелепо как летающие танки.

- Проблема: Ручное прописывание ключей компоновщику. При сборке через плагины, вам надо также вручную мышкой прописывать ключи для компоновщика . Например -lm для подключения математических функций (sin, log, cos).

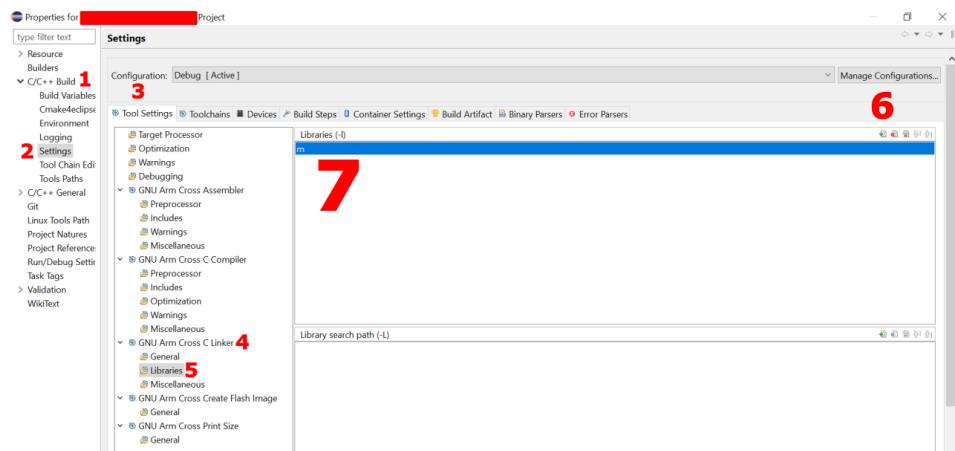


Рис. 17.8: Ручное прописывание ключей компоновщику.

Проблема в том, что это надо проделывать для каждой отдельной сборки, даже если код у сборок общий. То есть опять происходит бессмысленное дублирование конфигов и лишняя работа.

- Проблема с генерацией артефактов. К сожалению, Eclipse с ARM плагинами не может одновременно сгенерировать *.hex и *.bin артефакты. Либо *.hex либо *.bin. GUI прошивальщики требуют hex, а консольные прошивальщики требуют именно *.bin. Поэтому пришлось делать костыль и прописать синтез *.bin отдельно в скрипте перепрошивки утилитой arm-none-eabi-objcopy
- Вот скрипт файла flash_bin.bat
- Вот лог успешной перепрошивки:



Рис. 17.9: Дублирование конфигов это как самолёт с двумя штурвалами.

```
C:\Windows\System32\cmd.exe
project_dir=C:\projects\code_base_workspace\          \Projects\A
artefact_bin=C:\projects\code_base_workspace\          \Projects\
-----
Artery AT-Link Programmer V3.0.08
-----

Connect.....
AT-Link Plus FW: V2.2.10    Serials: F4A81400004001210C159507      (WinUSB)
AT-Link connected
MCU Part Number:  AT32F435ZMT7      Flash size: 4032KB
MCU connected

Disable flash access protection .....
Disable flash access protection finished!
Disable erase and program protection .....
Disable erase and program protection finished!
Download file opening.....
Open file succeed!
Erase MCU sectors of file size.....
Erase sectors finished
File downloading.....
File download finished!
Read memory and Verification.....
Verification finished!
Reset and run.....
Reset and run finished!

Done!
Press any key to continue ...
Press any key to continue . .
C:
```

Рис. 17.10: лог успешной перепрошивки

Либо, как вариант, можно добавить настройку генерировать бинарный *.bin файл как Post-build-steps. Находится это поле по следующему GUI-IDE адресу в GUI: Properties -> C/C++ Build -> Settings -> Build Steps-> Post-build-steps

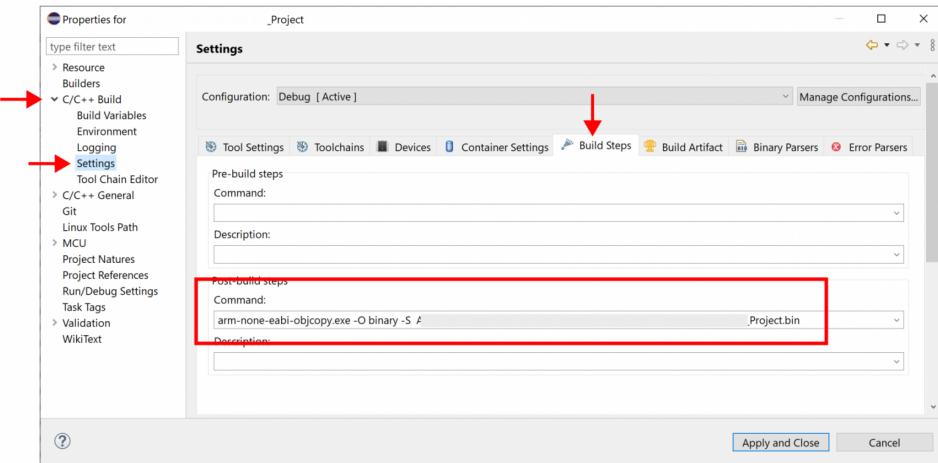


Рис. 17.11: Post-build-steps

Причем если взять один и тот же проект и на разных компах, добавить этот Post-build-steps, то .cproject будет, внимание, на 30 процентов разный! Хотя оба User(a) GUI-IDE мышкой выполнили одни и те же действия. Вот так, господа.... Вы уже "любите" Eclipse ARM плагины?

17.3 Пошаговая отладка

В принципе, пошаговой отладке абсолютно всё равно какая IDE. Пошаговой отладке нужен только GDB сервер, GDB клиент и специально подготовленный *.elf файл. Далее можно хоть из консоли код по строчкам проходить.

Пошаговая GDB отладка ARM процессора из консоли в Win10

17.4 Минусы сборки Eclipse плагинами из-под IDE

1. Конфиги внутри .cproject не отсортированы. Хотя IDE могла бы это и сделать. В результате у двух похожих проектов diff .cproject составляет 99,99 процентов . Это очень трудно проходит инспекцию программ.
2. Высокие накладные расходы на создание ещё одной сборки.

Как следствие создавать отдельные сборки для конкретных плат User(ам) IDE просто лень. В итоге у всех одна сборка - Франкенштейн сразу для всех электронных плат и всех микроконтроллеров!

Издевка судьбы ещё и в том, что одной сборки обычно никогда не бывает! Для каждой электронной платы нужно как минимум пять сборок! Следите за руками... Первичный загрузчик (MBR), Вторичный загрузчик, Generic release, assembly test, Generic debug. Плюс сборки для того чтобы гонять прошивку на отладочных платах от производителя микроконтроллера, пока ваше изделие находится в производстве. Уже получается минимум 6 сборок, господа. Вот и получается, что надо сразу начинать разработку на основе самостоятельно на-



Рис. 17.12: Высокие накладные расходы на создание ещё одной сборки средствами IDE



Рис. 17.13: сборка - Франкенштейн сразу для всех электронных плат и всех микроконтроллеров!

писанных make скриптов, а не на этих ваших пресловутых Arduino(ображных) GUI(невых) IDE. Вот такие вот пирожки с капустой, понимаете...

3. Eclipse плагины не позволяют менять код во время компиляции. Программа может собираться долго. И ваша работа парализована на время сборки. Прошивки надо именно, что варить так как собираются они по 7-15 минут
4. Eclipse плагины собирают всё, что лежит в папке. Как то что нужно, так и то, что не нужно. Это провоцируют путаницу и ошибки.
5. Много мышковозни. Вы конечно можете открыть файл настроек проекта .cproject и дополнять его в текстовом редакторе. Однако эти действия не легальны как бы... При этом если Вы случайно измените там в ненужном месте лишний символ или оставите пробел или TAB, то у вас перестанет собираться сборка! Весь проект на помойку! Вас же при этом от страха застигнут судороги, конвульсии и паралич. Оно вам надо? Может всё-таки на 7м году опыта уже лучше самим писать скрипты сборки? А?...

6. Разработчик GUI-IDE не за что не отвечает. Проблема IDE плагинов в том, что Вы отдаете управление над ситуацией каким-то левым плагинам, написанными непонятно кем. При этом этот кто-то не несет абсолютно никакой ответственности за свою работу. У вас происходит control leak. Это тоже, что у летчика самолет перестанет слушаться штурвала или джойстика.

17.5 Достоинства сборки из-под Eclipse с ARM плагинами

1. Если Вы ещё пока слабо разбираетесь в языке сборки make, то Вы можете проанализировать те makefile(ы), которые для Вас любезнейшим образом синтезировали Eclipse ARM plugins и, на основе этого, написать свои более структурированные и модульные makefile(лы). Вот, пожалуй, хорошее достоинство ARM plugins для Eclipse.
2. Eclipse IDE с плагинами это средство избавления от санкционных и дорогих IDE IAR и Keil.
3. Eclipse IDE с плагинами это бесплатный инструментарий для сборки прошивок. Для бедных.
4. Многие пользуются Eclipse IDE только потому, что им нужна функция автоматического форматирования отступов в исходниках горячей кнопкой Ctrl+Shift+F. А утилитой clang-format.exe или GNU indent они пользоваться не умеют.
5. В Eclipse плагинах есть функция - дымовая завеса кода
6. Плюс разработки в IDE в том, что можно годами, как пиявка, высасывать из организации зарплату за то, что через те же Make скрипты-сборки можно сделать максимум за 3-4 месяца. Именно по этой причине embedded стартапы и не используют IDE, а используют, как раз, именно самописные скрипты сборки (Make, CMake, Ninja, Meson и прочее).

17.6 Аналогии

Если проводить аналогии с атомной энергетикой то, сборка через GUI-IDE - это как реакторы на горизонтальных ТВЭЛах (сборках), а сборка из скриптов это реактор на вертикальных ТВЭЛах.

Горизонтальные реакторы неудобны, так как надо минимум две точки подвеса ТВЭЛа при загрузке топлива. Плюс нужны дополнительные усилия, чтобы проталкивать стержни в активную зону. Потом стержень может расплавиться и застрять. Тогда жди беды.

Напротив, вертикальные ТВЭЛы загружаются под действие силы тяжести и для транспортировки сборки нужна только одна точка подвеса. Расплавленные ТВЭЛы просто высыпаются и онисыпаются в поддон. Easy!

Это то же что, если бы другой посторонний человек диктовал бы вам каждый день как вам надо одеваться. И предлагал мятую одежду, рваные носки и рубашку

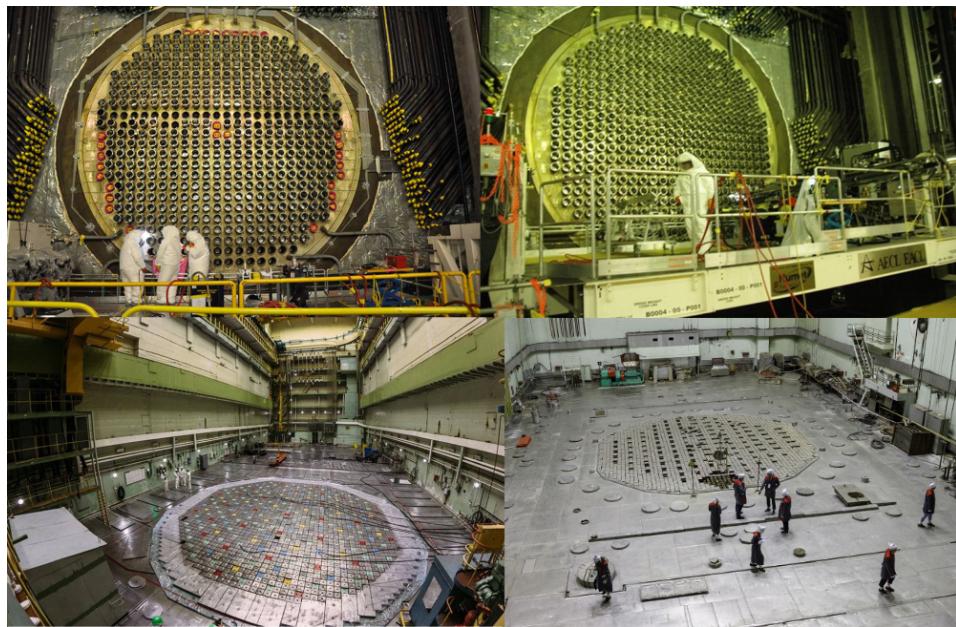


Рис. 17.14: реактор на горизонтальных ТВЭЛах

без некоторых пуговиц на рукавах. В дождь выходить без зонта и прочее. Оно вам надо? Вероятно вы сами хотите выбирать себе повседневную одежду? Так?

17.7 Что делать?

Пишите сами скрипты сборки. Вариантов масса: Make, Ninja, CMake, Meson, Python в конце концов. Если и возникнут ошибки сборки, то в логе хотя бы будет понятный комментарий об ошибке. Так же Вы сможете вообще использовать любой текстовый редактор, а не привязываться только к Eclipse.

Ездите на круглых колесах, а не на квадратных. Пишите скрипты сборки сами!

17.8 Итоги

Суммируя вышесказанное, крайне не рекомендую Вам собирать проекты ARM плагинами Eclipse! Это, к слову, также касается и Code Composer Studio, IDE-IAR и IDE-Keil.

В 2024 году в эпоху DevOps программировать в IDE с плагинами - это тоже, что лаптями щи хлебать.

Работа в GUI-IDE - это для тех, кто не в теме. Для OutSider-ов

Собирать прошивки в GUI IDE это как строить деревянные самолёты вместо металлических.

Лучше пишите сами Make или CMake скрипты сборки.

Это даже проще. Ваша жизнь заиграет новыми красками... Вы сможете масштабироваться хоть до Луны.

Обоснованных плюсов в сборке из-под IDE вообще мало.

Одни только проблемы. Темпы разработки очень-очень низкие. Новые сборки создавать трудно, предстоит полное дублирование конфигов, поддерживать прежние сборки тоже тяжко. Файлы настройки .cproject у всех разные как снежинки, даже если конфиги логически одинаковые.

Также Eclipse с плагинами крайне неприспособлен для сборки из-под командной строки. Это сразу перечеркивает автосборки в Jenkins(е) и вообще DevOps как таковой. И вы автоматически оказываетесь где-то в 198x...199x годах.

Eclipse с плагинами - это только кактус-соло разработка. Только ручные сборки. Ремесленничество, не фабрика. Понимаете?...



Рис. 17.15: Solo разработка в IDE это как вот этот solo полёт

Потом, собирать прошивки в Eclipse с плагинами IDE с плагинами это также неэффективно как ездить на квадратных колёсах.

17.9 Гиперссылки

1. Развёртывание среды разработки для STM32
2. Building Projects with Eclipse from the Command Line
3. Почему важно собирать код из скриптов
4. Настройка сборки прошивок для микроконтроллеров Artery из Makefile
5. Дымовая Завеса в Eclipse IDE
6. How to link Eclipse Project with -lm library for "floor" and "pow" function?

Глава 18

Почему важно собирать код из скриптов?

"То, что мы сейчас услышали, - это именно то, что нужно для решения всей проблемы.

Предлагаю Вам до конца месяца составить записку о проекте завода, в котором будут работать установки.

Чтобы уже в ближайшее время можно было бы получить изотопы, разделённые новым "простым" способом."

(Игорь Васильевич Курчатов)

18.1 Пролог

В период с 199x по 202x на территории РФ развелось порядка двадцати тысяч программистов-микроконтроллеров, которые никогда в своей жизни не вылезали из всяческих GUI IDE (IAR, KEIL, Code Composer Studio, Atolic True Studio, CodeVision AVR, Segger Embedded Studio и прочие). Как по мне, так это очень печально. Во многом потому, что специалист в Keil не сможет сразу понять как работать в IAR и наоборот. Другие файлы для настроек компоновщика. Другая xml настройки проекта. Миграция на другую IDE тоже вызывает большую трудность, так как это сводится к мышковозне в IDE-GUI. Каждая версия IAR не совместима с более новой версией IDE.

Это как если пилот летавший на Boeing 737 не сможет понять, как управлять AirBus A320 и наоборот. Там другой HMI. Нет штурвала, мониторы не на том месте и прочее. Всё как-то непривычно.

Дело в том, что GUI IDE появились в 199x...201x, когда не было расцвета DevOps(а), программист работал один и все действия выполнялись вручную. Мышкой. В то время работа в GUI казалась программистам-микроконтроллеров веселее, ведь в IDE много стразиков.

Но с усложнением кодовой базы, с увеличением сборок, с увеличением команд разработчиков появилась нужда в переиспользовании кода, нужда в автосборках, в авто тестах. Появилась методология



Рис. 18.1: boenig vs airbus

код отдельно, конфиги отдельно

и работа с IDE стала только тормозить процессы. Ведь конфиги хранятся в IDE-шной XML(ке). Приходилось дублировать конфиги платы для каждой сборки, что использовала эту плату. Пришлось дублировать код конфигов и этот процесс сопровождался ошибками, из-за человеческого фактора. При масштабировании работы с IDE кодовая база фактически превращалась в зоопарк в болоте.

Это мнение не оригинальное и уже много раз звучало в сообществе.

-  **iig** 7 дек 2019 в 16:19 ▲

Как только ваш проект станет немного сложнее чем helloworld, ему понадобится система сборки.

◆ +3 Ответить Bookmark ...

-  **igorpt1024** 7 дек 2019 в 16:19 ▲

Минусить — дурное дело, а вот контрагументировать — пожалуйста! Возражу насчёт вести сборку. Дело в том, что пока проект живёт в пределах ноутбука, можно собирать и с помощью IDE. А когда над ним трудятся несколько человек (а ещё и если команда распределённая), то без CI-сервера уже не обойтись. И никакого IDE там нет и быть не может. Дальше. Если сборка идёт скриптом, то обеспечивается воспроизводимость результата сборки. Т.е., если сборщик (make) собрал и получил нужный результат, то как локальную IDE ни настраивай, результат сборки не испортишь и у каждого (и на сервере) он будет одинаковым.

◆ +5 Ответить Bookmark ...

Рис. 18.2: comment

Подробнее про гаражный колхоз сборки из-под GUI-IDE можно почитать тут:
Почему сборка из-под IDE это тупиковый путь

Какие недостатки сборки исходников из-под IDE?

1. IDE монолитные и неделимые. Если Вы захотите поменять какую-то часть ToolChain(а): препроцессор, компилятор или компоновщик, а остальные фазы

ToolChain(a) оставить как есть, то ничего из этого у вас не выйдет, так как капот IDE нагло закрыт на замок.

2. IDE стоят дорого, порядка 3500 EUR на один компьютер. Это лишняя дань и оброк для вашей компании. Оно Вам надо?

IAR Systems Lead: from Yandex

Reply Forward Delete

IAR Systems переслали мне новый запрос, который Вы оставили у них на сайте. Наше предложение, которое я направляла Вам в марте, остается в силе. Для работы с STM32F413ZG Вам будет достаточно редакции Cortex-M.

Cortex-M версия с привязкой к ПК
EWARM-CM цена по акции EUR 2730 (стандартная цена EUR 2925)

Cortex-M версия с привязкой к аппаратному ключу
EWARM-CM-MB цена по акции EUR 3375 (стандартная цена EUR 3600)

Cortex-M версия с привязкой к серверу (доступ по паролю)
EWARM-CM-NW цена по акции EUR 3444 (стандартная цена EUR 3690)

Cortex-M версия (EWARM-CM) ориентирована на работу с Arm Cortex-M0, M0+, M1, M3, M4, M7, M23, M33.

Возможности среды разработки включают поддержку Cortex-M процессоров производства российской компании Миландр. Информация отражена в списке поддерживаемых устройств на сайте производителя.

Лицензии бессрочные. Поставляются текущие версии. В качестве аппаратных средств отладки в дополнение к любому варианту EWARM можем предложить отладчик IJET с нашего склада.

В течение года после покупки IDE можно будет бесплатно обновлять версии (обычно 2-3 раза), а далее со скидкой 80% (запрос на обновление за месяц до окончания годовой SUA) и 50% (1-3 гола с момента покупки).

Рис. 18.3: Стоимость IAR

3. IDE(шные) xml очень слабо документированы или не документированы вовсе. У всех вендоров xml имеет свой особенный снобский xml-like язык разметки для конфигурации проекта. При внесении незначительных изменений появляется огромный git diff.
4. Затруднена сборка из консоли. В основном инициировать сборку в IDE можно мышкой или горячими клавишами. Либо надо делать refresh мышкой из-под IDE.
5. Обратная несовместимость с новыми версиями IDE.

Рис. 18.4: Добавил в дерево IDE ссылку на одну папочку. Получился такой diff. Как думаете сколько часов его будут review(вить) ? Ответ: в среднем две недели

6. В условиях технологического Эмбарго и Санкций законно купить IDE европейского (Германия, Швеция, США) вендора невозможно. Они Вас просто пошлют, так как у них Ваша территория числится как criminal state
 7. IDE отжирают какие-никакие но ресурсы компьютера, как RAM как и CPU, IDE же надо много оперативки, чтобы отрисовывать окошки со стразиками. IAR и Code Composer, например, раз в день стабильно напрочь зависают, что помогает лишь перезагрузка розеткой.
 8. Покарашенный xml файл настроек IDE приведит к тому, что IDE просто не открывается и всё. В результате проект даже не собрать.

В общем распространение IDE это яркий пример известного ныне "технологического диктата"(vendor locking) запада для низкосортных народов из стран второго и третьего мира.

Навязывание GUI-IDE (Keil, IAR, CCS, плагинов Eclipse) это форма технологического диктата со стороны стран бывших метрополий. Они привыкли столетиями помыкать своими колониями и до сих пор продвигают эту подлую циничную политику подсовывая vendor locking средств разработки во всякие страны как Кракожия, Такистан, Элбония, Западно-Африканская республика и прочее.

Сами они там у себя этим суррогатом нелепым не пользуются. Понимаете? У них там CMake, Make, Ninja скрипты сборки и полный DevOps в Docker контейнерах. Я работал в одной английской конторе и видел это всё своими глазами.

А туземцам в СНГ они суют эту тухлую песочницу в которой только кривые куличи лепить возможно.

Мы Вам продаём песочницу (IDE), а вы сидите там за бортиками, улыбаетесь и лепите свои, никому даром не нужные, куличики (прошивки-паршивки).



Рис. 18.5: иллюстрация программирования микроконтроллеров в IDE

Понятное дело, что разработчики IDE во всю пользуются ситуацией и добавляют всяческие программные закладки, такие как слия исходников в здание с пятью углами, удаленное отключение функционала, ограничение размера выходного бинарного файла до 32kByte и всё на, что им там только хватит их извращённой фантазии!

Понятное дело, что в таких жестких рамках на "сделать что-то серьезное" туземцам рассчитывать не приходится. Сборка в IDE это всегда мелкая серия. Всегда малый ассортимент. Всегда ручное развертывание.

А когда запад нас в очередной раз кинул пришлось начать думать как теперь дальше жить... Хорошим решением оказалось сделать шаг назад в 197x 198x когда на компьютерах всё делали из консоли. Даешь сборку сорцов из скриптов! Можно вообще *.bat файл написать и он, в общем-то, инициирует запуск нужных утилит, однако исторически Си-код собирали культовой утилитой make.

Дело в том, что makefile придумали в 1976 (Stuart Feldman), тогда, когда к компьютеры были дорогие (65k USD только за несколько сотен килобайт RAM) и к компьютерам подпускали только PhD профессоров из топовых университетов мира. Поэтому и появились такие гениальнейшие утилиты как awk, make, grep, find, gdb, sed, sort, tsort, uniq, tr и прочие.

Тогда в далеких 197x у школоты в принципе не было возможности хоть как-то влиять на ход развития софта и генерировать спагетти код и программные смеси как сейчас в 200x...202x.

Что можно делать из скриптов сборки? Когда Вы собираете из Make Вы можете не только собирать исходники, но и

1. Сортировать конфиги.
2. Запускать статический анализатор.
3. Собирать документацию (вызвать Latex, Doxygen).
4. Строить графы зависимостей на graphviz.
5. Автоматически архивировать артефакты.
6. Можете прямо из make отправлять файлы прошивок потребителям.
7. Вызывать процедуру пере прошивки консольными утилитами программатора.
8. Автоматически генерировать функцию инициализации программы.
9. Автоматически генерировать конфигурации.
10. Автоматически обновлять версию прошивки.
11. Подписывать артефакты.
12. Автоматически выравнивать отступы в исходниках.
13. Генерировать список макросов для подсветки синтаксиса.
14. и многое другое...

Утилите make всё равно какие консольные утилиты вызывать. Утилита make всеядная. Понимаете? Make - это универсальный способ определения программных конвейеров.

Утилиту make можно использовать не только для дирижирования процессом сборки кода программ. Утилита make может также управлять авто генерацией преобразования расширений файлов для черчения или управлять сборкой документации, управлять DevOps(ом). Make можно использовать по-разному, как только фантазии хватит. Ибо Make совершенно инвариантен и абстрагируется от языка программирования как такового.

Всё что от вас требуется это для каждой сборки надо написать крохотный Makefile

Листинг 18.1: Makefile

```
MK_PATH=$(dir $(realpath $(lastword $(MAKEFILE_LIST))))  
WORKSPACE_LOC=$(MK_PATH)../..  
  
INCLUDE_PATHS += -I$(MK_PATH)  
INCLUDE_PATHS += -I$(WORKSPACE_LOC)  
  
include $(MK_PATH) config.mk  
include $(MK_PATH) cli_config.mk  
include $(MK_PATH) diag_config.mk  
include $(MK_PATH) test_config.mk  
  
include $(WORKSPACE_LOC) code_base.mk  
include $(WORKSPACE_LOC) rules.mk
```

и конфиг для сборки.

Листинг 18.2: Конфиг для сборки

```
TARGET=pastilda_r1_1_generic
@echo $(error TARGET=$(TARGET))
AES256=Y
ALLOCATOR=Y
```

.....

```
USB_HOST_HS=Y
USB_HOST_PROC=Y
UTILS=Y
XML=Y
```

Для каждого компонента *.mk файл. Язык make простой. Он, в сущности, очень похож на bash. Вот типичный *.mk файл для драйвера UWB радио-трансивера DW1000.

Листинг 18.3: mk файл для UWB трансивера

```
ifeq ($(DWM1000_MK_INC),Y)
DWM1000_MK_INC=Y

DWM1000_DIR = $(DRIVERS_DIR)/dwm1000
$(info + DWM1000)

INCLUDE_PATHS += -I$(DWM1000_DIR)
OPT += -DHAS_DWM1000
OPT += -DHAS_DWM1000_PROC
OPT += -DHAS_UWB

DWM1000_RANGE_DIAG=Y
DWM1000_RANGE_COMMANDS=Y

DWM1000 OTP_COMMANDS=Y
DWM1000 OTP_DIAG=Y

SOURCES_C += $(DWM1000_DIR)/dwm1000_drv.c

include $(DWM1000_DIR)/otp/dwm1000_otp.mk
include $(DWM1000_DIR)/registers/dwm1000_registers.mk

ifeq ($(DWM1000_RANGE),Y)
    include $(DWM1000_DIR)/range/dwm1000_range.mk
endif

ifeq ($(DIAG),Y)
    ifeq ($(DWM1000_DIAG),Y)
        $(info +DWM1000_DIAG)
        OPT += -DHAS_DWM1000_DIAG
        SOURCES_C += $(DWM1000_DIR)/dwm1000_diag.c
    endif
endif
```

```

ifeq  ($(CLI),Y)
    ifeq  ($(DWM1000_COMMANDS),Y)
        $(info +DWM1000_COMMANDS)
        OPT += -DHAS_DWM1000_COMMANDS
        SOURCES_C += $(DWM1000_DIR)/dwm1000_commands.c
    endif
endif
endif

```

18.2 Как отлаживаться

Сейчас же в 201x...202x какой-нибудь 43-летний Junior-embedded программист микроконтроллеров, привыкший к GUI-IDE (IAR, Keil), может логично провозгласить:

"Я вообще не представляю, как без помощи IDE и зелёного треугольника вверху делать пошаговую отладку программы?"

Тут есть 4+ ответа:

1. Использовать связку GDB Client + GDB Server и отлаживать код из командной строки.
2. Отлаживать код через интерфейс командной строки CLI поверх UART.
3. В коем-то веке покрывать свой код модульными тестами
4. Использовать другие косвенные признаки отладки кода: HeartBeat LED, Утилита arm-none-eabi-addr2line.exe, Assert(ы), DAC, STM Studio (Аналог ArtMoney из GameDev(a)), Health Monitor

18.3 Достоинства сборки кода из Make файлов

В чем достоинства сборки С-кода из Make файлов?

1. Скрипты сборки GNU Make (или CMake) хороши тем, что в скриптах сборки Вы всегда можете написать текстовые комментарии. Вы можете рядом с каждым ключом компилятора или опцией компоновщика явно указать себе и коллегам подсказку, что это значит и для чего нужно. Понимаете?.. Одновременно с этим в GUI-IDE нет возможности в GUI форме указать в combo-box, что значит тот или иной ключ GCC. И это делает *.xml конфиг GUI-IDE абсолютно нечитаемым. Вот так...
2. Makefile это самый гибкий способ управлять модульностью кодовой базы. Можно буквально одной строчкой добавлять или исключать один конкретный программный компонент (десятка файлов) из десятков сборок. В случае же сборки из-под IDE Вам бы пришлось вручную мышкой редактировать xml для каждой отдельной сборки.

3. Вы можете спросить: "А зачем запускать Eclipse из консоли?" или "Зачем в принципе командная строка?" Ответ прост... Для автоматизации. Автоматика! Слыхали про такое? Вся суть любого программирования - это и есть пресловутая автоматизация чего либо. Даже автоматические построения самих проектов. Наработка артефактов.

А сборку из Makefile очень легко автоматизировать. Достаточно в консоли выполнить make all и у вас инициируется процесс сборки, а через 3 мин в соседней папке будут лежать артефакты.

При сборке из скриптов у вас будет одна кнопка. Жмакнул на *.bat скрипт - получил *.bin прошивку. Жмакнул на другой *.bat скрипт - прошил плату. Ну что может быть еще проще?

Лично я хочу чтобы после комита мои 256 сборок в репозитории собирались пока я сплю.

4. Если сборка на скриптах, то каждый может использовать абсолютно любой текстовый редактор или IDE. В этом основное достоинство сборки из скриптов. Я вот люблю текстовый редактор в Eclipse, сосед через стол не представляет жизни без Visual Studio Code от Microsoft. Мы собирали проект на GNU Make вообще без проблем. Мы оба для построения прошивки просто кликаем на built.bat скрипт, который запускает консольную команду make all.

5. После сборки из скриптов вы получите полный лог сборки, в то время как IDE обычно показывают последние 3-4 экрана сообщений компилятора.

6. Дело в том, что GNU Make скрипты пишутся только один раз. Потом только лишь так косметически меняются при добавлении очередных программных компонентов.

7. В MakeFile очень просто менять компиляторы. Это, буквально, заменить одну строчку. С GCC на Clang или на GHS. Вот типичный основной makefile для любой сборки на ARM Cortex-Mxx

8. Параллельное написание Make файлов и C-кода стимулирует придерживаться модульности, изоляции программных компонентов и к прослеживанию зависимостей между компонентами. Если вы пишите код и make файлы примерно параллельно, то очень вероятно, что у вас получится чистый, аккуратный репозиторий сам собой.

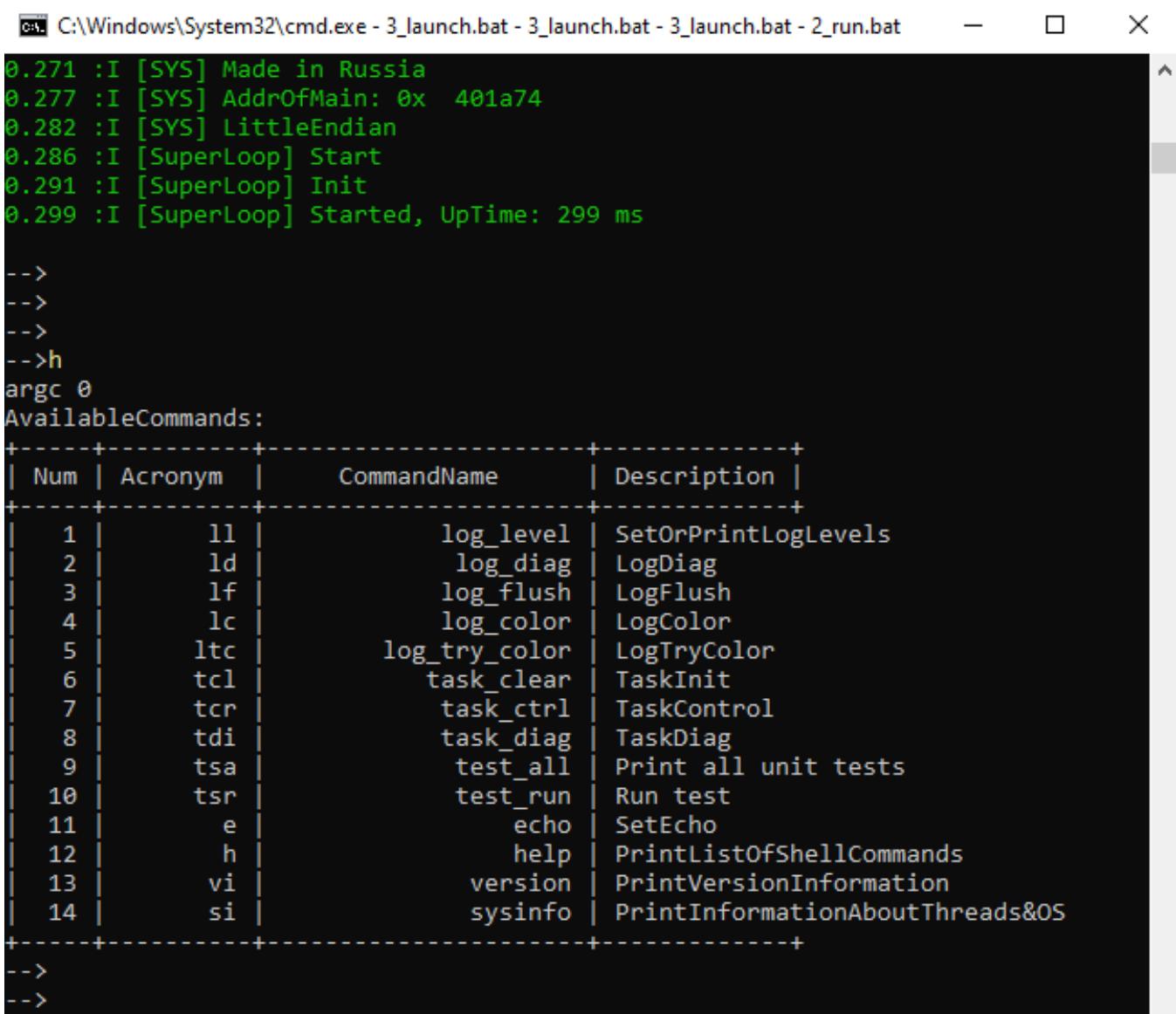
9. Makefile(лы) хороши тем, что можно добавить много проверок зависимостей и assert(ов) на фазе отработки Make-скриптов прямо в *.mk файлах еще до компиляции самого кода, даже до запуска препроцессора, так как язык программирования make поддерживает условные операторы и функции. Можно очень много ошибок отловить на этапе отработки утилиты make.

10. Язык Make очень прост. Во всяком случае много проще, чем тот же CMake. Вся спека GNU Make это всего 226 страниц. Для сравнения, спецификация CMake это 429 страниц!

11. Makefile(лы) прозрачные потому что текстовые. Всегда видно, где опции препроцессора, где ключи для компилятора, а где настройки для компоновщика.

Всё, что нужно можно найти утилитой gperf в той же консоли от GIT-bash даже при работе в Windows 10.

12. Конфиг для сборки можно формировать как раз на стадии make файлов и передавать их как ключи для препроцессора. Таким образом конфиги будут видны в каждом *.c файле проекта и не надо вставлять с конфигами. Всё можно передать как опции утилите cpp (препроцессора).
13. При сборке из makefile Вам вообще всё равно для какой целевой платформы собирать код прошивки. Вы можете минимальными изменениями в makefile собрать прошивку и крутить её даже на x86. Вместо UART имитировать CLI в Windows консольном приложении на PC.



```
C:\Windows\System32\cmd.exe - 3_launch.bat - 3_launch.bat - 3_launch.bat - 2_run.bat
0.271 :I [SYS] Made in Russia
0.277 :I [SYS] AddrOfMain: 0x 401a74
0.282 :I [SYS] LittleEndian
0.286 :I [SuperLoop] Start
0.291 :I [SuperLoop] Init
0.299 :I [SuperLoop] Started, UpTime: 299 ms

-->
-->
-->
-->h
argc 0
AvailableCommands:
+-----+-----+-----+
| Num | Acronym | CommandName | Description |
+-----+-----+-----+
| 1   | ll     | log_level | SetOrPrintLogLevels
| 2   | ld     | log_diag  | LogDiag
| 3   | lf     | log_flush | LogFlush
| 4   | lc     | log_color | LogColor
| 5   | ltc    | log_try_color | LogTryColor
| 6   | tcl    | task_clear | TaskInit
| 7   | tcr    | task_ctrl  | TaskControl
| 8   | tdi    | task_diag  | TaskDiag
| 9   | tsa    | test_all   | Print all unit tests
| 10  | tsr    | test_run   | Run test
| 11  | e      | echo       | SetEcho
| 12  | h      | help       | PrintListOfShellCommands
| 13  | vi    | version    | PrintVersionInformation
| 14  | si    | sysinfo   | PrintInformationAboutThreads&OS
+-----+-----+-----+
-->
-->
```

Рис. 18.6: эмулятор прошивки в консольном приложении

14. Внутри makefile вы можете выполнить какой-нибудь полезный скрипт. Например подписать прошивку состоянием репозитория

Листинг 18.4: подписать прошивку состоянием репозитория

```
GIT_SHA := $(shell git rev-parse --short HEAD)
OPT += -DGIT_SHA=0x$$(GIT_SHA)
```

таким образом, 3-мя строчками вы сделаете то, что отдельными скриптами заняло бы 50+ строк.

Листинг 18.5: Показать в логе загрузки версию репозитория

```
LOG_INFO(SYS, "GitSha: 0x%08x", GIT_SHA);
```

15. При сборке из скриптов (например из Make) очень легко добавить новую сборку. Достаточно только написать конфиг и 3 строчки в отдельном Makefile и вот у вас новая специфическая сборка. Одновременно с этим создание новой сборки в GUI-IDE сопряжено с многочисленной мышковозней и дублированием конфигов!



Рис. 18.7: скриптами проще добавить новую сборку в сравнении с IDE

16. Сборка скриптами хороша тем, что у GNU Make очень понятные сообщения об ошибках. GNU Make буквально пишет номер строки и причину ошибки. В свою очередь покарашенный xml файл привидит к тому, что IDE просто молча не открывается и всё.

18.4 Аналогии

Если проводить аналогию с атомной энергетикой то, сборка через GUI-IDE - это как реакторы на горизонтальных ТВЭЛах (сборках), а сборка из скриптов это реактор на вертикальных ТВЭЛах.

Горизонтальные реакторы неудобны, так как надо минимум две точки подвеса ТВЭЛА при загрузке топлива. Плюс нужны дополнительные усилия, чтобы проталкивать стержни в активную зону. Потом стержень может расплавиться и застрять. Тогда жди большой беды.

Напротив, вертикальные ТВЭЛы загружаются под действие силы тяжести и для транспортировке сборки нужна только одна точка подвеса. А расплавленные ТВЭЛы просто стекают в поддон. Easy!

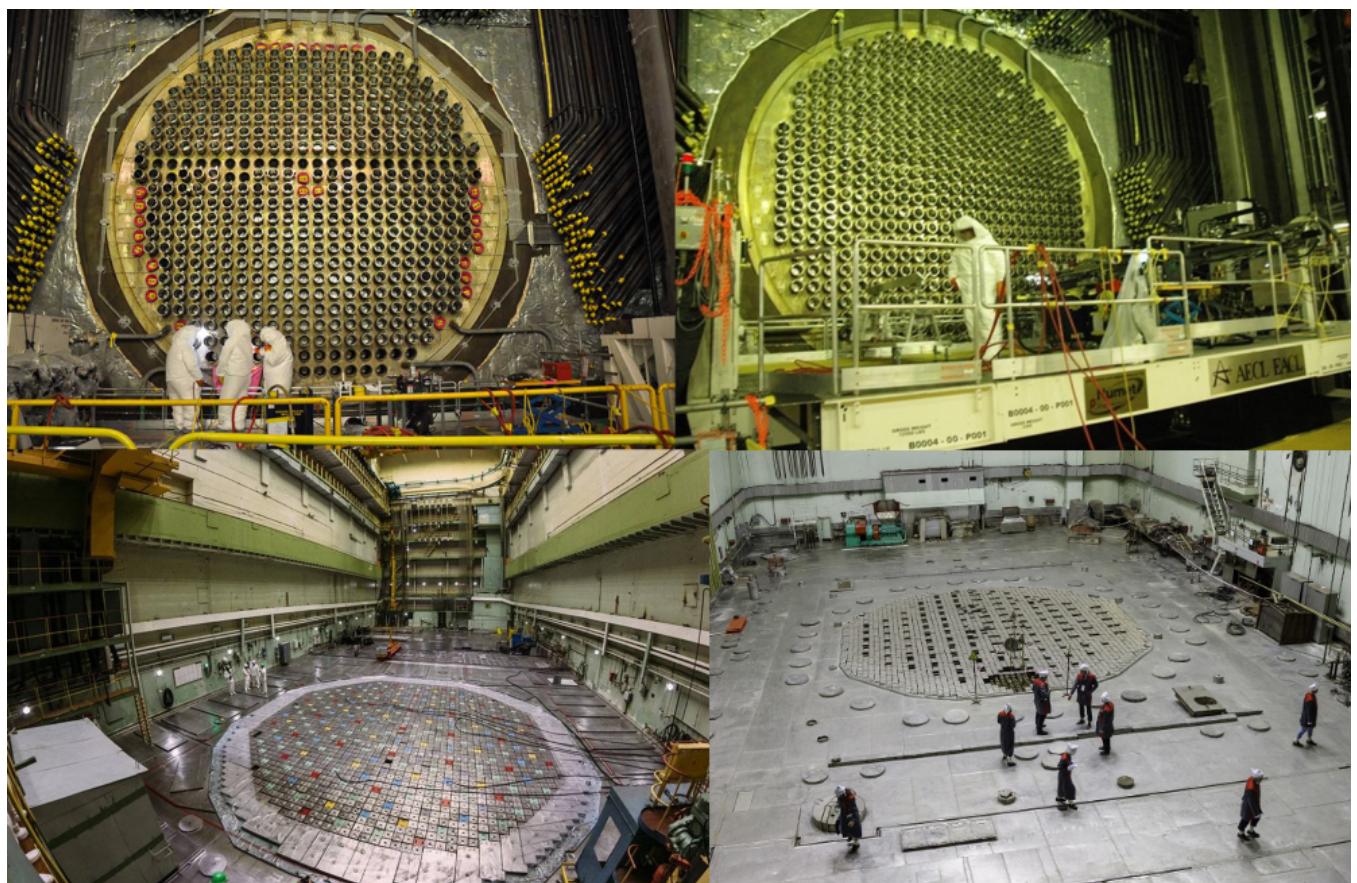


Рис. 18.8: Горизонтальные ТВЭЛы труднее загружать чем вертикальные ТВЭЛы

Make скрипты - это как катализатор в химии. Благодаря GNU Make всё происходит быстрее.

Скрипты сборки Make - это как стапели, но не для корабля, а для программы.

Если проводить аналогию между программированием микроконтроллеров и полиграфией, то GUI-IDE - это как Microsoft Word, а скрипты сборки - это LaTeX. Понимаете?

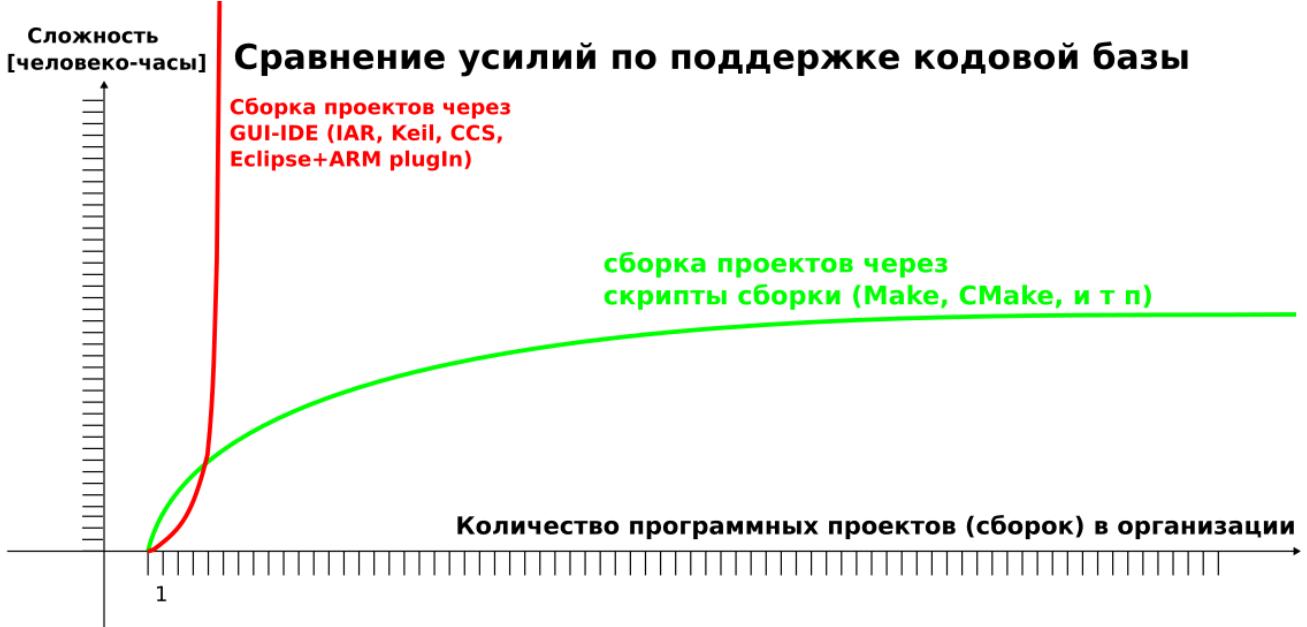


Рис. 18.9: скриптами проще добавать новую сборку чем в IDE

18.5 CMake или GNU Make?

CMake (Cross-platform Make) был разработан для кросс-платформенности. Переводя на кухонный язык, это чтобы одну и ту же программу можно было собирать в разнообразных операционных системах Windows, Linux, MacOS, FreeBSD. Если же Вы все в своей организации работаете только в Windows 10, то Вам CMake нужен как собаке бензобак. Да.. Именно так.. Вам много проще будет самим писать GNU Make скрипты, раз нужна только сборка по клику на *.bat файл.

Потом, если вы до этого никогда не писали никаких скриптов сборки, то сразу кидаться писать CMake пожалуй тоже нет резона. Дело в том, что CMake это даже не система сборки, а надстройка над всеми возможными системами сборки: Ninja, Make и проч. Как AutoTools. CMake, в зависимости от опций командной строки, сам генерирует скрипты сборки. CMake - это всего лишь кодогенератор. И Вам будет с непривычки будет архи сложно разобраться с огромным калейдоскопом разнообразных и новых для себя расширений файлов: *.cmake, CMakeList.txt, *.c.in *.mk и прочие. CMake очень навороченная и переизбыточная система.

Если у вас цель просто собирать код дергая скрипты в Jenkins, то лучше сосредоточиться на GNU Make. Тогда у вас в репозитории фактически будут только три типа файлов для версионного контроля: *.c *.h и *.mk файлы. Easy!

CMake же - это очень навороченная утилита и там много избыточного функционала. Много того, что Вам никогда даже не пригодится. В случае выбора CMake Вам, например, придется помимо ошибок компилятора чинить ещё ошибки отработки CMake скриптов, которые по логу порой даже понять трудно, потом чинить ошибки GNU Make, потом чинить ошибки компилятора, чинить ошибки компоновщика. Ещё CMake перед сборкой занимается тестированием компилятора. В результате долго отрабатывает скрипт. CMake работает в два прогона. Да и сами

скрипты сборки make которые выращивает CMake получаются очень грязными. С душком.... Да они работают, но читать их человеку просто не-ре-аль-но....

Лучше, быстрее и надежнее просто ликвидировать лишнюю стадию отработки CMake скриптов и просто самим взять и написать лаконичные скрипты сборки GNU Make. С точки зрения DevOps результат будет абсолютно тем же: сборка из скрипта, по клику. А раз так, то зачем платить больше? Не надо слишком сильно увлекаться FrameWork(о) строительством.

В GNU Make есть всё, что нужно для полноценной автоматизации: топологическая сортировка целей, регулярные выражения, выявление не поменявшихся файлов, функции, операторы. На GNU Make решаются 99,99 процентов всех задач по DevOps(y).

18.6 Кто собирает код из скриптов?

Почему в программировании микроконтроллеров, да ещё и в России не особо прижились скриптовые системы сборки?

Ответ прост. Прошивки это маленькие программы. Типичные bit(арь) обычно порядка 128kByte. Даже самая крупная прошивка собирается максимум за 3 мин. Её всегда можно из-под IDE собрать кликнув курсором мышки на зелёный треугольник.

А скриптовые системы сборки прежде всего изначально использовались в циклопических программных продуктах: BackEnd сайтов, DeskTop ПО, CAD системы, GameDev. Там *.exe бинари могут запросто быть по 2Gbyte и более. Да и компы в 1970....1990 слабые были, не то что сейчас. Один проект мог собираться три часа.

Естественно проектов много и собирают артефакты крупных проектов по ночам, пока программисты спят. Утром чинят ошибки компиляции, днем пилят функционал, вечером делают коммиты. Собирают автоматически дергая скрипты сборки на сервере типа Jenkins.

Однако и в embedded разработке сборку из скриптов уже оценил ряд серьезных и успешных российских организаций. Вот, например, Whoosh написал даже текст про свой опыт настройки DevOps: Сборка и отладка прошивки IoT-модуля: Python, make, апельсины и чёрная магия. Ещё Wiren Board выступали на конференции с докладом CI/CD прошивок для микроконтроллеров в Wiren Board.

Я знаю и другие компании, которые пользуются этой фундаментальной технологией, но пока не пишут про это сообществах.

18.7 Вывод

В сухом остатке, в наше время бахвалиться навыками пользования всяческими IDE должно быть уже стыдно. Надо признать, что сборка средствами GUI-IDE (IAR, CCS, KEIL, Eclipse+Plugins) - это уровень кружка робототехники 8-9го класса средней общеобразовательной школы (ГОУ СОШ).

Сборка прошивок GUI-IDE плагинами, а также из-под Keil, IAR, CCS - это признак Junior разработчика.

А сборка из скриптов - это, господа, как ни крути, но фундаментальная технология, которая по плечу только программистам с качественным опытом. Не путать с количественным опытом, когда программист микроконтроллеров просто просиживает штаны по 11 лет на одной работе-Богодельне.

Да, написание make скриптов требует некоторого образования и сноровки, но это плата за масштабирование, автосборки, единообразие, наследование конфигов и модульность репозитория. Усилия инвестированные в изучение make окупаются сторицей!

Потом санкционные реалии таковы, что настало время, чтобы российских программистов микроконтроллеров из детского садика под названием "GUI-IDE" перевести, наконец, в школу (т.е. приучить к makefile или хотя бы к CMake). А дальше приобщать к полноценному DevOps(у). А в идеале к чему-то типа Yocto project.

При этом надо смотреть в сторону Make, CMake. Как вариант, Ninja.

Понимаете, хорошие вещи как классика не устаревают. Вы же сами каждый день пользуетесь пуговицами... А пуговицы, между прочим, вообще в средневековье придумали... Что теперь, давайте без пуговиц ходить что-ли? Make это как пуговицы. Старая, простая и очень полезная вещь.

Сборка программ из скриптов это фундаментальная технология, величайшее достижение нашей эпохи.

Откровенно говоря, только тех, кто умеет собирать проекты из скриптов и можно считать настоящими программистами. А те кто как бы числится программистом и не умеет читать писать скрипты сборки, это либо самозванцы, устроившиеся на работу по блату или обыкновенная шко-ло-та. Без обид, но, что есть, то есть.

Я естественно понимаю, что это может неприятно читать такое, особенно когда тебе уже далеко за 40 и ты всегда собирал прошивки через GUI-IDE, мышкой, и ёщё не освоил в своей жизни никаких систем сборок. Даже CMake. Неграмотный в вопросах систем сборки. Однако учиться никогда не поздно. Для этого есть выходные, отпуска, государственные праздники. "Дорогу осилит идущий так ведь говорят?

Вы же не обижаетесь на некоторые не очень приятные явления природы. Например лютый мороз, радиоактивность, молнии, цунами, торнадо, наводнения, землетрясения и прочее. Но они, как ни крути, объективно существуют и к этому надо просто приспосабливаться.

При сборке из makefile прошивки для микроконтроллеров любых vendor(ов) собираются абсолютно одинаково. Будь-то RISC-V, ST(stm32), cc26x42, AVR (atmega8), Artery, Nordic (nrf53) или spc58nn. Надо просто открыть консоль и набрать make all. Easy! Как в песенке поется: "нажми на кнопку, получишь результат, и твоя мечта осуществится!"

Когда у тебя в организации только одна, максимум 4 прошивки, то в общем-то для работы и GUI-IDE достаточно. А скрипты так, полезный ликбез. Но вот если сборок много, десятки, сотни. Что чаще всего и получается. Когда надо

поддерживать на плаву прошлые проекты. Когда организация выпускает большой ассортимент продуктов и разрабатывает новые версии. То тут, друг, как ни крути, но нужны уже скрипты сборки. Да... Не зря же их придумали в своё время.

Есть два культовых доклада в ИТ конференциях, которые поясняют откуда взялся этот график. Находятся по названиям по первым ссылкам.

1. CI/CD прошивок для микроконтроллеров в WIREN BOARD (начало на 25:20)
2. Конвеерум 30: Эволюция рабочего окружения для embedded разработки

Собираете свои прошивки из самостоятельно написанных make скриптов, господа, в этом нет абсолютно ничего сложного!

Так как make появился ещё в 1976, то это пожалуй самая изученная тема во всем Computer Science. Материалов для ознакомления, обучения и освоения make просто немерено. Океан информации.

18.8 Гиперссылки

1. Настройка ToolChain(a) для Win10++C+Makefile+ARM Cortex-Mx+GDB
2. Пример Makefile
3. Эффективное использование GNU Make
4. Обновление Прошивки из Make Скрипта
5. CI/CD прошивок для микроконтроллеров в WIREN BOARD (начало на 25:20)
6. Конвеерум 30: Эволюция рабочего окружения для embedded разработки
7. GNU Make может больше чем ты думаешь

"Переносимая кодовая база - это плацдарм для будущих разработок."

Глава 19

ЛикБез по GNU Make

GNU Make - это консольная утилита, которая запускает другие консольные утилиты в желаемой последовательности. Только и всего. Конфигом для утилиты make является текстовый файл-скрипт хранящийся в файле по имени Makefile. Скрипт - это программа для интерпретатора. Поэтому утилите GNU Make называют системой сборки. Можно сказать, что Make - это интерпретатор языка make.

Прелесть утилиты make в том, что ей абсолютно всё равно с каким языком программирования работать. Более того, утилите make всё равно с какие утилиты вызывать. Утилита make всеядная. Понимаете?

С математической точки зрения, make делает топологическую сортировку ориентированного графа. Makefile прописывает список смежности вершин. Запускается команда make all, и утилита проходит все вершины в порядке ориентированного графа. По сути makefile определяет конвейер вызова утилит для метамарфоза файлов из одного расширения в другое расширение прямо на жестком диске РС.

При помощи make можно даже автоматически синтезировать инструкции по сборке самолётов или домов из миллионов деталей. Достаточно просто в make файле указать, что стоит соединять с чем. Затем вызвать make all и у вас появится текстовый файл с логом корректной инструкций сборки этого самолёта. Корректная инструкция в том плане, что у вас не будет такой ситуации, что очередная деталь упирается в узкий проём и её не вставить, из-за чего надо опять разбирать, скажем, пол двигателя.

Всё то же самое происходит и в сборке компьютерных программ. Нет смысла генерировать bin файл, когда нет elf файла. Нет смысла генерировать elf файл, когда нет obj файла. И тому подобное.

Достоинство make в том, что он пересобирает только те файлы, которые были изменены. Времена последней модификации, читаются от файловой системы. То есть в make с самого начала были заложены элементы контроля версий.

Признаком комментария является символ решетка . Пустые строки и строки, начинающиеся с , игнорируются.

19.1 Переменные make

Как и в любом скрипте make позволяет создавать переменные. Например `CFLAGS`. Получить доступ к переменной можно заключив ее в круглые скобки.

Листинг 19.1: Показать содержимое переменной `CFLAGS`

```
$(error CFLAGS=$(CFLAGS))
```

Листинг 19.2: специальные переменные

```
$@ -  
$^ -  
  
$< -  
  
$* -  
  
$? - ,  
      ,  
$+ - $^, ,  
      ,  
      ,  
      ,  
$% -
```

19.2 Как организовать скрипты сборки GNU Make?

Любой GNU Make скрипт начинается с `Makefile`. У каждой сборки есть свой `Makefile`. По хорошему `Makefile` должен быть очень маленьким. Все общие скрипты должны быть вынесены за скобки в отдельные `*.mk` файлы. Да, GNU Make поддерживает свой собственный препроцессор. Поэтому вы будете часто встречать ключевое слово `include`.

Листинг 19.3: Корневой `Makefile`

```
MK_PATH:=$(dir $(realpath $(lastword $(MAKEFILE_LIST))))  
WORKSPACE_LOC:=$(MK_PATH)../..  
WORKSPACE_LOC:=$(realpath $(WORKSPACE_LOC))  
MK_PATH:=$(realpath $(MK_PATH))  
#@echo $(error MK_PATH=$(MK_PATH))  
INCDIR += -I$(MK_PATH)  
INCDIR += -I$(WORKSPACE_LOC)  
  
DEPENDENCIES_GRAPHVIZ=Y
```

```

TARGET=boardname_configname_gcc_m

include $(MK_PATH)/config.mk

ifeq ($(CLI),Y)
    include $(MK_PATH)/cli_config.mk
endif

ifeq ($(DIAG),Y)
    include $(MK_PATH)/diag_config.mk
endif

ifeq ($(TEST),Y)
    include $(MK_PATH)/test_config.mk
endif

include $(WORKSPACE_LOC)/make_scripts/code_base.mk
include $(WORKSPACE_LOC)/make_scripts/rules.mk

```

Тут функция realpath вычисляет абсолютный путь учитывая арифметику над путями.

При сборке из make по сути все сборки в репозитории отличаются только одним лишь файлом. Это config.mk. Вот так он выглядит.

Листинг 19.4: Конфиг сборки

```

CLI=Y
DEBUG=Y
DEPENDENCIES_GRAPHVIZ=Y
GENERIC=Y
GPIO=Y
LED_MONO=Y
.....
TIME=Y
UART2=Y
UNIT_TEST=Y
YTM32B1ME05G0MLQ=Y
YTM32B1M_EVB_0144_REV_B=Y

```

Внутри config.mk декларативно перечисляется из каких программных компонентов должна состоять данная программа.

Листинг 19.5: Конфиг для диагностики

```

$(info Add Diag)

DIAG=Y
LOG_DIAG=Y

ifeq ($(ALLOCATOR),Y)
    ALLOCATOR_DIAG=Y
endif

ifeq ($(CORTEX_M33),Y)
    CORTEX_M33_DIAG=Y
endif

```

```
.....  
ifeq $(TIMER) ,Y  
    TIMER_DIAG=Y  
endif
```

```
ifeq $(WATCHDOG) ,Y  
    WATCHDOG_DIAG=Y  
endif
```

Листинг 19.6: Конфиг для CLI

```
$(info CLI_CONFIG_MK_INC=$(CLI_CONFIG_MK_INC) )
```

```
ifneq $(CLI_CONFIG_MK_INC) ,Y  
    CLI_CONFIG_MK_INC=Y
```

```
    CLI_CMD_HISTORY=Y  
    CLI=Y
```

```
ifeq $(BUTTON) ,Y  
    BUTTON_COMMANDS=Y  
endif
```

```
ifeq $(NVIC) ,Y  
    NVIC_COMMANDS=Y  
endif
```

```
.....
```

```
ifeq $(UART) ,Y  
    UART_COMMANDS=Y  
endif
```

```
ifeq $(UNIT_TEST) ,Y  
    UNIT_TEST_COMMANDS=Y  
endif
```

```
ifeq $(WATCHDOG) ,Y  
    WATCHDOG_COMMANDS=Y  
endif
```

```
endif
```

У каждого программного компонента свой отдельный make скриптов сборки. Все они будут выглядеть одинаково. Отличие только в одном ключевом слове.

Листинг 19.7: Скрипт сборки программного компонента

```
$(info BUTTON_MK_INC=$(BUTTON_MK_INC))
```

```
ifneq $(BUTTON_MK_INC) ,Y  
    BUTTON_MK_INC=Y
```

```
BUTTON_DIR = $(SENSITIVITY_DIR)/button_fsm  
#@echo $error BUTTON_DIR=$(BUTTON_DIR)
```

```
INCDIR += -I$(BUTTON_DIR)
```

```

SOURCES_C += $(BUTTON_DIR)/button_drv.c

BUTTON=Y
OPT += -DHAS_BUTTON
OPT += -DHAS_BUTTON_PROC

ifeq ($(BUTTON_DIAG),Y)
    OPT += -DHAS_BUTTON_DIAG
    SOURCES_C += $(BUTTON_DIR)/button_diag.c
endif

ifeq ($(CLI),Y)
    ifeq ($(BUTTON_COMMANDS),Y)
        OPT += -DHAS_BUTTON_COMMANDS
        SOURCES_C += $(BUTTON_DIR)/button_commands.c
    endif
endif
endif

```

Это скрипт с правилами сборки проекта. Его прелесть в том, что он общий для всех сборок в репозитории. Обратите внимание на ключевое слово vpath. Оно позволяет перенаправлять объектные файлы в папку build и, тем самым, не засорять репозиторий временными объектными файлами.

Листинг 19.8: Правила сборки rules.mk

```

CSTANDARD = -std=c99
#CSTANDARD = -std=c11 c99
#CSTANDARD = -std=gnu99

mkfile_path := $(abspath $(lastword $(MAKEFILE_LIST)))
$(info mkfile_path:$(_mkfile_path) )
MK_PATH := $(subst /cygdrive/c/,C:/, $(MK_PATH))
$(info MK_PATH=$(MK_PATH))

BUILD_DIR=build

EXTRA_TARGETS=

INCDIR := $(subst /cygdrive/c/,C:/, $(INCDIR))
SOURCES_TOTAL_C += $(SOURCES_C)
SOURCES_TOTAL_C += $(SOURCES_CONFIGURATION_C)
SOURCES_TOTAL_C += $(SOURCES_THIRD_PARTY_C)
SOURCES_TOTAL_C := $(subst /cygdrive/c/,C:/, $(SOURCES_TOTAL_C))

SOURCES_ASM := $(subst /cygdrive/c/,C:/, $(SOURCES_ASM))

LIBS := $(subst /cygdrive/c/,C:/, $(LIBS))
LDSCRIPT := $(subst /cygdrive/c/,C:/, $(LDSCRIPT))

WORKSPACE_LOC := $(realpath $(WORKSPACE_LOC))
WORKSPACE_LOC := $(subst /cygdrive/c/,C:/, $(WORKSPACE_LOC))

include $(WORKSPACE_LOC)/make_scripts/toolchain.mk

AS_DEFS =
AS_INCLUDES =

```

```

MICROPROCESSOR += $(CPU)
MICROPROCESSOR += $(FPU)
MICROPROCESSOR += $(FLOAT-ABI)

include $(WORKSPACE_LOC)/make_scripts/compiler_options.mk
include $(WORKSPACE_LOC)/make_scripts/linker_options.mk

ASFLAGS += $(MCU)
ASFLAGS += $(AS_DEFS)
ASFLAGS += $(AS_INCLUDES)
ASFLAGS += $(OPT)
ASFLAGS += $(COMPILE_OPT)
ASFLAGS += -Wall
ASFLAGS +=-fdata-sections
ASFLAGS +=-ffunction-sections

CPP_FLAGS += $(CSTANDARD) $(INCDIR) $(OPT) $(COMPILE_OPT)

EXTRA_TARGETS += generate_definitions

ARTIFACTS += $(BUILD_DIR)/$(TARGET).bin
ARTIFACTS += $(BUILD_DIR)/$(TARGET).hex
ARTIFACTS += $(BUILD_DIR)/$(TARGET).elf

.PHONY: all

all: $(EXTRA_TARGETS) $(ARTIFACTS)

.PHONY: generate_definitions

generate_definitions:
    $(info GenerateDefinitions...)
    $(PREPROCESSOR_TOOL) $(CPP_FLAGS) $(WORKSPACE_LOC)/empty_source.c -dM -E>
        cDefinesGenerated.h
    $(SORTER_TOOL) -u cDefinesGenerated.h -o cDefinesGenerated.h

OBJECTS = $(addprefix $(BUILD_DIR)/,$(notdir $(SOURCES_TOTAL_C:.c=.o)))
vpath %.c $(sort $(dir $(SOURCES_TOTAL_C)))

# list of ASM program objects
OBJECTS += $(addprefix $(BUILD_DIR)/,$(notdir $(SOURCES_ASM:.S=.o)))
vpath %.S $(sort $(dir $(SOURCES_ASM)))

TOTAL_FILES := $(words $(OBJECTS))
$(info TOTAL_FILES: $(TOTAL_FILES))

$(BUILD_DIR)/%.o: %.c Makefile | $(BUILD_DIR)
    $(eval CURRENT_CNT=$(shell echo $$(( $(CURRENT_CNT)+1))))
    @echo Compiling $(CURRENT_CNT)/$(TOTAL_FILES) $@
    @$(CC) -c -MD $(CFLAGS) -Wa,-a,-ad,-alms=$(BUILD_DIR)/$(notdir $(<..c=.lst)) $<-o $@

$(BUILD_DIR)/%.o: %.S Makefile | $(BUILD_DIR)
    $(AS) -c $(CFLAGS) $<-o $@

```

```

$(BUILD_DIR) / $(TARGET).elf: $(OBJECTS) Makefile
    $(CC) $(OBJECTS) $(LDFLAGS) -o $@
    $(SZ) $@

$(BUILD_DIR) /%.hex: $(BUILD_DIR) /%.elf | $(BUILD_DIR)
    $(HEX) $< $@

$(BUILD_DIR) /%.bin: $(BUILD_DIR) /%.elf | $(BUILD_DIR)
    $(BIN) $< $@

$(BUILD_DIR):
    mkdir -p $@

.PHONY: clean

clean:
    -rm -fR $(BUILD_DIR)

# dependencies
-include $(wildcard $(BUILD_DIR) /*.d)

```

Как можно заметить, в зависимости от значения переменных окружения, make скрипт добавить в сборку те или иные программные компоненты. Каждый программный компоненте оформляется как *.mk файл. Внутри *.mk файла происходит добавление файлов исходников в переменную окружения SOURCES_{COPT}.

Листинг 19.9: Настройка кодовой базы

```

ifeq ($(CODE_BASE_MK),Y)
    CODE_BASE_MK=Y
    DEPENDENCIES_GRAPHVIZ=Y

    include $(WORKSPACE_LOC)/make_scripts/code_base_preconfig.mk
    include $(WORKSPACE_LOC)/make_scripts/verify_build.mk

    INCDIR += -I$(WORKSPACE_LOC)
    $(info WORKSPACE_LOC=$(WORKSPACE_LOC))

    GIT_SHA := $(shell git rev-parse --short HEAD)
    OPT += -DGIT_SHA=0x0$(GIT_SHA)
    OPT += -DHAS_MAKEFILE_BUILD

    ifeq ($(NORTOS),Y)
        OPT += -DHAS_NORTOS
    endif

    ifeq ($(MULTIMEDIA),Y)
        OPT += -DHAS_MULTIMEDIA
    endif

    ifeq ($(DEBUG),Y)
        OPT += -DHAS_DEBUG
    endif

    ifeq ($(DIAG),Y)
        OPT += -DHAS_DIAG
    endif

```

```

endif

ifeq  ($(MODULES) ,Y)
    include $(WORKSPACE_LOC) / modules / modules .mk
endif

ifeq  ($(TERMINAL) ,Y)
    include $(WORKSPACE_LOC) / make_scripts / terminal .mk
endif

ifeq  ($(MBR) ,Y)
    SUPER_CYCLE=Y
    include $(WORKSPACE_LOC) / make_scripts / mbr .mk
endif

ifeq  ($(GENERIC) ,Y)
    include $(WORKSPACE_LOC) / make_scripts / generic .mk
endif

ifeq  ($(MICROCONTROLLER) ,Y)
    FIRMWARE=Y
    include $(WORKSPACE_LOC) / microcontroller / microcontroller .mk
endif

ifeq  ($(BOARD) ,Y)
    include $(WORKSPACE_LOC) / boards / boards .mk
endif

ifeq  ($(PROTOTYPE) ,Y)
    include $(WORKSPACE_LOC) / prototypes / prototypes .mk
endif

ifeq  ($(X86) ,Y)
    SUPER_CYCLE=Y
    OPT += -DX86
    OPT += -DHAS_X86
    FLOAT_UTILS=Y
endif

ifeq  ($(X86_64) ,Y)
    SUPER_CYCLE=Y
    #@echo $(error stop)
    OPT += -DX86_64
    OPT += -DHAS_X86_64
endif

ifeq  ($(THIRD_PARTY) ,Y)
    include $(WORKSPACE_LOC) / third_party / third_party .mk
endif

ifeq  ($(CORE) ,Y)
    include $(WORKSPACE_LOC) / core / core .mk
endif

ifeq  ($(APPLICATIONS) ,Y)
    include $(WORKSPACE_LOC) / applications / applications .mk

```

```

endif

ifeq  ($(MCAL) ,Y)
    include $(WORKSPACE_LOC) / mcal / mcal.mk
endif

ifeq  ($(ADT) ,Y)
    include $(WORKSPACE_LOC) / adt / adt.mk
endif

ifeq  ($(CONNECTIVITY) ,Y)
    include $(WORKSPACE_LOC) / connectivity / connectivity.mk
endif

ifeq  ($(CONTROL) ,Y)
    include $(WORKSPACE_LOC) / control / control.mk
endif

ifeq  ($(COMPONENTS) ,Y)
    include $(WORKSPACE_LOC) / components / components.mk
endif

ifeq  ($(COMPUTING) ,Y)
    include $(WORKSPACE_LOC) / computing / computing.mk
endif

include $(WORKSPACE_LOC) / compiler / compiler.mk

ifeq  ($(SENSITIVITY) ,Y)
    include $(WORKSPACE_LOC) / sensitivity / sensitivity.mk
endif

ifeq  ($(STORAGE) ,Y)
    include $(WORKSPACE_LOC) / storage / storage.mk
endif

ifeq  ($(SECURITY) ,Y)
    include $(WORKSPACE_LOC) / security / security.mk
endif

ifeq  ($(ASICS) ,Y)
    include $(WORKSPACE_LOC) / asics / asics.mk
endif

ifeq  ($(MISCELLANEOUS) ,Y)
    include $(WORKSPACE_LOC) / miscellaneous / miscellaneous.mk
endif

ifeq  ($(UNIT_TEST) ,Y)
    include $(WORKSPACE_LOC) / unit_tests / unit_test.mk
endif

SOURCES_C += $(WORKSPACE_LOC) / main.c
endif

```

Сборка прошивок это всегда коросс-компиляция. Поэтому надо в make скрипте явно указать каким именно компилятором мы будем собирать исходные тексты

программ (сорцы).

Листинг 19.10: Выбор Toolchain-a

```
PREFIX = arm-none-eabi-
$(info GCC_PATH=$(GCC_PATH))

ifdef GCC_PATH
$(info WithPath)
PREPROCESSOR_TOOL=$(GCC_PATH)$(PREFIX)cpp
CC = $(GCC_PATH)$(PREFIX)gcc
CPP = $(GCC_PATH)$(PREFIX)g++
AS = $(GCC_PATH)$(PREFIX)gcc -x assembler-with-cpp
CP = $(GCC_PATH)$(PREFIX)objcopy
SZ = $(GCC_PATH)$(PREFIX)size
else
$(info WithOutPath)
PREPROCESSOR_TOOL = $(PREFIX)cpp
CC = $(PREFIX)gcc
CPP = $(PREFIX)g++
AS = $(PREFIX)gcc -x assembler-with-cpp
CP = $(PREFIX)objcopy
SZ = $(PREFIX)size
endif

HEX = $(CP) -O ihex
BIN = $(CP) -O binary -S
```

Компилятор это консольная утилита. Как и у любой консольной утилиты в gcc есть опции командной строки. Вот типичный пучок опций для сборки прошивок на MCU. Что значит каждая опция компилятора можно посмотреть в документе Using the GNU Compiler Collection.

Листинг 19.11: Опции компилятора

```
COMPILE_OPT += -Wall
COMPILE_OPT += -fdata-sections
COMPILE_OPT += -ffunction-sections
COMPILE_OPT += -Werror=address
COMPILE_OPT += -Werror=switch
COMPILE_OPT += -Werror=array-bounds=1
COMPILE_OPT += -Werror=comment
COMPILE_OPT += -Werror=div-by-zero
COMPILE_OPT += -Werror=duplicated-cond
COMPILE_OPT += -Werror=shift-negative-value
COMPILE_OPT += -Werror=duplicate-decl-specifier
COMPILE_OPT += -Werror=enum-compare
COMPILE_OPT += -Werror=uninitialized
COMPILE_OPT += -Werror=empty-body
COMPILE_OPT += -Werror=unused-but-set-parameter
COMPILE_OPT += -Werror=unused-but-set-variable
COMPILE_OPT += -Werror=float-equal
COMPILE_OPT += -Werror=logical-op
COMPILE_OPT += -Werror=implicit-int
COMPILE_OPT += -Werror=implicit-function-declaration
```

```

COMPILE_OPT += -Werror=incompatible-pointer-types
COMPILE_OPT += -Werror=int-conversion
COMPILE_OPT += -Werror=old-style-declaration
COMPILE_OPT += -Werror=maybe-uninitialized
COMPILE_OPT += -Werror=redundant-decls
COMPILE_OPT += -Werror=sizeof-pointer-div
COMPILE_OPT += -Werror=misleading-indentation
COMPILE_OPT += -Werror=missing-declarations
COMPILE_OPT += -Werror=missing-parameter-type
COMPILE_OPT += -Werror=overflow
COMPILE_OPT += -Werror=parentheses
COMPILE_OPT += -Werror=pointer-sign
COMPILE_OPT += -Werror{return-type}
COMPILE_OPT += -Werror=shift-count-overflow
COMPILE_OPT += -Werror=strict-prototypes
COMPILE_OPT += -Werror=unused-but-set-variable
COMPILE_OPT += -Werror=unused-function
COMPILE_OPT += -Werror=unused-variable
COMPILE_OPT += -Werror=type-limits
COMPILE_OPT += -Werror=override-init
COMPILE_OPT += -Werror=duplicate-decl-specifier
COMPILE_OPT += -Werror=int-conversion
COMPILE_OPT += -Wno-stringop-truncation
COMPILE_OPT += -Wno-format-truncation
COMPILE_OPT += -Wno-restrict
COMPILE_OPT += -Wno-format
COMPILE_OPT += -Wno-cpp #TODO temp
COMPILE_OPT += -Wno-discarded-qualifiers
COMPILE_OPT += -Wmissing-prototypes
COMPILE_OPT += -Werror=traditional
COMPILE_OPT += -Werror=missing-prototypes
COMPILE_OPT += -fdce
COMPILE_OPT += -fdse
COMPILE_OPT += -fmessage-length=0
COMPILE_OPT += -fsigned-char
COMPILE_OPT += -fno-common
COMPILE_OPT += -fstack-usage
COMPILE_OPT += -fzero-initialized-in-bss
COMPILE_OPT += -finline-small-functions
COMPILE_OPT += -Wmissing-field-initializers
COMPILE_OPT += -Werror=missing-field-initializers
COMPILE_OPT += -Werror=unused-but-set-variable
COMPILE_OPT += -Werror=implicit-function-declaration
COMPILE_OPT += -Werror=unused-variable
COMPILE_OPT += -Wformat-overflow=1
# Generate dependency information
COMPILE_OPT += -MMD -MP -MF"$(@:.o=%d)"

ifeq ($(DEBUG), Y)
    COMPILE_OPT += -O0
    COMPILE_OPT += -g3 -ggdb -gdwarf-2
else
    COMPILE_OPT += -Os
endif

COMPILE_OPT += $(CSTANDARD)

```

```
COMPILE_OPT += $(MICROPROCESSOR)
COMPILE_OPT += $(INCDIR)
```

```
CFLAGS += $(OPT)
CFLAGS += $(COMPILE_OPT)
```

Подобно компилятору, ключи надо передавать и компоновщику. Вот типичный пучок опций для настройки компоновщика. Что значит каждая опция компоновщика можно посмотреть в доке The GNU linker.

Листинг 19.12: Опции компоновщика

```
LINKER_FLAGS += -Xlinker --gc-sections
LINKER_FLAGS += -Xlinker --print-memory-usage

ifeq ($(LIBC_NANO), Y)
    LINKER_FLAGS += --specs=nano.specs
endif

ifeq ($(LIBC_RDIMON), Y)
    LINKER_FLAGS += --specs=rdimon.specs
endif

ifeq ($(LIBC), Y)
    LIBS += -lc
endif

ifeq ($(MATH_LIB), Y)
    LIBS += -lm
endif

LIBDIR +=

LDFLAGS += -t
LDFLAGS += $(MICROPROCESSOR)
LDFLAGS += -T$(LDSCRIPT)
LDFLAGS += $(LIBDIR)
LDFLAGS += $(LIBS)
LDFLAGS += -Wl,--cref
LDFLAGS += -Wl,--gc-sections
LDFLAGS += -Wl,-Map=$(BUILD_DIR)/$(TARGET).map
LDFLAGS += $(LINKER_FLAGS)
```

В переменную окружения LDFLAGS через ключ -T передается скрипт компоновщику. Он сообщает компоновщику, как организовать код из входных объектов.

19.3 Итоги

Вот вы теперь обладаете минимальным джентельменским набором для полноценной работы с системой сборки GNU Make.

Помимо сборки сорцов make позволит вам делать много других полезностей: автоматически форматировать отступы в исходниках, выставлять предваритель-

ные конфиги, прогнать препроцессор, запустить кодогенератор, построить документацию и многое другое.

19.4 Гиперссылки

1. Инструкция к GNU make на русском
2. GNU make
3. Магия makefile на простых примерах
4. Перевод Makefile Mini HOWTO
5. Everything You Never Wanted To Know About Linker Script
6. Пример Makefile
7. Эффективное использование GNU Make
8. Обновление Прошивки из Make Скрипта
9. Настройка ToolChain(a) для Win10++C+Makefile+ARM Cortex-Mx+GDB
10. GNU Make может больше чем ты думаешь

Глава 20

Диспетчер Задач для Микроконтроллера

"Прошивка сама себя не напишет."

20.1 Постановка задачи

В программировании микроконтроллеров часто нужно написать простые тестировочные прошивки. При этом надо некоторые функции вызывать чаще, а некоторые реже. Вот например как тут:

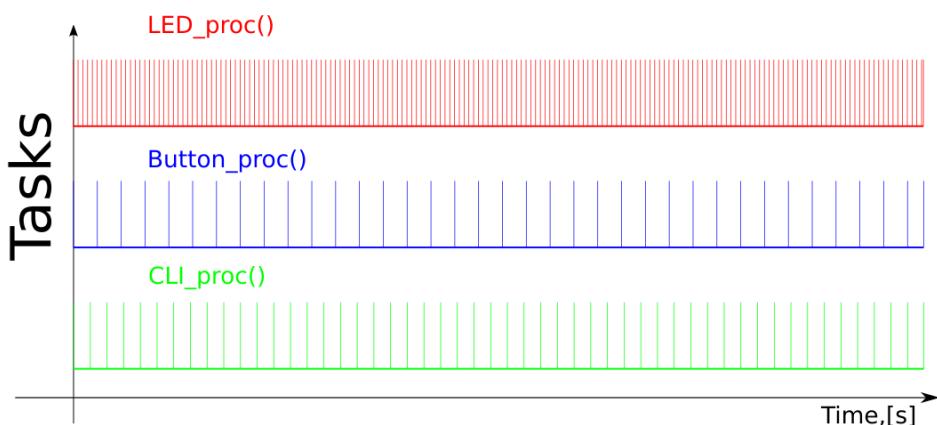


Рис. 20.1: Временная диаграмма отработки задач

Для этого конечно можно запустить FreeRTOS, однако тогда код не будет переносим на другие RTOS, например Zephyr RTOS, TI-RTOS, RTEMS, Keil RTX, Azure RTOS или SafeRTOS. Потом прошивку как часто приходится частично отлаживать на PC а там никакой RTOS в помине нет.

Поэтому надо держать наготове какой-нибудь простенький универсальный переносимый кооперативный NoRTOS планировщик с минимальной диагностикой и возможностью в run-time отключать какие-то отдельные задачи для отладки оставшихся.

Проще говоря нужен диспетчер задач для микроконтроллера.

20.2 Определимся с терминологией

Кооперативный планировщик - это такой способ управления задачами, при котором задачи сами вручную передают управления другим задачам.

Супер-цикл - тело оператора бесконечного цикла в NoRTOS прошивках. Обычно бесконечный цикл в прошивках можно найти по таким операторам как `for(;;)` или `while(1)`

Bare-Bone сборка - это сборка прошивки на основе API какой-нибудь RTOS, где только один поток и этот поток прокручивает супер-цикл с кооперативным планировщиком. Эта сборка нужна главным образом только для отладки RTOS: настройки стека, очередей и прочего.

Ядром любого планировщика является генератор или источник стабильно-го тактирования. Желательно с высокой разрешающей способностью. В идеале микросекундный таймер. Это может быть SysTick таймер внутри ядра ARM с пересчетом в микросекунды или отдельный аппаратный таймер общего назначения. Обычно в микроконтроллерах аппаратных таймеров от 3-х до 14-ти в зависимости от модели конкретного микроконтроллера. Также важно, чтобы таймер возрастал. Так проще и интуитивно понятнее писать код нам человекам, так как мы привыкли к тому, что время оно всегда непрерывно идет вперед, а не назад.

Для определенности будем считать, что все задачи имеют одинаковых прототипов

Листинг 20.1: Прототип задачи

```
bool task_proc(void);
```

Сначала надо определить типы данных.

Листинг 20.2: Тип данных задачи

```
#ifndef TASK_GENERAL_TYPES_H
#define TASK_GENERAL_TYPES_H

/*Mainly for NoRtos builds but also
 for so-called RTOS Bare-Bone build*/
#include <stdbool.h>

#include "task_const.h"
#include "limiter.h"

typedef struct {
    uint64_t period_us;
    bool init;
    Limiter_t limiter;
    const char* const name;
} TaskConfig_t;

#endif
```

Ключевым компонентом планировщика, его ядром является программный компонент называемый Limiter. Это такой программный компонент, который не поз-

волит вызывать функцию function чаще чем установлено в конфиге. Например вызывать функцию не чаще чем раз в секунду или не чаще чем раз в 10 ms.

Листинг 20.3: Тип данных для Limiter

```
#ifndef LIMITER_TYPES_H
#define LIMITER_TYPES_H

#include <stdbool.h>
#include <stdint.h>

#include "data_types.h"

typedef bool (*TaskFunc_t)(void);

typedef struct {
    bool init;
    bool on_off;
    uint32_t call_cnt;
    uint64_t start_time_next_us;
    uint64_t run_time_total_us;
    uint64_t start_time_prev_us;
    U64Value_t duration_us;
    U64Value_t start_period_us;
    TaskFunc_t function;
} Limiter_t;

#endif
```

Как вариант, помимо установки частоты исполнения задачи можно ещё добавить возможность управлять фазой вызова задачи.

Бот API планировщика. Механизм очень прост. Limiter измеряет время с момента подачи питания up_time_us, смотрит на расписание следующего запуска start_time_next_us и, если текущее время (up_time_us) больше времени запуска, назначает следующее время запуска и запускает задачу (limiter_task_frame).

Листинг 20.4: Алгоритм для Limiter

```
bool inline limiter(Limiter_t* const Node,
                     uint32_t period_us,
                     uint64_t up_time_us) {
    bool res = false;
    if (Node->on_off) {
        if (Node->start_time_next_us < up_time_us) {
            Node->start_time_next_us = up_time_us + period_us;
            res = limiter_task_frame(Node);
        }
        if (up_time_us < Node->start_time_prev_us) {
            LOG_DEBUG(LIMITER, "UpTimeOverflow %llu", up_time_us);
            Node->start_time_next_us = up_time_us + period_us;
        }
        Node->start_time_prev_us = up_time_us;
    }
    return res;
```

```
}
```

Limiter также ведёт аналитику. Измеряет время старта и окончания задачи, вычисляет продолжительность исполнения задачи (duration), вычисляет минимум (run_time.min) и максимум (duration.max), суммирует общее время, которое данная задача исполнялась на процессоре (run_time_total).

Листинг 20.5: Алгоритм для Limiter

```
static inline bool limiter_task_frame(Limiter_t* const Node) {
    bool res = false;
    if (Node) {
        uint64_t start_us = 0;
        uint64_t stop_us = 0;
        uint64_t duration_us = 0;
        uint64_t period_us = 0;

        start_us = limiter_get_time_us();

        if (Node->start_time_prev_us < start_us) {
            period_us = start_us - Node->start_time_prev_us;
            res = true;
        } else {
            period_us = 0;
            res = false;
        }

        Node->start_time_prev_us = start_us;
        if (res) {
            data_u64_update(&Node->start_period_us, period_us);
        }

        res = is_flash_addr((uint32_t)Node->function);
        if (res) {
            Node->call_cnt++;
            res = Node->function();
        } else {
            res = false;
        }

        stop_us = limiter_get_time_us();

        if (start_us < stop_us) {
            duration_us = stop_us - start_us;
            data_u64_update(&Node->duration_us, duration_us);
            Node->run_time_total_us += duration_us;
            res = true;
        } else {
            duration_us = 0;
            res = false;
        }
    }
    return res;
}
```

Стоит заметить, что перед непосредственным запуском конкретной задачи

Limiter может проверить, что указатель на функцию в самом деле принадлежит NOR-Flash памяти микроконтроллера. В основном супер цикле достаточно только перечислить те задачи, которые будут исполняться. Вот функция одной итерации супер-цикла.

Листинг 20.6: Одна итерация суперцикла

```

bool inline tasks_proc(uint64_t loop_start_time_us) {
    bool res = false;
    uint32_t cnt = task_get_cnt();
    uint32_t t = 0;
    for (t=0; t<cnt; t++) {
        if (TaskInstance[t].limiter.on_off) {
            res = limiter(&TaskInstance[t].limiter,
                           TaskInstance[t].period_us,
                           loop_start_time_us);
        }
    }
    return res;
}

bool super_cycle_iteration(void) {
    bool res = false;
    if (SuperCycle.init) {
        res = true;
        SuperCycle.spin_cnt++;
        SuperCycle.run = true;
        SuperCycle.start_time_us = time_get_us();
        if (SuperCycle.prev_start_time_us <
            SuperCycle.start_time_us) {
            SuperCycle.error++;
        }
        SuperCycle.duration_us.cur =
            (uint32_t)(SuperCycle.start_time_us -
                      SuperCycle.prev_start_time_us);
        SuperCycle.duration_us.min =
            (uint32_t)MIN(SuperCycle.duration_us.min,
                          SuperCycle.duration_us.cur);
        SuperCycle.duration_us.max =
            (uint32_t)MAX(SuperCycle.duration_us.max,
                          SuperCycle.duration_us.cur);
        super_cycle_check_continuity(&SuperCycle,
                                     loop_start_time_us);
        tasks_proc(SuperCycle.start_time_us);
        SuperCycle.prev_start_time_us = SuperCycle.start_time_us;
    }

    return res;
}

```

Бот код запуска супер цикла. Из функции `super_cycle_start()` и будет исполняться вся прошивка, за исключением вызова обработчиков прерываний ISR.

Листинг 20.7: запуск супер цикла

```
_Noreturn void super_cycle_start(void) {
```

```

LOG_INFO(SUPER_CYCLE, "Start");
super_cycle_init();
SuperCycle.start_time_ms = time_get_ms();
LOG_INFO(SUPER_CYCLE,
        "Started ,UpTime:%u ms",
        SuperCycle.start_time_ms);
for (;;) {
    super_cycle_iteration();
}
}

```

Такая сформировалась зависимость между программными компонентами данного планировщика. Для пуска планировщика нужен модуль flash, limiter, time и timer или systick.

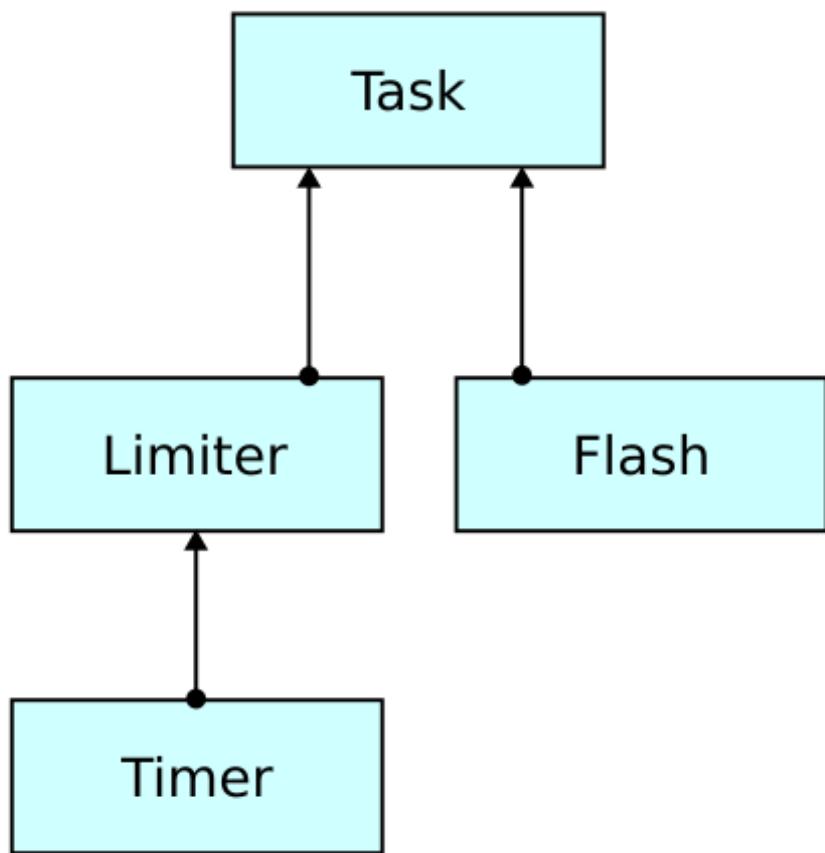


Рис. 20.2: Зависимости у компонента планеровщик

20.3 Отладка планировщика

Очевидно, что надо как-то наблюдать за работой планировщика. Подобно тому как в Windows мы открываем диспетчер задач. Для этого планировщик и были разработан, чтобы снимать метрики. Включать или останавливать задачи. Для этого можно воспользоваться интерфейсом командной строки CLI поверх UART.

В данном скриншоте можно заметить, что больше всего процессорного времени потребляет задача DASHBOARD (приборная панель). Тут же видно, что были

такие итерации супер цикла, что задача DASHBOARD непрерывно исполнялась аж 0.33 сек!

```

1:11-->tdr
task cnt 12
max loop duration 0 us
total run time 0 us
up_time 75059 ms
total_time 0 us
total_time 0 ns

+-----+
| Num | id | TaskName | On | Calls | Calls/s | CPU[%] | Rmin | Recur | Rmax |
+-----+
| 0 | 0 | BOOT | On | 37 | 0 | 0.137 | 57.0 ms | 57.0 ns | 57.0 ms |
| 1 | 1 | CLI | On | 5343 | 71 | 0.315 | 1.0 ms | 2.0 ns | 102.0 ms |
| 2 | 2 | FLASH_FS | On | 2 | 0 | 0.262 | 54.0 ms | 55.0 ns | 55.0 ms |
| 3 | 3 | HEAL_MON | On | 19 | 0 | 0.002 | 1.0 ms | 1.0 ns | 1.0 ms |
| 4 | 4 | PARM | On | 0 | 0 | 0.000 | 0.000 | 0.000 | 0.000 |
| 5 | 5 | BUTTON | On | 391 | 5 | 0.022 | 1.0 ms | 1.0 ns | 1.0 ms |
| 6 | 6 | DASHBOARD | On | 146 | 1 | 65.110 | 4.0 ms | 112.0 ns | 330.0 ms |
| 7 | 7 | DECADRIVER | On | 15835 | 210 | 0.269 | 1.0 ms | 2.0 ns | 3.0 ms |
| 8 | 8 | DS_TMR | On | 15834 | 210 | 1.569 | 1.0 ms | 1.0 ns | 18.0 ms |
| 9 | 9 | IWDG | On | 392 | 5 | 0.005 | 1.0 ms | 1.0 ns | 1.0 ms |
| 10 | 10 | LED_MONO | On | 15835 | 210 | 0.038 | 1.0 ms | 1.0 ns | 1.0 ms |
| 11 | 11 | SSD1306 | On | 146 | 1 | 32.271 | 92.0 ms | 92.0 ns | 93.0 ms |

75.288 :I [Scheduler] MaxTaskContinuousRunTime
1:15-->
1:16-->

```

Рис. 20.3: Диагностика по задачам

Можно измерить период с которым вызывалась каждая из задач и сопоставить с конфигом для каждой задачи. Тут видно, что в среднем реже всего вызывается задача FLASH_FS (менеджер файловой системы). Одновременно драйвер свето-диода LED_MONO отрабатывает с частотой (44 Hz). А чаще всего происходит опрос DecaDriver(a).

```

2:1-->
2:1-->tdp
123.133 :I [Scheduler] TaskCnt 11

+-----+
| init | Calls | Calls/s | Err | MinUs | CurUs | MaxUs | AvrUs |
+-----+
| On | 1078587 | 8759 | 0 | 4294967295 | 1000 | 0 | 112 |

123.281 :I [SuperLoop] Runtime: 121002800 us

+-----+
| id | TaskName | On | Calls | Calls/s | CPU[%] | Tset | Tmin | Tcur | Tmax | Tavg |
+-----+
| 0 | BOOT | On | 61 | 0 | 0.086 | 2.0 s | 2.0 s | 2.0 s | 2.0 s | |
| 1 | CLI | On | 2046 | 16 | 3.236 | 30.0 ms | 31.0 ms | 713.0 ms | 60.2 ms |
| 2 | FLASH_FS | On | 3 | 0 | 0.252 | 1.0 m | 1.0 m | 1.0 m | 41.0 s |
| 3 | HEAL_MON | On | 31 | 0 | 0.000 | 4.0 s | 4.0 s | 4.0 s | 4.0 s |
| 4 | BUTTON | On | 716 | 5 | 0.060 | 100.0 ms | 101.0 ms | 101.0 ms | 172.0 ms |
| 5 | DASHBOARD | On | 248 | 1 | 59.340 | 500.0 ms | 593.0 ms | 593.0 ms | 513.1 ms |
| 6 | DECADRIVER | On | 29373 | 238 | 0.211 | 1.0 ms | 2.0 ms | 2.0 ms | 4.2 ms |
| 7 | DS_TMR | On | 29372 | 238 | 1.749 | 1.0 ms | 2.0 ms | 2.0 ms | 4.2 ms |
| 8 | IWDG | On | 0 | 0 | 0.000 | 0 us | 307445734561.8 m | 0 us | 0 us | 0 us |
| 9 | LED_MONO | On | 5477 | 44 | 0.021 | 10.0 ms | 11.0 ms | 440.0 ms | 22.5 ms |
| 10 | SSD1306 | On | 248 | 1 | 35.045 | 500.0 ms | 501.0 ms | 871.0 ms | 513.1 ms |

123.446 :I [Scheduler] TotalSupLPRunTime: 121070000 us
123.454 :I [Scheduler] AllTasksDuration: 63009999 us, 100 %
123.464 :I [Scheduler] SchedulerDuration: 580600001 us
123.472 :I [Scheduler] SchedulerOverhead: 47.956 %
123.481 :I [Scheduler] Tmin - MinimumTaskExecutionPeriod
123.489 :I [Scheduler] Tavg - AverageTaskExecutionPeriod
123.498 :I [Scheduler] Tmax - MaximumTaskExecutionPeriod
2:3-->

```

Рис. 20.4: Диагностика по задачам

Тут заметно даже, что накладные расходы на данный планировщик составляют 47.9 процент по времени. Это лишь потому, что в обработчиках самих задач не происходит пока никакой обработки. Прошивка работает вхолостую. Не было прерываний и флаги не устанавливаются. Не приходят пакетов в интерфейсы. Ничего не происходит.

Также высокое процентное значение Scheduler Overhead это признак того, что периоды у всех задач большие и процессору нечего делать. А значит можно смело добавлять в супер цикл больше кооперативных задач или уменьшать периоды у нынешних задач.

Анализируя эти ценнейшие метрики данного импровизированного планировщика можно принимать решения по оптимизации кода всего проекта. Получился своеобразный Code Coverage.

20.4 Достоинства данного планировщика

1. Простота, очевидность, прозрачность, мало кода.
2. Можно вычислить процент загрузки процессора по каждой задаче.
3. Переносимость. Можно его прокручивать хоть на микроконтроллере, хоть на BareBone потоке в RTOS, хоть в консольном приложении на LapTop PC.
4. Приоритет задачи задается периодом её запуска. Чем ниже период, тем выше приоритет.
5. Легко масштабировать прошивку. Просто добавляем новые строчки в super цикл.
6. Можно весь этот планировщик вообще реализовать на функциях препроцессора. И тогда не будет накладных расходов на запуск функций. Однако так будет невозможно осуществлять пошаговую отладку программы.
7. Можно переназначать функции для узлов планировщика и таким образом перепрограммировать устройство далеко в Run-Time.

20.5 Недостатки данного планировщика

1. Если одна задача зависла, то считай, что зависли все остальные задачи.
2. Надо проектировать задачи так, чтобы они что-то делали за один прогон и не тратили много времени внутри себя. Например переключили состояние конечного автомата и вышли. Совсем не здорово, если какая-то задача начнет расшифровывать 150kByte KeePass файл внутри общего супер цикла или вычислять обратную матрицу 100x100. У Вас перестанет мигать Heart Beat LED, перестанет отвечать CLI и пользователь будет с полной уверенностью считать, что прошивка просто взяла и зависла! А на самом деле программа через 57 секунд снова воспряннет.
3. Требуются накладные расходы (в виде процессорного времени) для вычисления метрик за которыми следит Limiter. Но это не такая и большая проблема, так как отладочные метрики можно включать или исключать на стадии препроцессора ifdef(ами).

20.6 Итог

Вот и Вы умеете делать кооперативный планировщик. Супер цикл это не такая уж и плохая вещь. Его можно отлично использовать и в RTOS прошивках.

Если проводить аналогию, то данный диспетчер задач для микроконтроллера является эдаким игровым движком из GameDev.

Есть код которому точно нужен RTOS. Это BLE, LwIP стек, однако всё остальное: LED, Button, CLI может отлично работать в пределах супер цикла в отдельном BareBone потоке. Благодаря супер циклу Вы сэкономите на переключении контекста. Надеюсь, что этот текст поможет кому-нибудь писать прошивки и оценивать нагрузку на процессор.

20.7 Гиперссылки

1. Планировщик для микроконтроллера
2. Почти ОС реального времени: event-driven
3. Отладка многопоточных программ на базе FreeRTOS
4. Учебный Курс. Архитектура Программ
5. Учебный Курс. Архитектура Программ. Часть 4

20.8 Контрольные вопросы

1. В какую сторону в ARM Cortex-M4 считает Sys Tick таймер?
2. Как измерить загруженность процессора в NoRTOS прошивке?
3. Сколько тактов процессора нужно для вызова Си-функции на микропроцессоре ARM Cortex-M4?
4. Сколько тактов процессора нужно для вызова обработчика прерываний на микропроцессоре ARM Cortex-M4?

Глава 21

NVRAM для микроконтроллеров

"Любая вещь лучше, когда внутри её есть NVRAM."

21.1 Пролог

В разработке на микроконтроллерах хорошей практикой считается, когда на устройстве есть число-хранилище для запоминания чиселок между перебросом электропитания питания. Все это называют по-своему: NVRAM, StoreFS, FlashFS, Энергонезависимая Key-Value Map(а), HashMap(ка), NVS, memory-manager, накопитель и прочее и прочее.

21.2 Nvram по-русски

Неоднократно я был свидетелем, как в России выкручиваются от неспособности запрограммировать полноценный NVRAM. Выглядит этот так. Схемотехники в плату закладывают три или 6 GPIO пинов и джамперами выставлять бинарный код на GPIO, чтобы при старте дать прошивке какую-то команду. Получается $1 \times 3 \times 6 = 18$ кубических миллиметров на бит.

NVRAM может пригодится для хранения конфигураций умных навороченных микросхем типа беспроводных трансиверов, драйверов шаговых двигателей, значение одометра, пароли, своего CAN-ID(шника), MAC/IP адреса, SSID, пароля от маршрутизатора, серийные номера и т.п. Наличие NVRAM позволяет на порядок уменьшить количество сборок в репозитории, так как устройства всегда можно до программировать уже в Run-Time(е). Буквально открыв UART Shell можно прямо руками в TeraTerm/Putty прописать во NVRAM конфиги и перезагрузить (reset(нуть)) гаджет, чтобы новые настройки применились в инициализации при загрузке до запуска суперцикла. Easy! И не надо варить отдельные прошивки с какими-то специфическими настройками, когда есть NVRAM. И не надо джамперами на GPIO колхозить установку каких-то адресов. NVRAM добавляет в программу положительную динамику.

Прошивка без NVRAM это как LapTop компьютер без жёсткого диска. Нужен ли вам такой?...

21.3 Достоинства on-chip NorFlash

Почему именно on-chip NorFlash?

1. Это дешево. Как правило после накатывания прошивки всегда остается ещё пара пустых страниц(секторов) Flash(a). За неё "уже заплачено" при покупке микроконтроллера. Нет нужды ставить еще отдельный off-chip SPI-NorFlash чипы и тем самым увеличивать габариты, стоимость, расширять логистику, уменьшать надежность, увеличивать сложность устройства. NVRAM предусматривают сами производители микроконтроллеров. Например в STM32 первые 4 сектора вообще по 16kByte, остальные больше по 128kByte. Это сделано специально, с расчетом на то, чтобы запускать на маленьких секторах по 16kByte NVRAM(сы) с плавно конфигурируемым размером.

У меня в проектах на STM32 NVRAM обитает во втором и третьем секторах on-chip Nor Flash(a).

2. On-chip FS безопаснее чем off-chip FS, так как нет возможности подцепиться к SPI или SDIO проводам на PCB и тем самым считать передаваемые данные обыкновенным китайским логическим анализатором за 500 рублей.

Однако у Nor-Flash памяти есть одна проблема. Стирать её (заполнять 0xFF(ками)) можно только страницами по несколько килобайт (обычно это 4kByte, 8kByte, 16kByte, 64kByte, 128kByte). Одновременно запись заключается в том, что можно только биты 1 сбросить в 0. Иногда даже нельзя до сбросить уже сброшенные биты, например, в байте 0x55. Это как писать шариковой ручкой по бумаге. Написанное не сотрешь ластиком. Есть смысл исписать весь лист, а затем вырвать его и выбросить.

Вот так обычно выглядит API для работы с On-Chip Nor-Flash памятью.

Листинг 21.1: API Flash

```
#ifndef FLASH_DRV_H
#define FLASH_DRV_H

#include <stdbool.h>
#include <stdint.h>

#include "flash_types.h"

#ifdef HAS_FLASH_WRITE
bool flash_wr(uint32_t addr,
              uint8_t* array, uint32_t array_len);
bool flash_wr4(uint32_t flash_addr,
               uint32_t* wr_array, uint32_t byte_size);
bool flash_erase(uint32_t addr, uint32_t len);
bool flash_erase_pages(uint8_t page_start,
                      uint8_t page_end);
#endif

bool flash_read(uint32_t in_flash_addr,
                uint8_t* rx_array, uint32_t array_len);
```

```

bool flash_init(void);
bool flash_scan(uint8_t* base, uint32_t size,
                float* usage pec, uint32_t* spare, uint32_t* busy);
bool is_erased(uint32_t addr, uint32_t size);
bool is_flash_spare(uint32_t flash_addr, uint32_t size);
bool is_flash_addr(uint32_t flash_addr);

#endif /* FLASH_DRV_H */

```

Нам же, людям, удобнее просто записывать данные по ключу. Это как в театре. Даешь номерок, получаешь свою куртку. Или как в телефонной записной книжке. Даешь имя, получаешь номер телефона.

Нужен определенный уровень абстракции, который бы давал такой API и делал бы всю остальную работу под капотом с массивами сырой Flash памяти. Даешь число и массив, Массив записывается. Через неделю даешь число, и получаешь массив.

Это как писать карандашом. Можно написать и можно стереть ластиком и написать что-н другое. Удобно? Очень.

21.4 Типичные требования к NVRAM

Попробую перечислить самые базовые требования для таких embedded on-chip файловых систем NVRAM.

1. Рациональное использование Nor-Flash памяти (endurance optimization)
2. Защита данных от внезапного пропадания питания (power off tolerance).
3. Простота реализации чтобы нечemu было ломаться.
4. Файл должен быстро находится по имени
5. Файл должен быстро записываться
6. Файл должен быстро обновляться
7. Файл должен быстро стираться
8. Компактность кода, который реализует этот алгоритм NVRAM. Чем меньше кода, тем больше файлов.

21.5 Терминология

Определимся с терминологией. Что такое File?

File- это именованный бинарный массив байтов в памяти. В качестве памяти может выступать RAM, ROM (Flash), FRAM, EEPROM, Flash-NAND в SD картах.

Что значит именованный массив? Это значит, что к этим данным можно обращаться по значению. В самом простом случае имя файла - это натуральное 16-битное число. Так проще. Так как это массив, то рядом с данными также надо хранить и длину этого массива. Вот так может выглядеть примерный заголовок записи в NVRAM.

Листинг 21.2: заголовок записи в NVRAM

```
struct xNVRAMFileHeader_t {
    uint16_t id;
    uint16_t nid; /* bit inverted id */
    uint16_t length;
    uint8_t crc8; /*only for payload*/
} __attribute__((packed)); /*to save flash memory*/
typedef struct xNVRAMFileHeader_t NVRAMFileHeader_t;
```

Эта структура по сути является преамбулой к данным. Во Flash переменные будут в little-endian формате.

21.6 Основная идея механизма NVRAM

Если записывать файлы с одинаковыми ID, то они будут записываться последовательно. Этим достигается защита данных от пропадания питания. Всегда можно взять предыдущий файл. Каждый файл оснащен 8ми битной контрольной суммой CRC. Это позволит выявлять только реальные файлы, а не просто случайные числа в памяти. Чтобы каждый раз не считать контрольную сумму для каждого отступа, есть преамбула из ID(шников). Это FileID и его инвертированная копия. Зачем нужен инвертированный ID? Это для того чтобы ускорить поиск нужного файла. Дело в том, что процедура вычисления CRC это весьма продолжительная процедура. Было бы расточительно рассчитывать CRC для каждого отступа, чтобы понять файл тут или не файл. Поэтому алгоритм рассчитывает CRC только по тем отступам, где прописана валидная преамбула.

И вот заполнилась страница флеш памяти. Что делать? Надо отчистить вторую страницу и перекопировать в неё попарно отличные самые свежие файлы.

При копировании страницы перебрасываются только самые последние версии файлов. Старые остаются и ждут своего удаления вместе со всей страницей. При следующем переключении их отчистят вместе со всей страницей NorFlash.

Помимо функций чтения и записи файлов нужна еще вспомогательная функция, которая будет как раз следить за самой файловой системой. Как только страница "B" переполнится, то background процедура должна выполнить процедуру toggle flash page, то есть отчистить страницу "A". Взять самые свежие файлы из страницы "B" и перекопировать их в страницу "A".

Вот так может выглядеть API для микроконтроллерной файловой системы

Листинг 21.3: API для микроконтроллерной файловой системы

```
#ifndef NOR_FLASH_H
#define NOR_FLASH_H

#include <stdbool.h>
#include <stdint.h>

#include "flash_fs_config.h"
#include "flash_fs_types.h"
```

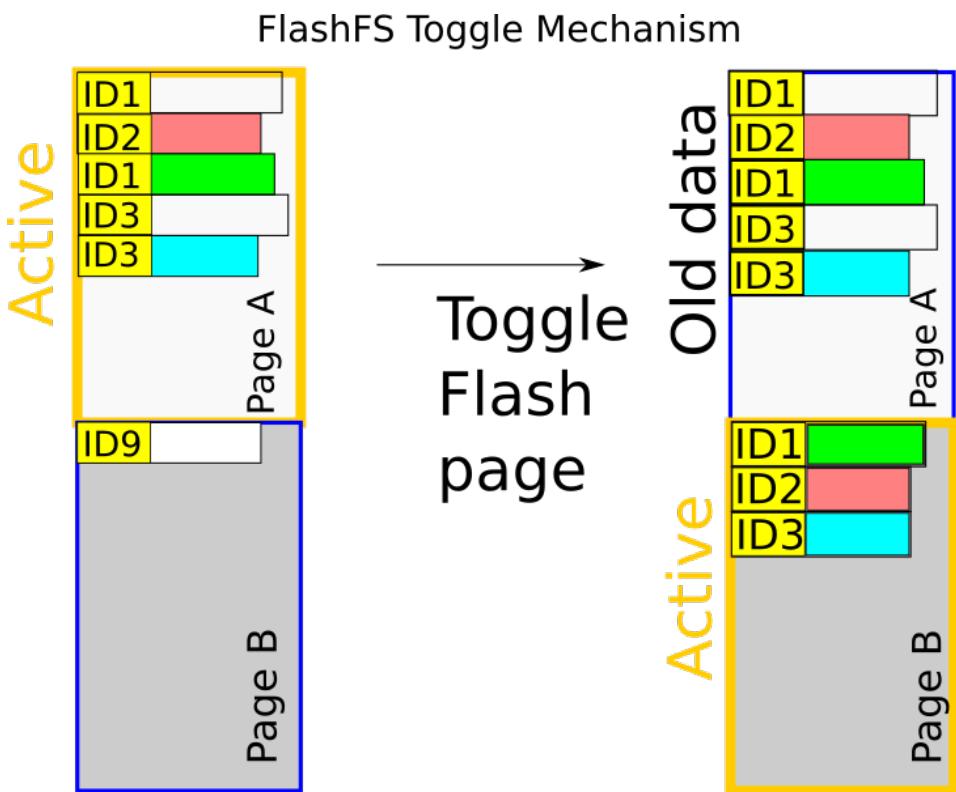


Рис. 21.1: Схема переключения страниц

dec	hex	kByte	kByte	
sector	Address	size	offset	содержимое
0	0x0800 0000	16	16	первичный загрузчик
1	0x0800 4000	16	32	первичный загрузчик
2	0x0800 8000	16	48	FlashFS Page 1
3	0x0800 C000	16	64	FlashFS Page 2
4	0x0801 0000	64	128	Generic прошивка
5	0x0802 0000	128	256	Generic прошивка
6	0x0804 0000	128	384	Generic прошивка
7	0x0806 0000	128	512	Generic прошивка
8	0x0808 0000	128	640	Generic прошивка
9	0x080A 0000	128	768	Generic прошивка
10	0x080C 0000	128	896	Generic прошивка
11	0x080E 0000	128	1024	Вторичный загрузчик

Рис. 21.2: Разметка памяти STM32

```
#ifdef HAS_FLASH_FS_WRITE
bool flash_fs_format(void);
bool flash_fs_erase(void);
bool flash_fs_invalidate(uint16_t data_id);
bool flash_fs_set(uint16_t data_id,
                  uint8_t* new_file,
                  uint16_t new_file_len);
bool flash_fs_maintain(void);
bool flash_fs_turn_page(void);
```

```

#endif

bool flash_fs_is_active(uint8_t page_num);
bool flash_fs_init(void);
bool flash_fs_proc(void);
bool flash_fs_get(uint16_t data_id,
                  uint8_t* value,
                  uint16_t max_value_len,
                  uint16_t* value_len);
bool flash_fs_get_active_page(uint32_t* page_start,
                             uint32_t* page_len);
bool flash_fs_get_address(uint16_t data_id,
                         uint8_t** value_address,
                         uint16_t* value_len);
bool is_flash_fs_addr(uint32_t addr);
uint32_t flash_fs_get_page_size(uint8_t page_num);
uint32_t flash_fs_get_page_base_addr(uint8_t page_num);
uint32_t flash_fs_cnt_files(uint32_t start_page_addr,
                           uint32_t page_len,
                           uint32_t* spare_cnt);
uint32_t flash_fs_get_remaining_space(void);
uint8_t addr2page_num(uint32_t page_start);

#endif /* MEMORY_MANAGER_NOR_FLASH_H */

```

Зависимости программных компонентов для NVRAM можно показать так

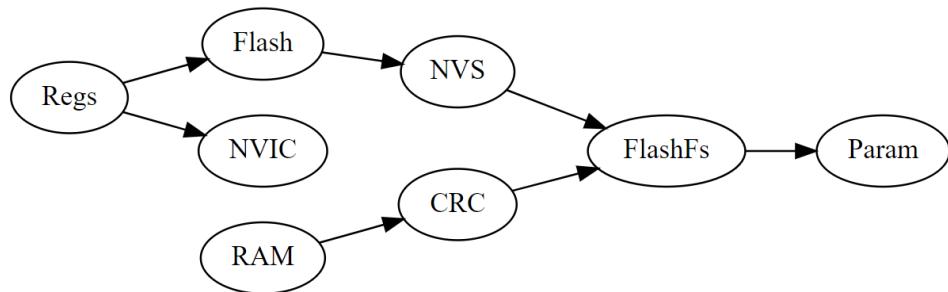


Рис. 21.3: Зависимости программных компонентов для NVRAM

Или так

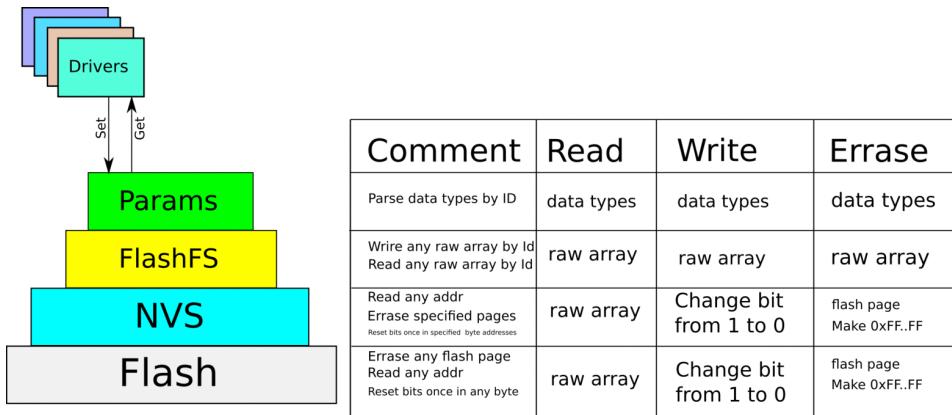


Рис. 21.4: Иерархия программных компонентов

Вот так может выглядеть диагностика файловой системы . Это список файлов в файловой системе, их адреса, размер, данные внутри, контрольная сумма, ID(шник)

4051.7-->ffc						
4052.806:I [FLASH_FS] Page 0x08120000...0x08140000						
num	id	len	crc	addr	ldata	
0	11	4	0xb8	0x81200d5	0x00000208	
1	12	1	0xbd	0x81205a6	0x00	
2	60	4	0xa7	0x8120641	0x01000000	
3	13	1	0xea	0x8120689	0x08	
4	1	2	0xa4	0x8120691	0x9700	
4052.967:I [FLASH_FS] Page 0x08140000...0x08160000						
num	id	len	crc	addr	ldata	

Рис. 21.5: Диагностика Flash FS

21.7 Добавление IDшников и интерпретатора (PARAM)

С файлами разобрались. Но читать сырье данные в памяти тоже как-то не очень-то удобно. Для человека это просто последовательность нимблов (hex разрядов). Надо как-то интерпретировать эти данные в реальные физические величины и типы данных. Этим займется программный компонент PARAM.

Поверх Flash Fs должен работать еще один уровень абстракции. Я его называю параметры (Param). Работает он так. Даешь ID файла и данные (hex массив с длинной), получаешь тип данных и его значение.

Вот так могут выглядеть прототипы функций для компонента PARAM:

Листинг 21.4: прототипы функций для компонента PARAM

```
#ifndef PARAM_DRV_H
#define PARAM_DRV_H

#include <stdbool.h>
#include <stdint.h>

#include "param_types.h"

#ifndef HAS_PARAM
#error "+HAS_PARAM"
#endif /*HAS_PARAM*/

bool param_init(void);
bool param_proc(void);
```

```

#ifndef HAS_PARAM_SET
bool param_set(Id_t id, uint8_t* in_data);
#endif /*HAS_PARAM_SET*/

bool param_get(Id_t id, uint8_t* out_data);
ParamType_t param_get_type(Id_t id);
uint16_t param_get_real_len(Id_t id) ;
uint16_t param_get_len(Id_t param_id);
uint16_t param_get_type_len(ParamType_t type_id);
uint32_t param_get_cnt(void);

#endif /* PARAM_DRV_H */

```

Вот теперь с этим можно работать. Переменные понятные, значения интерпретируются. Успех.

No	id	group	VarName	len	val	name
1	70	IHDG	iHdgBootOn	1	0	Undef ID
2	72	IHDG	iHdgGenerOn	1	0	Undef ID
3	71	IHDG	IHDGOnPeriodMs	4	1000	Undef ID
4	79	KEEPASS	FileName	7	db.kdbx	db.kdbx
5	75	KEEPASS	TransformKey	96		Undef ID
6	74	SDIO	ClockDiv	1	64	Undef ID
7	1	BOOT	ReBootCnt	2	106	Undef ID
8	60	?	MaxUpTime	4	9364260	9364.3s=156.1min=2.60 h
9	25	?	SerialNum	4	0	Undef ID
10	12	BOOT	BootCmd	1	0	launch
11	13	BOOT	BootCnt	1	0	Undef ID

151.174 :I [SYS] CmdOk!

Рис. 21.6: Диагностика NVRAM

21.8 Итоги

Как видите программы для микроконтроллеров строятся из уровней абстракции, которые работают один поверх другого.

Теперь вы представляете как делать энергонезависимую файловую систему для хранения всякого разного (настроек, прошивок, калибровочных параметров). Не обязательно делать файловую систему именно на on-chip Nor Flash. Можно и на off-chip Nor Flash или EEPROM.

Добавляйте в свои прошивки файловые системы. В этом нет ничего сложного. Любая вещь лучше, когда внутри её есть NVRAM.

21.9 Гиперссылки

1. NVRAM для микроконтроллеров
2. NVRAM Поверх off-chip SPI-NOR Flash

3. Открытый проект файловой системы для внутренней памяти STM32H
4. ESP32 и файловая система SPIFFS
5. LittleFS – файловая система для ARM MK
6. MicroFAT: A File System for Microcontrollers
7. Устройство NVRAM в UEFI-совместимых прошивках, часть первая
8. Как избежать износа EEPROM

Глава 22

Конечные автоматы на службе программирования микроконтроллеров

"Лучший алгоритм - это таблица"

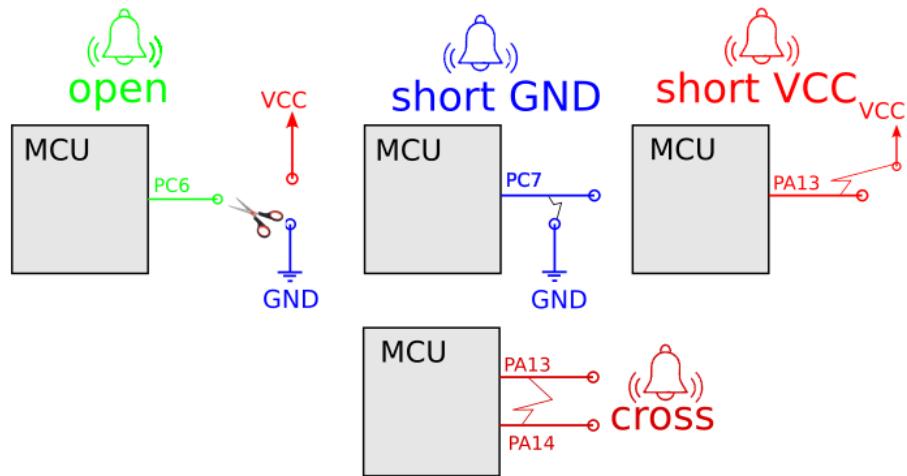


Рис. 22.1: Разновидности дефектов в пайке электронных плат

22.1 Пролог

Конечный автомат это культовый классический шаблон проектирования программных и аппаратных алгоритмов. Материалов по этой теме океан.

В этот тексте я покажу как работать с конечным автоматом на реальном примере. Я синтезирую программный компонент cross-detect для проверки качества пайки. За одно вы сможете сами использовать cross-detect в своих проектах.

Как водится в программировании, разработка любого программного компонента сводится к тому, что надо реализовать специфический конечный автомат

(FSM) для этой задачи. Любой нормальный программный компонент как правило построен на основе конечного автомата внутри. Конечный автомат это золотой шаблон в программировании. Конечные автоматы хороши тем, что процесс разработки FSM лет таксто поставлен на рельсы и состоит из девяти чисто формальных фаз:

22.2 Фазы проектирования конечного автомата

1. Написать модульные тесты для конечного автомата
2. Определить входы конечного автомата.
3. Перечислить все возможные состояния конечного автомата.
4. Определить легальные действия, которые будет производить конечный автомат
5. Построить таблицу переходов (State Table)
6. Построить таблицу выходов (Action Table)
7. Нарисовать граф конечного автомата
8. Написать код
9. Отладить код

22.3 Терминология конечных автоматов

1. граф - это множество вершин и ребер (палочки и кружочки).
2. ориентированный граф — граф у которого ребра имеют направление
3. конечный автомат - ориентированный граф. Вершины - это состояния, ребра - события
4. конечный автомат Мили- это конечный автомат у которого действия происходят в момент переходов
5. конечный автомат Мура - это конечный автомат у которого действия происходят в состоянии
6. токен - текстовая строчка без пробелов. Служит ссылкой на более длинное предложение

Конечные автоматы - это абстрактное понятие. Поэтому лучший способ их понять - это сделать что-нибудь полезное на основе КА.

22.4 Демонстрационная задача на FSM

"Электроника - это наука о контактах."
(Профессор в университете)

Представьте ситуацию. Вам с производства принесли 7 экземпляров самой первой ревизии PCB, прототипа нового сложного электронного продукта прямо с поверхностного монтажа.

Платы ещё ни разу не включали.

Как проверить, что в электронных платах отсутствует брак монтажа: короткие замыкания на GND или VCC, короткие замыкания соседних пинов друг на друга и прочие аппаратные ошибки?

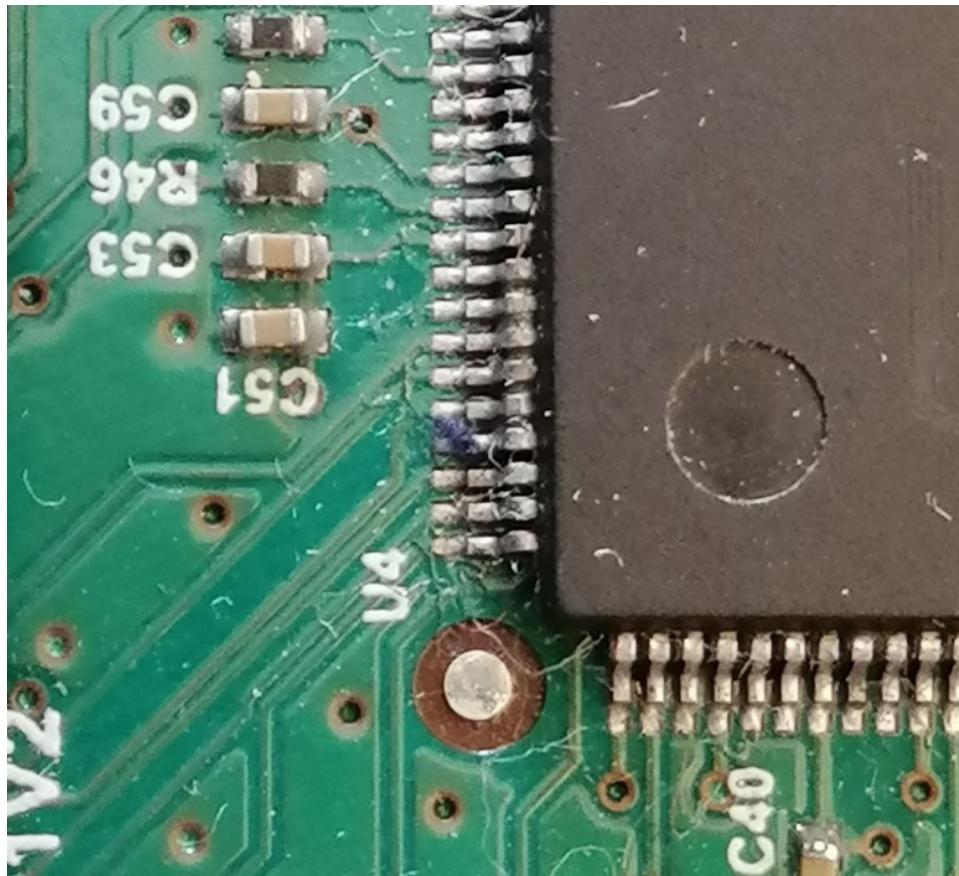


Рис. 22.2: Пример короткого замыкание соседних пинов

По-хорошему надо тестировать PCB на короткие замыкания и разрыв проводников еще до того как на печатную плату будут припаиваться компоненты. А для этого нужны тестировочные стенды (test jig) под каждую версию PCB. Разработку test jig могут позволить себе далеко не все организации.

Как можно в более кустарных условиях протестировать электронную плату на предмет качества пайки?

22.5 Основная идея

Подойдем к решению постановки задачи издалека. Как вы думаете зачем в микроконтроллерах есть такая аппаратная функция как подтяжки напряжения к GND, VCC или вовсе отключенные подтяжки? Иногда это нужно для работы интерфейсов например I2C, 1-Wire, UART Rx, кнопок, но есть и более подходящая работа для подтяжек пинов.

Самое главное достоинство подтяжек напряжения в том, что установкой подтяжки невозможно, что бы то ни было сжечь на электронной плате. Подтяжка всегда подключает резистор очень высокого сопротивления 10k Ω -40k Ω , поэтому и токи в электронной цепи протекают незначительные (80 μ A ... 300 μ A).

Суть этого текста в том, что если правильным образом манипулировать подтяжками напряжений на пинах, то можно распознать такие высокоуровневые события как замыкание проводов друг на друга, или замыкания конкретного провода на GND или VCC.

В этом тексте я представил Cross-Detect алгоритм, который помимо замыканий на землю и питание может распознать ещё и короткое замыкание не только между соседними пинами, но и между любыми проводниками на электронной плате (PCB)! Это просто мечта любого человека, который занимается bring-up(ом) электронных плат с производства.

Основная идея алгоритма cross-detect состоит в том, чтобы взять два различных микроконтроллерных пина и рассматривать их как плечи Н-моста. Вот так выглядит электрическая цепь Н-моста

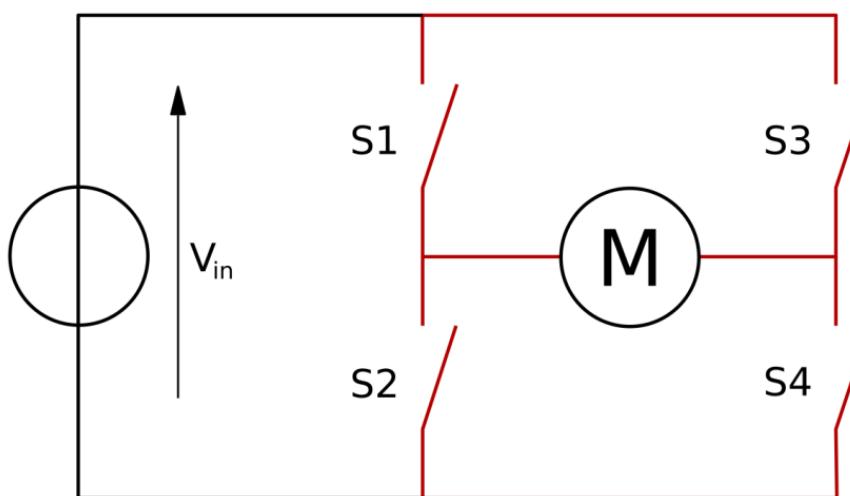


Рис. 22.3: Схема Н моста

Здесь же в контроле качества пайки PCB ситуация наоборот. Оторванная нагрузка это как раз благоприятный вариант. Тут будет всё как в том тексте про регистрацию обрыва нагрузки с той лишь разницей, что вместо плечей моста будут выступать просто два пина микроконтроллера и появилась еще и подтяжка к земле.

Я описал полный путь от идеи до реализации.

22.6 Какие могут быть проблемы в Н-мосте?

Да целая куча. Могут быть так, что левое плечо замкнуто на землю, а правое на VCC. Может быть так, что пины вдруг замкнулись друг на друга там, где этого не должно происходить. Или оба пина замкнуты на VCC.

#	left shoulder	right shoulder	shoulder	solution	fault	fine
1	1		Left	SHORT_GND	1	--
2	1		Left	SHORT_VCC	1	--
3	1		Left	OPEN	--	1
4		1	Right	SHORT_GND	1	--
5		1	Right	SHORT_VCC	1	--
6		1	Right	OPEN	--	1
7	1	1		CROSS	1	--

Рис. 22.4: Список типичных ошибок H-моста

22.7 Определить входы конечного автомата

С входами тут всё очень просто. Вход только один. Это событие переполнения таймера окончания переходного процесса. Таймер может быть как аппаратный, так и программный. Можно заметить, что при установке подтяжки напряжения реальная длительность переходного процесса редко превышает 3 ms. Но я для надежности поставил 10...100ms.

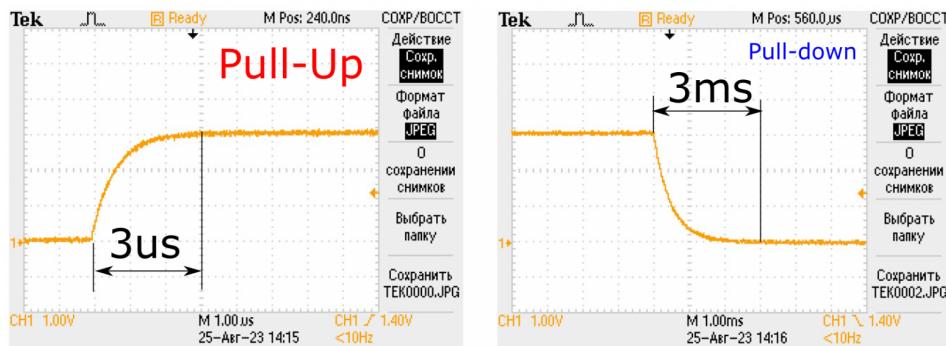


Рис. 22.5: Переходной процесс установки подтяжки напряжения

22.8 Перечислить все возможные состояния конечного автомата

Тут чистая комбинаторика. Сколько способов установить три различные подтяжки на 2 пина? Согласно правилу произведения, $3^2=9$ способов. Значит у нас будет девять состояний.

order	Left pin	Right pin	Token
1	pull air	pull air	LARA
2	pull up	pull air	LURA
3	pull down	pull air	LDRA
4	pull down	pull down	LDRD
5	pull down	pull up	LDRU
6	pull air	pull up	LARU
7	pull up	pull up	LURU
8	pull up	pull down	LURD
9	pull air	pull down	LARD

Рис. 22.6: Список состояний конечного автомата

22.9 Определить действия для данного конечного автомата

У каждого конечного автомата есть действия, которые он может делать (легальные действия) и действия, которые он не может делать. Конечному автоматау контроля пайки достаточно всего-навсего вот этих девяти действий.

#	shoulder	Action	left shoulder	right shoulder
1	left	set pull up	1	--
2	left	set pull down	1	--
3	left	set pull air	1	--
4	left	read GPIO level	1	--
5	right	set pull up	--	1
6	right	set pull down	--	1
7	right	set pull air	--	1
8	right	read GPIO level	--	1
9	--	calc solution	--	--

Рис. 22.7: Список действий конечного автомата

Это те действия которые происходят как бы за кулисами. Пользователю конечного автомата совершенно не надо знать, что автомат что-то там делает.

22.10 Построить таблицу переходов

Есть одна особенность. Переходить из одного состояния в другое надо так, чтобы каждый раз устанавливалась только одна подтяжка. Это нужно по двум причинам:

1. Для уменьшения энергопотребления микроконтроллера
2. Для уменьшения времени переходного процесса. Один переходной процесс всегда закончится быстрее, чем два параллельных переходных процесса.

В двоичной системе счисления такие коды, где меняется только один бит - называются кодами Грея. У нас же тут, по сути, тоже коды Грея, только в третичной системе счисления.

State		Input	TimeOut	
		LARA	LURA	
		LURA	LDRA	
		LDRA	LDRD	
		LDRD	LDRU	
		LDRU	LARU	
		LARU	LURU	
		LURU	LURD	
		LURD	LARD	
		LARD	LARA	

Рис. 22.8: Таблица переходов конечного автомата

22.11 Построить таблицу выходов (Action Table)

В общем, на каждом переходе конечный автомат делает одно и тоже: перезапускает таймер. Однако на последнем переходе автомат увеличивает счетчик итераций, вычисляет решение, сохраняет решение в отчет и, если надо, выбирает следующую пару пинов для анализа коротких замыканий.

State	Input		TimeOut
	1	2	
LARA			reset timer start timer
LURA			reset timer start timer
LDRA			reset timer start timer
LDRD			reset timer start timer
LDRU			reset timer start timer
LARU			reset timer start timer
LURU			reset timer start timer
LURD			reset timer start timer
LARD			reset timer start timer calc solution

Рис. 22.9: Таблица действий конечного автомата

22.12 Нарисовать граф конечного автомата

Граф можно нарисовать вручную в программе inkscape (как я и сделал) или сгенерировать на языке генерации авто-документации Graphviz. Тут уж кому как удобнее. Вот такой получился простенький граф для конечного автомата проверки качества пайки.

Тут всё как на ладони: и таблица переходов, и таблица выходов отражены в одной картинке. Как говорится:

"Картинка стоит тысячи слов"
 (A picture is worth a thousand words).
 (английская народная пословица)

Для чистоты эксперимента в каждый отдельный момент времени работает обработка только одной пары пинов. Конечный автомат cross-detect прокрутится N итераций, вычислит решение для пары пинов и занесет его в матрицу решений.

Конечный автомат Cross-Detect это конечный автомат Мура, так как тут состояние, как раз, напрямую состояние соответствует подтяжкам напряжения. Однако он же автомат Мили, так как все реальные действия по переключению подтяжек происходят в момент перехода между состояниями.

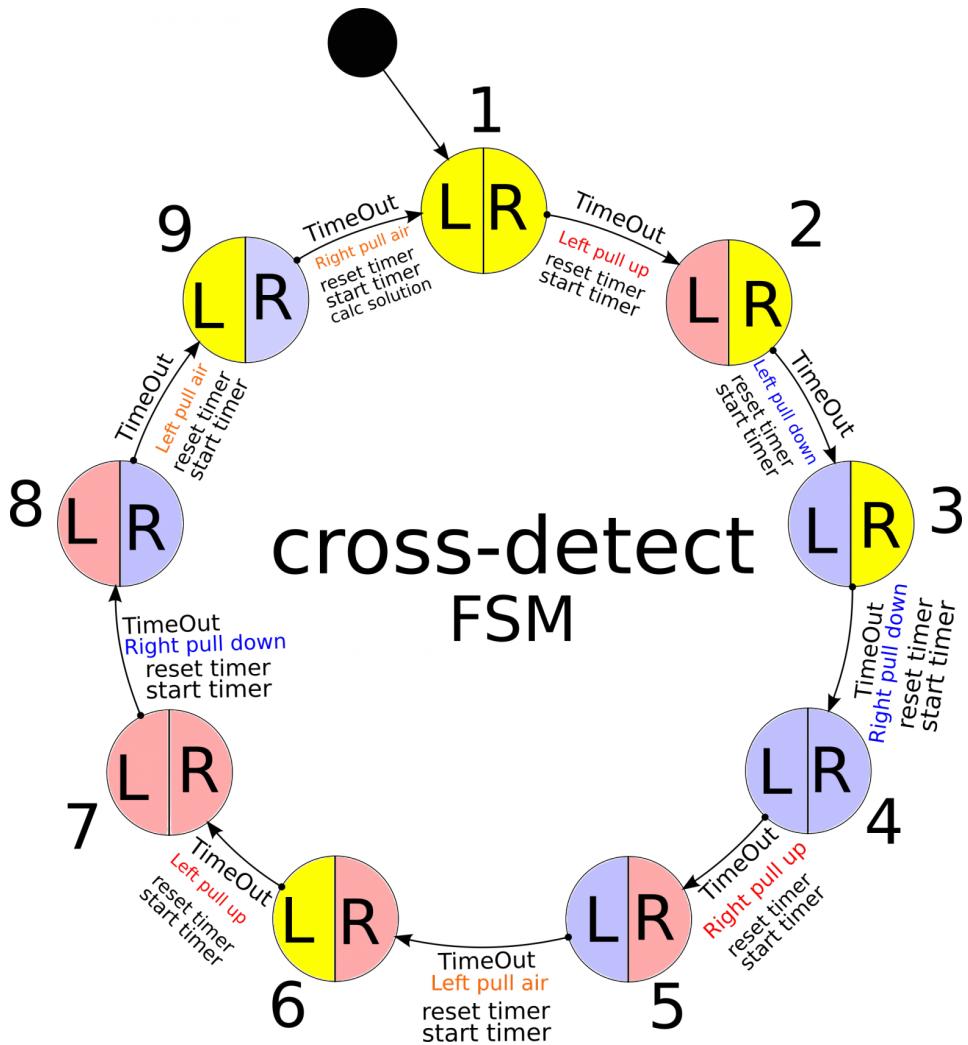


Рис. 22.10: граф для конечного автомата проверки качества пайки

22.13 Сформировать Look-Up таблицу для принятия решения

Вот тут-то и начинается вся аналитика и комбинаторика. Формально, если у нас есть два GPIO пина, которые выходят через форточку на улицу, то согласно комбинаторике, GPIO микроконтроллера может измерить только четыре различные состояния этой пары.

При этом конечный автомат cross-detect за одну итерацию пробегает через девять состояний. И в каждом состоянии формально может быть четыре различных исходов измерения GPIO. Таким образом таблица принятия решения получится из $9 \times 4 = 36$ строчек.

Задача усложняется тем, что в H-мосте может быть одновременно две неисправности. Например на левом плече короткое замыкание на землю, на правом плече короткое замыкание на питание. Один H-мост и две неисправности.

Поэтому переменная, которая отвечает за обнаруженный код ошибки фактически должна быть битовым полем на 5 бит.

Что является признаком того, что в H-мосте два пина не соединены перемыч-

кой? Если GPIO на каждом пине показывают разные значение, значит перемычка отсутствует. Очевидно, что если в пайке оказалась клякса из припоя, то на пинах будет одинаковое напряжение.

Units:->	num	pull		Volts		solution		solution		solution		solution		solution		solution	
		State Num	pull set	pull set	measures	measures	cross both	ShortGND Left	short-VCC Left	ShortGND Right	short-VCC Right	Error --	open both	Ok --			
#		Left	Right	Left	Right												
1	1	pull air	pull air	0	m	m	0	m	0	m	0	m	1				
2	2	pull up	pull air	0	0	m	1	0	m	0	1	m	1				
3	3	pull down	pull air	0	0	2	m	0	m	0	1	m	1				
4	4	pull down	pull down	0	0	m	m	0	m	0	0	m	1				
5	5	pull down	pull up	0	0	m	m	0	1	0	1	m	1				
6	6	pull air	pull up	0	0	m	m	0	1	0	1	m	1				
7	7	pull up	pull up	0	0	m	1	0	1	0	1	m	1				
8	8	pull up	pull down	0	0	m	1	0	m	0	1	m	1				
9	9	pull air	pull down	0	0	2	m	0	m	0	1	m	1				
10	1	pull air	pull air	0	3.3	0	m	0	0	m	1	1	1				
11	2	pull up	pull air	0	3.3	0	1	0	0	m	1	1	1				
12	3	pull down	pull air	0	3.3	0	m	0	0	m	1	1	1	--			
13	4	pull down	pull down	0	3.3	0	m	0	0	0	1	1	1				
14	5	pull down	pull up	0	3.3	0	m	0	0	m	1	1	1				
15	6	pull air	pull up	0	3.3	0	m	0	0	m	1	1	1				
16	7	pull up	pull up	0	3.3	0	1	0	0	m	1	1	1				
17	8	pull up	pull down	0	3.3	0	1	0	0	0	1	1	1				
18	9	pull air	pull down	0	3.3	0	m	0	0	0	1	1	1				
19	1	pull air	pull air	3.3	0	0	0	m	m	0	1	1	1				
20	2	pull up	pull air	3.3	0	0	0	m	m	0	0	1	1				
21	3	pull down	pull air	3.3	0	0	0	1	m	0	1	1					
22	4	pull down	pull down	3.3	0	0	0	1	m	0	1	1					
23	5	pull down	pull up	3.3	0	0	0	1	0	1	0	1	1				
24	6	pull air	pull up	3.3	0	0	0	m	1	0	0	1	1				
25	7	pull up	pull up	3.3	0	0	0	m	1	0	1	1					
26	8	pull up	pull down	3.3	0	0	0	m	m	0	0	1	1				
27	9	pull air	pull down	3.3	0	0	0	m	m	0	1	1					
28	1	pull air	pull air	3.3	3.3	m	0	m	0	m	1	m					
29	2	pull up	pull air	3.3	3.3	2	0	m	0	m	0	1	m	--			
30	3	pull down	pull air	3.3	3.3	m	0	1	0	m	1	1	m				
31	4	pull down	pull down	3.3	3.3	m	0	1	0	0	1	1	m				
32	5	pull down	pull up	3.3	3.3	m	0	1	0	0	m	1	m				
33	6	pull air	pull up	3.3	3.3	2	0	1	0	m	0	1	m				
34	7	pull up	pull up	3.3	3.3	m	0	m	0	m	0	1	m				
35	8	pull up	pull down	3.3	3.3	m	0	m	0	0	1	1	m	1			
36	9	pull air	pull down	3.3	3.3	m	0	m	0	0	1	1	m				

Рис. 22.11: Таблица принятия решения

Как работать с этой таблицей? Очень просто. После одной итерации конечного автомата в программе запускается функция `bool IsLeftShortGnd(Node_t Node)`. Она проверяет, что в состояниях 2, 7 и 8 на левом плече моста всегда GPIO измеряет 0V.

Это и является необходимым и достаточным условием, что левый pin накоротко замкнут на GND.

Аналогично с выявлением короткого замыкания соседних pinов. Если в состояниях 2, 6 на обоих плечах GPIO измеряет 3.3V, а в состояниях 3, 9 GPIO измеряет 0 V, то это верный признак того, что эли два пина соединены перемычкой.

Для полноты картины покажем, что этот автомат также может выявлять короткие замыкания на VCC. Когда на правом плече включена подтяжка к земле, а GPIO измеряет 3.3 V, то это явный признак, что на правом плече на самом-то деле короткое замыкание на VCC.

Как видите, нельзя делать выводы глядя только на одно состояние. Надо проанализировать измерения GPIO во всех девяти состояниях, чтобы однозначно сказать о том, какая именно неисправность присутствует в электрической цепи. По сути это яркий пример того как математическая логика пригождается в программировании микроконтроллеров для решения реальных задач прямо из производства.

22.14 Отладка

Хорошо. Код мы написали. Но как получить отчет работы программного компонента cross-detect?

Обычно для тестов на пропай для каждой версии платы собирают отдельную сборку прошивки с суффиксом IO-Bang. Эта прошивка как раз и содержит компонент cross-detect. Также в прошивке есть UART-CLI для связи с внешним миром, а все пины сконфигурированы на вход.

Вот тут-то нам и пригодится интерфейс командной строки поверх UART. Соединяем плату и LapTop переходником USB-UART, открываем программу TeraTerm, нажимаем help и видим набор доступных команд компонента cross-detect.

Так как этот конечный автомат разрабатывался прежде всего для определения замыканий соседних pinов, то вот это первым делом и проверим. Как видно прошивка в run-time распознала установленную проводную перемычку P0.04-P0.05.

	Num	LGND	LVCC	LeftPad	RightPa	RGND	RVCC	cross		LWireName	RWireName	Upda
0	--	--	P0.4	P0.5	--	--	Cross	A0	A1	156		
1	--	--	P0.4	P0.23	--	--	--	A0	SW1	156		
2	--	--	P0.4	P0.24	--	--	--	A0	SW2	156		
3	--	--	P0.4	P0.8	--	--	--	A0	SW3	156		
4	--	--	P0.4	P0.9	--	--	--	A0	SW4	156		
5	--	--	P0.5	P0.4	--	--	Cross	A1	A0	156		
6	--	--	P0.5	P0.23	--	--	--	A1	SW1	156		
7	--	--	P0.5	P0.24	--	--	--	A1	SW2	156		
8	--	--	P0.5	P0.8	--	--	--	A1	SW3	156		
9	--	--	P0.5	P0.9	--	--	--	A1	SW4	156		
10	--	--	P0.23	P0.4	--	--	--	SW1	A0	156		
11	--	--	P0.23	P0.5	--	--	--	SW1	A1	156		
12	--	--	P0.23	P0.24	--	--	--	SW1	SW2	156		
13	--	--	P0.23	P0.8	--	--	--	SW1	SW3	156		
14	--	--	P0.23	P0.9	--	--	--	SW1	SW4	156		
15	--	--	P0.24	P0.4	--	--	--	SW2	A0	156		
16	--	--	P0.24	P0.5	--	--	--	SW2	A1	156		
17	--	--	P0.24	P0.23	--	--	--	SW2	SW1	156		
18	--	--	P0.24	P0.8	--	--	--	SW2	SW3	156		
19	--	--	P0.24	P0.9	--	--	--	SW2	SW4	156		
20	--	--	P0.8	P0.4	--	--	--	SW3	A0	154		
21	--	--	P0.8	P0.5	--	--	--	SW3	A1	152		
22	--	--	P0.8	P0.23	--	--	--	SW3	SW1	152		
23	--	--	P0.8	P0.24	--	--	--	SW3	SW2	152		
24	--	--	P0.8	P0.9	--	--	--	SW3	SW4	152		
25	--	--	P0.9	P0.4	--	--	--	SW4	A0	152		
26	--	--	P0.9	P0.5	--	--	--	SW4	A1	0		
27	--	--	P0.9	P0.23	--	--	--	SW4	SW1	0		
28	--	--	P0.9	P0.24	--	--	--	SW4	SW2	0		
29	--	--	P0.9	P0.8	--	--	--	SW4	SW3	0		

Рис. 22.12: Прошивка распознала установленную перемычку

Этот конечный автомат проверил 6 проводников каждый с каждым за 22 секунды. По сути получилась прозвонка цепи подобно тому как это происходит в DMM.

22.15 Достоинства cross-detect

1. Использование этого способа тестирования практически ничего не стоит. Всё можно написать с нуля за 3 дня.

Num	LeftPad	RightPa	cross	LGND	LVCC	RGND	RVCC	LWireName	RWireName	Upda
0	P0.4	P0.5	--	--	--	--	Vcc	A0	A1	668
1	P0.4	P0.23	--	--	--	--	--	A0	SW1	668
2	P0.4	P0.24	--	--	--	--	--	A0	SW2	668
3	P0.4	P0.8	--	--	--	--	--	A0	SW3	668
4	P0.4	P0.9	--	--	--	--	--	A0	SW4	668
5	P0.5	P0.4	--	Vcc	--	--	--	A1	A0	668
6	P0.5	P0.23	--	Vcc	--	--	--	A1	SW1	668
7	P0.5	P0.24	--	Vcc	--	--	--	A1	SW2	668
8	P0.5	P0.8	--	Vcc	--	--	--	A1	SW3	668
9	P0.5	P0.9	--	Vcc	--	--	--	A1	SW4	668
10	P0.23	P0.4	--	--	--	--	--	SW1	A0	668
11	P0.23	P0.5	--	--	--	--	Vcc	SW1	A1	668
12	P0.23	P0.24	--	--	--	--	--	SW1	SW2	668
13	P0.23	P0.8	--	--	--	--	--	SW1	SW3	668
14	P0.23	P0.9	--	--	--	--	--	SW1	SW4	668
15	P0.24	P0.4	--	--	--	--	--	SW2	A0	668
16	P0.24	P0.5	--	--	--	--	Vcc	SW2	A1	668
17	P0.24	P0.23	--	--	--	--	--	SW2	SW1	668
18	P0.24	P0.8	--	--	--	--	--	SW2	SW3	668
19	P0.24	P0.9	--	--	--	--	--	SW2	SW4	668
20	P0.8	P0.4	--	--	--	--	--	SW3	A0	668
21	P0.8	P0.5	--	--	--	--	Vcc	SW3	A1	667
22	P0.8	P0.23	--	--	--	--	--	SW3	SW1	664
23	P0.8	P0.24	--	--	--	--	--	SW3	SW2	664
24	P0.8	P0.9	--	--	--	--	--	SW3	SW4	664
25	P0.9	P0.4	--	--	--	--	--	SW4	A0	664
26	P0.9	P0.5	--	--	--	--	--	SW4	A1	0
27	P0.9	P0.23	--	--	--	--	--	SW4	SW1	0
28	P0.9	P0.24	--	--	--	--	--	SW4	SW2	0
29	P0.9	P0.8	--	--	--	--	--	SW4	SW3	0

Рис. 22.13: Регистрация короткого замыкания на VCC

Num	LGND	LVCC	LWireName	LeftPad	RightPa	RWireName	RGND	RVCC	cross	Upda
0	--	--	A0	P0.4	P0.5	A1	--	--	--	176
1	--	--	A0	P0.4	P0.23	SW1	--	--	--	176
2	--	--	A0	P0.4	P0.24	SW2	Gnd	--	--	176
3	--	--	A0	P0.4	P0.8	SW3	--	--	--	176
4	--	--	A0	P0.4	P0.9	SW4	--	--	--	176
5	--	--	A1	P0.5	P0.4	A0	--	--	--	176
6	--	--	A1	P0.5	P0.23	SW1	--	--	--	176
7	--	--	A1	P0.5	P0.24	SW2	Gnd	--	--	176
8	--	--	A1	P0.5	P0.8	SW3	--	--	--	176
9	--	--	A1	P0.5	P0.9	SW4	--	--	--	176
10	--	--	SW1	P0.23	P0.4	A0	--	--	--	176
11	--	--	SW1	P0.23	P0.5	A1	--	--	--	176
12	--	--	SW1	P0.23	P0.24	SW2	Gnd	--	--	176
13	--	--	SW1	P0.23	P0.8	SW3	--	--	--	176
14	--	--	SW1	P0.23	P0.9	SW4	--	--	--	176
15	Gnd	--	SW2	P0.24	P0.4	A0	--	--	--	176
16	Gnd	--	SW2	P0.24	P0.5	A1	--	--	--	176
17	Gnd	--	SW2	P0.24	P0.23	SW1	--	--	--	176
18	Gnd	--	SW2	P0.24	P0.8	SW3	--	--	--	176
19	Gnd	--	SW2	P0.24	P0.9	SW4	--	--	--	176
20	--	--	SW3	P0.8	P0.4	A0	--	--	--	176
21	--	--	SW3	P0.8	P0.5	A1	--	--	--	176
22	--	--	SW3	P0.8	P0.23	SW1	--	--	--	176
23	--	--	SW3	P0.8	P0.24	SW2	Gnd	--	--	176
24	--	--	SW3	P0.8	P0.9	SW4	--	--	--	176
25	--	--	SW4	P0.9	P0.4	A0	--	--	--	176
26	--	--	SW4	P0.9	P0.5	A1	--	--	--	172
27	--	--	SW4	P0.9	P0.23	SW1	--	--	--	172
28	--	--	SW4	P0.9	P0.24	SW2	Gnd	--	--	172
29	--	--	SW4	P0.9	P0.8	SW3	--	--	--	172

Рис. 22.14: Отчет в TeraTerm что пин P0.24 замкнул на GND

- Это чисто софтверный способ тестирования. Из оборудования нужен только копеечный переходник USB-UART.
- Программный компонент cross-detect позволяет определять все 4 типа возможных неисправностей:

4. Cross-detect может выявить короткое замыкание (КЗ) пинов прямо в BGA корпусах, где даже щупом осциллографа или рогом pins к пинам не подлезть! С BGA у Вас только два варианта: просвечивать плату с BGA рентгеновским сканером (X-ray) и глазами искать КЗ между пинами, либо прогонять cross-detect.

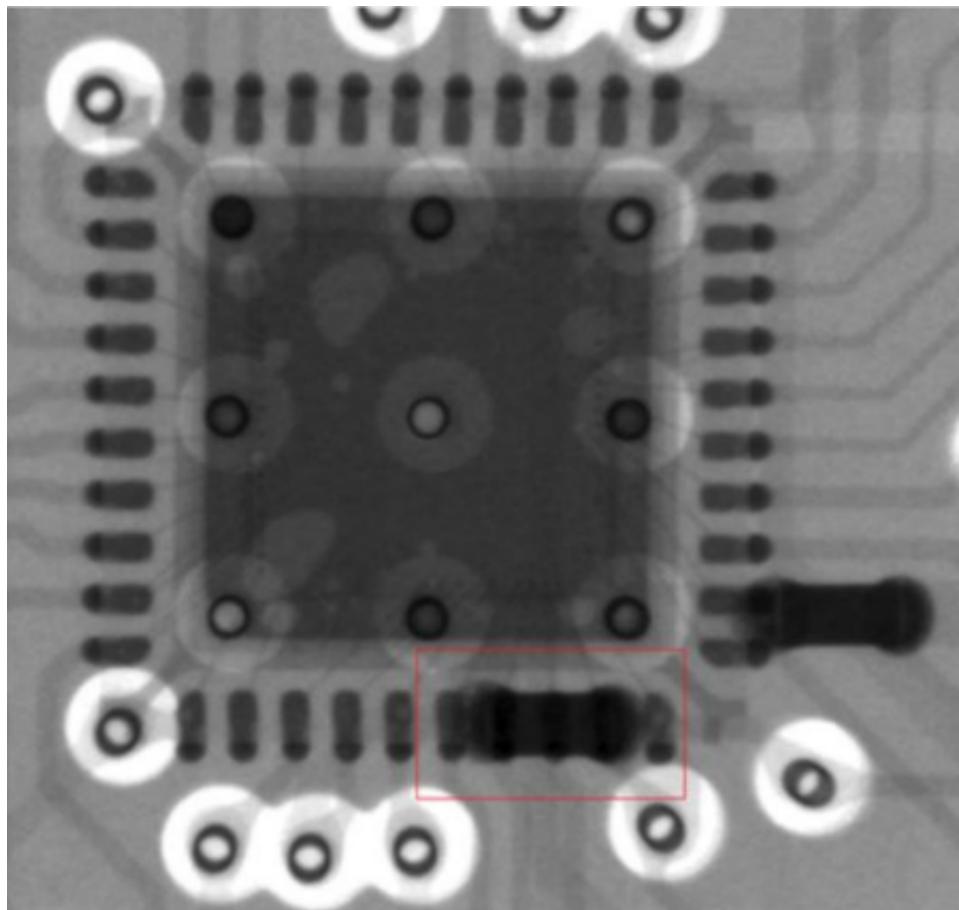


Рис. 22.15: В данном случае рентгеновские лучи не помогут Вам выявить КЗ на соседних пинах

22.16 Что можно ещё улучшить?

На самом деле исходный код cross-detect вовсе не обязательно исполнять на target устройстве. Например на ARM Cortex-M4 или ARM Cortex-M33. Cross-detect - просто алгоритм. Можно соединить PC и Target-PCB интерфейсом JTAG через микросхему переходник USB-JTAG (FT232H), написать на DeskTop консольное приложение симулятор прошивки с CLI(шкой) в stdin-stdout (или GUI) и прокручивать этот же самый конечный автомат опроса GPIO прямо на LapTop(e) NetTop(e). По сути надо заменить CMSIS на код драйвера переходника USB-JTAG, чтобы читать и писать физическую память микроконтроллера. Через интерфейс JTAG можно получить тотальный доступ ко всем подсистемам микроконтроллера или микропроцессора (GPIO, SPI, PLL, RCC, UART и прочее).

При этом управляя микроконтроллером по JTAG прерывания отключить, так

как иначе ваша программа должна будет также определить и таблицу векторов прерываний, как и сами обработчики ISR.

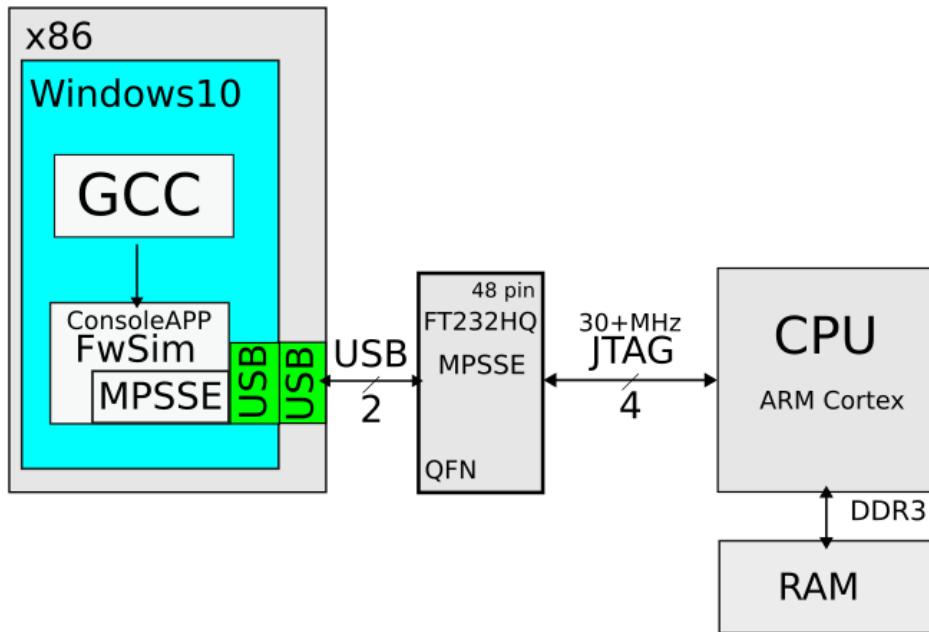


Рис. 22.16: Внешнее управление микроконтроллером программой на РС по JTAG

Если говорить метафорами, то получается микроконтроллер-марионетка. Вместо куклы - микроконтроллер, вместо ниточек 4 провода JTAG. А кукловод - это Windows-процесс на Host PC. Easy.

По сути те же самые утилиты для прошивки по JTAG (например STM32 ST-LINK Utility.exe или nrfjprog.exe) так и работают. Они по JTAG подключаются к карте физической памяти микроконтроллера, обращаются к контроллеру Flash памяти, снимают защиту на запись, записывают по частям бинарь, включают обратно защиту на запись Flash и перезагружают ядро микроконтроллера. Это отлаженная технология начиная с 198х годов.

22.17 Итоги

Вот так и работает методология конечных автоматов при проектировании и разработки ПО.

"Конечный автомат - это золотой молоток программирования микроконтроллеров"

Можно заметить, что благодаря конечно автоматной методологии написание кода по сути сводится к составлению табличек. А таблички, в свою очередь, идеально представляются массивами в языках программирования. В самой прошивке в коде на Си таблицу переходов и таблицу выходов можно для компактности объединить в одной константной статической таблице. Вот и получается, что лучший алгоритм - это таблица.

Стоит заметить что программистам надо уметь рисовать графы автомата, прежде чем кидаться писать код. При этом составленные таблицы и графы плавно перейдут в документацию на ПО. Вполне нормальная ситуация, когда программист 6 дней подряд заполняет таблицы переходов, таблицу выходов, рисует граф автомата, а потом за два дня пишет код, который всё это реализует.

В случае же с примером, если говорить метафорично, то cross-detect - это такие гномы, которые бегают по электронной плате и находят дефекты в производстве PCB. Вообще подтяжки напряжения просто идеально подходят для распознавания коротко-замкнутых проводов и обрывов. По сути плата тестирует сама себя изнутри. Поэтому я и выбрал для демонстрационного примера FSM именно cross-detect , так как cross-detect пригодится вам абсолютно в любой разработке на микроконтроллере для проверки электронной платы на качество пайки.

Надеюсь этот текст поможет программистам микроконтроллеров быстро и эффективно находить дефекты внутри всяческих электронных плат.

Как видите, благодаря этому нехитрому алгоритму можно буквально софтом автоматически определить такие высокоуровневые явления как замыкания на плате! Волшебство да и только!

22.18 Контрольные вопросы

1. Как выявить короткое замыкание между соседними пинами в уже припаянной BGA микросхеме?
2. Электронная плата с производства не работает. Что Вы предпримете чтобы выявить причину поломки?
3. Чем конечный автомат Мура отличается от конечного автомата Мили?

22.19 Гиперссылки

1. Cross-Detect для Проверки Качества Пайки в Электронных Цепях
2. Load-Detect для Проверки Качества Пайки
3. Н-мост: Load Detect (или как выявлять вандализм)
4. Что такое UART-CLI
5. Как перестать писать прошивки для микроконтроллеров и начать жить

Глава 23

Коллоквиум по программированию микроконтроллеров

Это список вопросов для тех кто числится программистом микроконтроллеров и занимается разработкой электроники. Вопросы в частности взяты из технических собеседований при устройстве на работу в разные реальные компании. Постарался отобрать только самые приближенные к практике вопросы, которые можно выделить после 12 лет InSider(ского) опыта. Тут не будет моветонных вопросов из серии "как инвертировать связанный список". Тут представлен обогащенный концентрат. Всё исключительно и только по делу.

23.1 Вопросы по коду

1. Зачем static?
2. Зачем ключевое слово volatile C?
3. Может ли быть const volatile?
4. Зачем ключевое слово register?
5. Зачем ключевое слово restrict?
6. Зачем ключевое слово weak?
7. Зачем в С(ях) нужны битовые поля?
8. Зачем в С(ях) нужны объединения?
9. Как проверить, что в числе установлен/сброшен бит?
10. Как проверить, что два float числа равны между собой?
11. В какую память попадет глобальная переменная с ключевым словом const?
12. Какие есть способы передачи переменных в С функцию?
13. Есть ли способ запустить С-код до запуска main?

14. Зачем нужен препроцессорный error?
15. Какое значение в локальной static переменной при первом вызове?
16. В чем недостаток inline функций?
17. Зачем нужен оператор препроцессора ?
18. Как делать примитивы инкапсуляции в С?
19. Какие знаешь адекватные правила MISRA 2004 или MISRA 2012?
20. Как делать примитивы полиморфизма в С?
21. Напишите функцию, которая при передаче по аргументу значения 1 печатает "One". При передаче 2 печатает "Two". Запрещено использовать оператор if и оператор switch.
22. Может ли С функция во время исполнения определить, что ее вызвали рекурсивно?
23. Может ли С функция с переменным числом аргументов узнать сколько у нее аргументов?
24. Назови три способа вернуть массив из функции.
25. Зачем используют do... while(0); если это всего лишь 1 итерация?
26. Зачем нужен extern "C"?
27. Напишете одной строчкой установку значения 0x11223344 по абсолютному адресу 0x20000016.
28. Как упаковать структуру в компиляторе GCC?
29. Зачем нужны упакованный структуры кроме экономии RAM памяти?
30. Есть константный Си-массив структур, который формируется препроцессором (cpp.exe) до компиляции gcc из разных файлов проекта. Как проверить во время компиляции (до исполнения кода), что в финальном массиве структур нет повторяющихся элементов?

23.2 Системы сборки

1. Что такая система сборки?
2. Зачем использовать какую бы то ни было систему сборки (хоть GNU Make) если можно просто написать *.bat или *.sh скрипт который скармливает компилятору исходники? В cmd или bash скриптах же тоже есть переменные и условные операторы.
3. Чем система сборки Ninja отличается от системы сборки Make?
4. Зачем в GNU Make нужно ключевое слово vpath?

23.3 Структуры данных

1. Чем циклический буфер отличается от FIFO?
2. Как удалить элемент из связанного списка не зная указателя на предыдущий элемент?

23.4 Про DevOps

1. Зачем собирать из скриптов, если всегда можно мышкой щелкнуть на зеленый треугольник в GUI-IDE?
2. Зачем нужны все эти сервера сборки типа Jenkins(а)?
3. Какие файлы следует подвергать версионному контролю в GIT?
4. Что для тебя значит рефакторинг? Что ты подразумеваешь под словом рефакторинг?

23.5 Про прерывания

1. Что такое прерывание?
2. Зачем нужны программные прерывания? Можно ведь просто функцию вызвать.
3. Что такое реентерабельная функция?
4. Сколько тактов процессора нужно для запуска возникшего прерывания на Cortex-M4?
5. Сколько тактов процессора нужно для вызова функции?
6. Что такое таблица прерываний?
7. Каков алгоритм обработки прерываний? Что происходит во время срабатывания прерывания?
8. что такое вектор прерываний?
9. Какие есть внутренние прерывания?
10. Как регистр программного счетчика PC узнает куда возвращаться после обработки прерывания?

23.6 Про ToolChain

1. Как проверить, что конкретный *.c или *.h файл вообще собирается?
2. Какой путь проходит Си-код с момента написания до попадания во flash память?

3. Как отключить/подавить какое-либо отдельное предупреждение компилятора GCC (например -Wrestrict)?
4. Компоновщик пишет, что прошивка не собирается из-за нехватки on-chip NOR-Flash памяти. Какие меры ты предпримешь, чтобы утрамбовать прошивку?
5. Что такое ABI (application binary interface)?
6. На какие сегменты разбита вся память прошивки? На какие сегменты разбита RAM память?
7. Какие доки(спеки) нужны для того, чтобы разрабатывать встраиваемый софт? Назовите минимум 4 дока.
8. Зачем нужен ассемблерный startup код в расширении *.S? Почему нельзя просто взять и обнулить секцию bss функцией memset(), а секцию data проинициализировать функцией memset()?
9. Компилятору подали 5 *.c файлов и 20 *.h файлов. Сколько будет *.o файлов?
10. Тебе предоставили файл *.c чрезвычайно запутанный препроцессором. Как ты поймешь, что там происходит и в какой последовательности?
11. Что такое binutils? Какие знаете? Что можно с ними сделать?
12. Какие расширения файлов являются результатом работы разработчика MCU (артефакты)? Для чего нужен каждый из них?
13. В каких случаях артефакты в *.hex файлах предпочтительнее артефактов в *.bin файлах?

23.7 Вопросы про RTOS(ы)

1. Что такое Bare-Bone сборка RTOS прошивки?
2. Что такое поток?
3. что такое гонки в программах?
4. Что такое bit-banding и зачем нужен bit-banding?
5. Что такое контекст потока?
6. Что такое spinlock?
7. Что такое deadlock?
8. Что такое preemptive многозадачность?
9. Что такая критическая секция?
10. Что такое мьютекс?
11. Что такое семафор?
12. Пример атомарной операции?
13. Что такая инверсия приоритетов?

14. Как бороться с инверсией приоритетов?
15. В стеке какого потока работают прерывания?
16. Что значит thread-safe код?
17. В чём разница между мьютексом и семафором?
18. Что такое Reentrancy?
19. В чём разница между Joined и Detached потоками?
20. Написать функцию атомарного обмена содержимого переменных.
21. Что такое атомарные операции?
22. Как измерить процент загрузки MCU в прошивке без ОС?

23.8 Про цифровые фильтры

1. В чём достоинство цифровых фильтров в отличие от аналоговых?
2. В чём недостаток FIR фильтра в сравнении с IIR фильтром?
3. Дан цифровой FIR фильтр. Исходников нет, реализован в виде статической библиотеки *.a файла. Как узнать массив его коэффициентов $B[0...N]$?
4. Зачем в квадратурном смесителе нужно два смесителя (перемножителя), а не один, три или четыре?

23.9 Вопросы по аналоговой схемотехнике

1. Каким напряжением открывается NPN транзистор?
2. Каким напряжением на затворе открывается P-channel MOSFET?
3. Что такое PUSH-PULL, а что OPEN-DRAIN?
4. Чем резистор, конденсатор и катушки индуктивности отличаются друг от друга? В чём их сходство?

23.10 Про железо (аппаратное обеспечение)

1. В чём отличие режима thumb от режима thumb-2 в процессорных ядрах ARM Cortex-M ядрах?
2. В чём достоинство работы ARM процессора в режиме thumb?
3. Зачем микроконтроллерам функция Pull-Up/Pull-Down, если всегда можно включить LED просто установив логический уровень на GPIO?
4. Как на 8MHz(цтовом) микроконтроллере можно измерить частоту примерно 100MHz прямоугольного сигнала с GPIO?
5. Как проверить, что два PWM сигнала на 2x GPIO синфазные?

6. Какие регистры есть у микропроцессора ARM Cortex-M3 и для чего они нужны?
7. Сколько регистров общего назначения в FPU сопроцессоре в ARM Cortex-M4?
8. Что значит суперскалярный микропроцессор?
9. Почему частота часовогого кварца именно 32768 Hz?
10. Что нужно сделать программе с микроконтроллером, чтобы моргать светодиодом? Напишите словами каждый шаг.
11. Как сделать проверку-защиту, что firmware в самом деле предназначено именно для этой электронной платы?
12. По какому интерфейсу код взаимодействует с железом (ядром микроконтроллера)?
13. Что такое scatter/gather IO?
14. В чем отличия между архитектурами 8051, AVR, ARM, Xtensa, PowerPC, MIPS, RICS-V, x86, SPARC, ARC?
15. Что происходит с микроконтроллером между подачей питания и запуском функции main()?
16. Какие виды памяти есть в микроконтроллере.
17. На какие части обычно делится Flash память?
18. На какие части делится RAM память?
19. Как обрабатывать кнопку? Как преодолевать дребезг контактов?
20. Как при помощи микроконтроллера измерить сопротивление выводного резистора?
21. Зачем внутри микропроцессоров нужен MPU?

23.11 По интерфейсам

1. Какое напряжение на UART TX в режиме idle?
2. Зачем UART опция 2 стоповых бита, если это уменьшает data rate?
3. На одной SPIшине 2 Slave чипа. На оба подали одновременно Chip Select 0V и начали вычитывать регистры в которых разные данные. Что произойдет? Сгорит или не сгорит?
4. Как измерить процент загрузки CAN шины?
5. Какая разность потенциалов в CAN когда ничего не передается?
6. Есть два Lin интерфейса. У одного подтяжка data провода к 24V у другого подтяжка data провода к 12V. Data провода соединили. Что будет? Сгорит /не сгорит?
7. может ли i2c работать в режиме нескольких мастеров?

8. Чем CAN принципиально отличается от Ethernet?
9. У тебя на шине RS485 N устройств. Как мастер устройству узнать количество ведомых устройств на RS485 шине и их 32 битные адреса за минимальное время?

23.12 По протоколам

1. В каких протоколах у переменных big endian, а в каких протоколах у переменных little endian?
2. Зачем в TCP пакете контрольная сумма, если контрольная сумма есть в Ethernet пакете?
3. Зачем нужен IP-адрес, если уже есть MAC-адрес?
4. Как передавать пакеты по 1024 байт, если в Payload транспортного протокола помещается всего только 256 байт?
5. Почему CRC часто в конце пакета, а не в заголовке пакета?
6. Зачем нужно кодирование Base64 в Embedded?
7. Насколько процентов кодировка Base64 расширяет размер оригинального бинаря в самом худшем случае?

23.13 Вопросы про стек

1. Что происходит когда мы вызываем функцию?
2. Что хранится в стековой памяти?
3. Что такое стековый кадр? И что в нем хранится?
4. Какой код копирует в стек адрес возврата?
5. Можно ли на стеке выделить массив длины которого задается аргументом функции?
6. Какой код копирует из стека адрес возврата из функции для регистра программного счетчика?
7. Кто инициализирует локальные переменные если их не проинициализирован явно ?
8. В какую сторону растет стек?
9. Сколько указателей стека в ARM Cortex-M4?
10. Что определяет в каком направлении будет расти стековая RAM память?
11. Какое значение в локальной переменной если ничего не присвоено при создании?
12. Что произойдет при переполнении стека?
13. Как определить на какую максимальную глубину заполнялась стековая память с момента запуска программы?

23.14 Беспроводные интерфейсы

1. Как определить, что передатчик в самом деле передает что-то?
2. Нет радио Link(a) (например в LoRa). Как выявить в чем дело? Передатчик не передает или приемник не принимает?

23.15 Про heap память

1. Как определить размер блока выделенного malloc?
2. Как бороться с фрагментацией памяти?
3. Как проверить сколько памяти выделено в куче в случайном месте программы?

23.16 Про загрузчики (Bootloader)

1. Зачем нужен загрузчик во встраиваемых системах? Назовите минимум 3 его функции.
2. Как загрузчик может обмениваться данными с приложением?
3. В чем опасность вызова функций загрузчика из приложения?
4. Как защитить микроконтроллер от загрузки чужеродного кода через загрузчик?
5. Как загрузчику понять, что загрузчик принял в самом деле прошивку, а не набор случайных циферок с правильной CRC?
6. Как сделать обновление прошивки по TCP/IP, если в загрузчике хватает NorFlash памяти только для драйвера UART?
7. Можно ли сделать так, чтобы загрузчик стартовал не с адреса начала Main Flash 0x0800_0000, а например с адреса 0x0806_0000?
8. Почему в микроконтроллерах STM32 секторы NOR Flash(a) разных размеров?
9. Вам прислали прошивки в *.bin файле. Как загрузить и запустить эту прошивку по произвольному отступу в on-chip Nor Flash памяти?
10. Что код загрузчика должен сделать перед прыжком в приложения на ARM-Cortex-Mx?
11. Каким образом кнопочные Siemens, Motorola, Nokia телефоны могли в run-time до устанавливать игры без пере прошивки микроконтроллера внутри?

23.17 Решение проблем (TroubleShooting)

1. Тебе дали дорогую плату запрограммировать прямо с производства. Плату ещё ни разу не включали в питание. Крайне вероятно, что плата сгорит при

первом же включении из-за брака монтажа. Как ты проверишь плату не испортив ценный полуфабрикат?

2. Прошивка зависла, ваши действия?
3. Какие утечки вы знаете кроме утечки памяти?
4. Прошивка после подачи питания постоянно и непрерывно перезагружается. Как вы станете это ремонтировать?
5. Ты пишешь код, собираешь, запускаешь и вдруг прошивка перезагружается. Твои действия?
6. Как отладить большой кусок кода, если нет возможности пройти JTAG/SWD отладчиком?
7. Какие меры увеличения надежности софта предлагает стандарт ISO-26262?
8. По ходу добавления функционала вы столкнулись с нехваткой RAM памяти для своих глобальных переменных. Как отобразить все глобальные переменные одной командой в консоли?
9. По ходу добавления функционала вы столкнулись с нехваткой ROM памяти. Как отобразить все функции одной командой в консоли?
10. Зачем мультиметру функция True RMS?
11. Что такое полоса пропускания в осциллографе?

23.18 Вопросы для развернутого устного ответа (System Design)

1. Как из одного потока передать массивы разных размеров другому потоку без динамического выделения памяти?
2. Как можно реализовать энергонезависимую Key-Val Map(ку) на микроконтроллере?
3. Как померить процент загрузки микроконтроллера в конкретное время (прошивка NRTOS)?
4. Как можно реализовать надежную доставку пакетов поверх протокола UDP?
5. Как бы ты реализовал механизм FOTA? Т.е. обновления прошивки по беспроводному интерфейсу (Bluetooth, WiFi, LoRa, RFID, UWB и т.п.)?
6. У тебя на шине RS485 N устройств. Как мастер устройству узнать количество ведомых устройств на RS485шине и их 32битные MAC адреса за минимальное время?
7. Чем конечный автомат Мура отличается от конечного автомата Мили?

23.19 Вопросы для проверки навыков пользования компьютером

1. Есть текстовый файл-лог размером 50Mbyte. Строки с ошибками обозначены как [E]. Как узнать есть ли в логе ошибки и сколько их?
2. Диск переполнился. Комп тормозит. Как быстро выяснить размер каждой папки?
3. Как из консоли рекурсивно открыть в Notepad++ все файлы с расширение *.mk?
4. Как рекурсивно удалить все файлы расширения *.bak?
5. Что такое регулярные выражения?
6. Как отобразить все 3-буквенные слова в текстовом файле?
7. Напиши bash команду, которая ищет во всех файлах папки проекта макрос с под именем "LED" только в файлах board.h
8. Как в папке открытой в консоли рекурсивно заменить слово old_word на new_word во всех файлах внутри папки

23.20 Трудные вопросы (со звездочкой *)

1. Как измерить покрытие микроконтроллерного кода после отработки модульных тестов?
2. Опиши как работает JTAG под капотом (установка точки останова).
3. Почему на некоторых MCU RAM память не является непрерывной, а разделена на несколько отдельный непрерывных диапазонов адресов?
4. Как узнать время сборки каждого *.c файла?
5. Как рассчитать CRC на стадии компиляции, чтобы положить результат в константный массив?
6. Как добавить еще одну отладочную кнопку, если уже все пины заняты.
7. Какой путь проходят данные с момента излучения с GNSS спутника до выхода в NMEA протоколе GNSS приемника?

23.21 Вопросы на способность тестирования и отладки

1. Какие существуют способы отлаживать прошивки? Назовите как минимум 10 способов.
2. Какой самый сложный программный или аппаратный баг приходилось искать и починить?
3. Как перезагрузить прошивку? Перечислите как можно больше способов. Минимум 3 способа.

4. Для чего нужны модульные тесты (скрепы)? Назовите 2+ причины.
5. Как отобразить UART лог в коде, который отрабатывает до инициализации отладочного UART?
6. Сколько способов подключить 4 провода к 8 ми клеммникам? Речь идет про конец каждого провода. В один клеммник устанавливается только 1 конец провода.
7. Как избежать чрезмерного, избыточного количества модульных тестов?
8. Как проверить, что инфракрасный передатчик IR в самом деле излучает хоть что-то?
9. Как проверить, что два массива это перестановка одних и тех же чисел?
10. Как протестировать драйвер графического I2C дисплея с SSD1306 в режиме write only?

23.22 Варианты для тестового задания дома

1. Напишите функцию для вычисления угла между 2D векторами с учетом знака (правая тройка).
2. Напишите прошивку под STM32F4, которая генерирует на GPIO два аппаратных PWM с возможностью менять фазу, частоту, скважность через UART в run-time.
3. Напишите энергонезависимую FlashFS(NVRAM) для, например, STM32 микроконтроллера. Предусмотрите endurance optimization и защиту данных от пропадания питания.
4. Напишите heap allocator или попросту реализуйте malloc() free().
5. Даны две GNSS координаты. Вычислить азимут в градусах. Покрыть тестами.
6. Напишите минималистичную прошивку STM32 загрузчика (MBR), которая только прыгает в определенный адрес (например 0x08016000), чтобы запустить приложение. Постарайтесь уместить *.bin файл в 1kByte. У кого меньше бинарь, тот и победил.
7. Напишите диагностическую утилиту интерпретатор 19ти 8ми битных регистров RTC чипа DS3231. Регистровый dump считывать из текстового файла.
8. Напишите Си функцию-переходник, которая преобразует PDM сигнал с MEMS микрофона в PCM сэмплы для загрузки в интерфейс I2S.
9. Написать Си-функцию, которая распознает вещественное число из строчки. То есть универсальный парсер типа данных double. Вот несколько тест кейсов:
".> 0.0; ".5> 0.5; "5.> 5.0; "6> 6.0; "+1e2> 100; "1/3> 0.33;

Глава 24

Как Чинить Баги при Программировании Микроконтроллеров

"Электротехника — это наука о контактах, а вернее — об их отсутствии."



Рис. 24.1:

24.1 Пролог

Программисты микроконтроллеров регулярно занимаются починкой багов. Более того 60 %-80 % работы программиста - это как правило починка багов.

При разработке прошивок вы непременно будете сталкиваться с ошибками в программах.

Зачастую новых программистов нанимают чисто для того, чтобы они непрерывно только и чинили чужие ошибки за преждних программистов.

Самый типичный баг - это зависание прошивки. Переслал мигать heart beat LED, UART-CLI перестала отвечать на команды. В таких случаях не надо подвергаться конвульсиям, судорогам и параличу. Надо просто спокойно и методично разбираться в ситуации.

Выявление причин багов порой работе врача терапевта. Надо поставить правильный диагноз и выбрать способ лечения. При этом порой очень трудно выявить подлинную причину бага. Нужны все виды диагностики.

Починка багов сродни работы детектива. Ты следуешь по горячим следам, мыслишь последовательно, фокусируешься, делаешь гипотезы, ставишь эксперименты, доказываешь или опровергаешь предположения. Делаешь заключения.

Сначала разработчик и вовсе идет по ложному следу, ходит кругами, а конце концов выясняется (порой случайно), что причина на самом деле была проста, как солдатский валенок.

Вот об этом сейчас и поговорим...

24.2 Примеры багов из реальной разработки

У каждого программиста микроконтроллеров с опытом формируется золотая коллекция решенных багов. Баги появляются и исчезают, как вспышки на Солнце. Некоторые из них в высшей степени эпичные. Многие уже читали про ошибку переполнения при конвертации double в int16_t в коде ракеты Ариан-5. Ариан-5 - это уже хрестоматийный пример.

24.2.1 Курящий разработчик

Отлаживаем телематическую плату в CAN сети. Подключили переходник USB-CAN. Автомобиль не заводится. Стартер работает, но зажигания (вспышек в цилиндрах) не происходит. Разобрали половину впускного коллектора. Рукой шевелили дроссельную заслонку. Три часа искали причину поломки. Оказывается в OBD-II разъем попал кусок фольги от папиресок, которые замкнули три крайних пина.

Стоило пинцетом извлечь мусор, как автомобиль завёлся с пол-оборота. Вот так просто и не затейливо.

24.2.2 Толстые пальцы

Схемотехники разработали электронную плату с разъемом Tag-Connect. Плату разрабатывали полгода. Плата скомпонована с плотностью нейтронной звезды. А после сборки выяснилось, что никак не примонтировать самый главный разъем - тот что для программирования. Ручной зажим Tag-Connect не защелкнуть, так как пальцы банально не пролезают между стабилизатором напряжения Traco Power и разъемом Tag-Connect.

Да... Всё не предвидишь. В результате пришлось просить женщин с тонкими пальцами, чтобы соединить программатор и электронную плату.



Рис. 24.2: в OBD-II разъем попал кусок фольги от папиросок

24.2.3 USB-A

При питании от USB электронная плата не включается. Вернее она на мгновение мигнет и сразу отрубается. Сначала я подумал, что происходит это из-за инициализации BLE и UWB. Якобы они сильно просаживают напряжение. Но при отключении всего wireless connectivity из сборки пропадание электропитания осталось. Но потом случайно заметил. Когда USB вилка вставлена до упора, то на плату не подается питание. Однако если же USB вилку аккуратно выдвинуть всего только на полдлины, то так питание поступает! Оказывается был просто

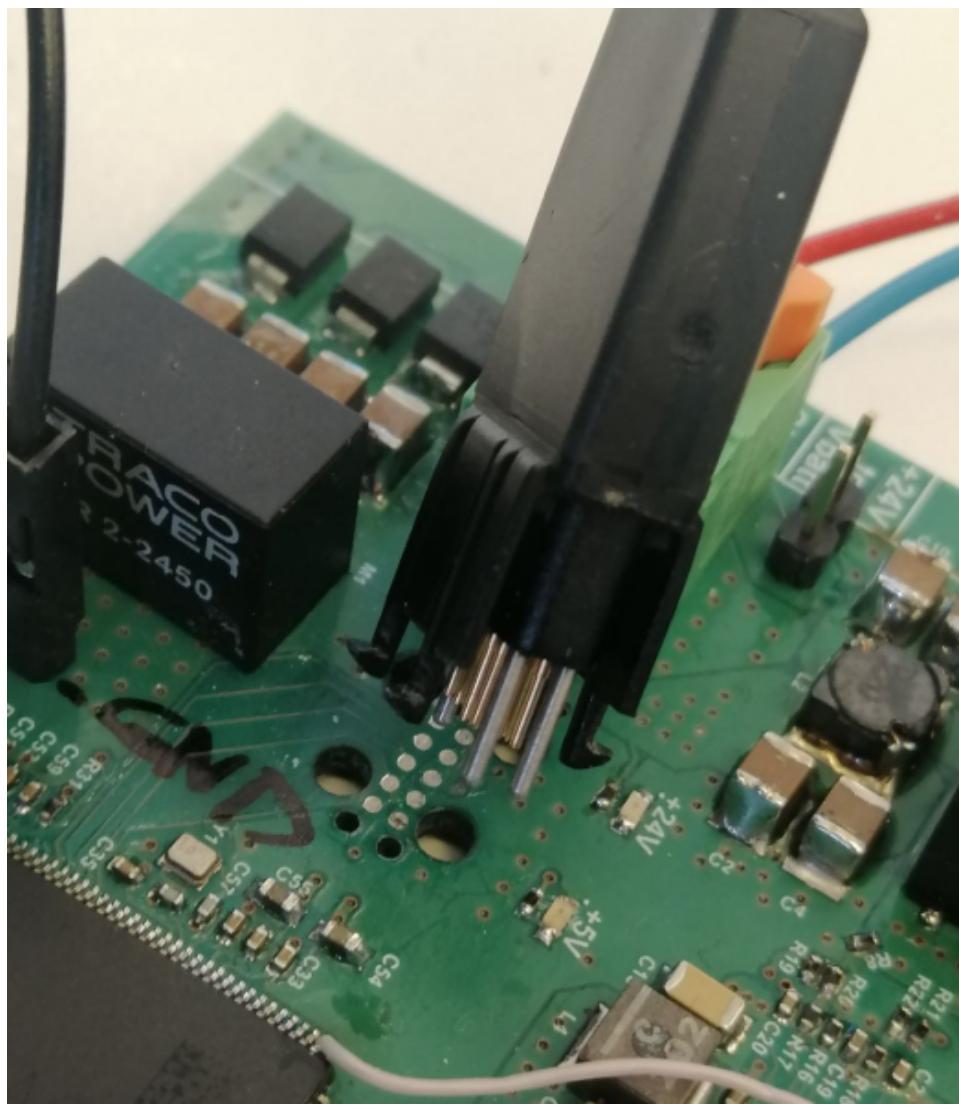


Рис. 24.3: пальцы не пролезают между стабилизатором и разъёмом

бракованный разъём гнезда для USB-A. На плату просто не поступало питание по USB. Мораль: "Не надо жалеть денег на кабели USB".

24.2.4 Slip-ring

Отсутствие SWD link(a). Программатор не видит target по SWD. Собрали стенд, положили SWD длиной 90 см и нет Link(a) с MCU. Когда кабель 12 см, то link есть. Оказывается, что пакеты SWD шины просто не проходили через slip-ring. Slip-ring - это такое устройство, чтобы пропускать провода через подшипник. Чтобы провода не наматывались на ось и не рвались. Пришлось прошивать электронные платы внутри турели навесу.

24.2.5 Сектор конфигурации SoC-а

Плата с CC26x2 зависает при перезагрузке. Плата работает под отладчиком, а при пере сбросе питания не стартует прошивка. На другой плате это не проявляется. Эта ситуация потребовала неделю на выяснение причины и устранение. Баг



Рис. 24.4: турель

чуть не довел меня до полного обезвоживания. Оказывается у CC26x4 MCU от TI надо прописать спец адреса в конце Flash памяти, чтобы чип переключился на внешний осциллятор. Там в последней странице Flash живет сектор конфигурации SoC-а. Кроме как у техасовцев ни у кого больше такого нет. Поэтому во все новые платы надо первым делом записывать прошивку `blinky_backdoor_select_btn26x2.bin` из SDK при помощи утилиты SmartFR Flash Programmer 2. Прошивка `blinky_backdoor_s` корректно настраивает конфигурационные CFG регистры за счет того, что они лягут в последней страницу Flash. Потом уже накатывать свою прошивку поверх. Вот так.

24.2.6 Статическое электричество

Статическое электричество постоянно портит жизнь. Порой ставишь электронную плату на недельный тест, чтобы наработать логи. Приходишь через неделю снять характеристики по UART, ценнейшие логи из RAM памяти. Только дотрагиваешься до края электронной платы... Дзык! Проскальзывает искра статического электричества, которая убивает содержимое RAM памяти и прошивка заклинивает прямо на глазах. И весь недельный тест с драгоценными логами внутри накрывается медным тазом... Пришлось поднимать на устройстве NVRAM и черный ящик в SD карту, чтобы писать логи в энергонезависимую память. Вот такие вот пирожки с капустой... Понимаете?...

24.2.7 Хрипящий голос

Из Bluetooth classic модуля BT1026 по I2S приходит хриплый голос. Сначала подумал, что DMA не так настроено. Микроконтроллер тактирует WS на частоте 50kHz. Оказывается, что если BT1026 опрашивать с частотой дискретизации чаще 48kHz, то по I2S data out будут идти нули. Модуль FSC-BT1026C не может выдавать I2S семплы чаще чем 48kHz. Эти вкрапления нулей и были слышны как хрип. Решение было в том, чтобы добиться частоты следования семплов не выше 48kHz. Микроконтроллер nRF5340 самое близкое мог выдавать на I2S WS только 50kHz. Пришлось сконфигурировать мастером шины I2S сам BT1026. Так звук стал непрерывным, чистым и приятным.

24.2.8 Забыли вставить в розетку

У меня был случай, когда три схемотехника спроектировали электрическую цепь электронной платы (схемотехнику), развели топологию (layouts), отправили производство в другую страну, а спустя три месяца на выходе выяснилось, что они забыли вообще даже подать на микроконтроллер электропитание! Вот тот самый МГТФ проводок, который пофиксил ошибку дизайна.

Вот так... Как говорят:

"У семи нянек дитя без глаза."

24.2.9 Не тот кварцевый осциллятор

Никак не проходит auto-negotiation в Ethernet физике по 100-Base-TX. Не проходит ping. Оказывается в монтажных мастерских техник по ошибке припаял кварц для физики fast-ethernet, не на частоту 25 MHz, а на частоту 23 MHz! Перекинули кварц и ping-и сразу побежали, как горный ручей. Не тот был кварц.

Или вот, плата Olimex-STM32-H407 в UART3 вместо лога загрузки сыплются кракозабры. Оказывается, что прошивка думает, что кварц 25 мегагерц, а там на самом деле 12 мегагерц. Пересобрал прошивку с настройками тактирования от 12 мегагерц и побежал понятный лог на ожидаемой битовой скорости .

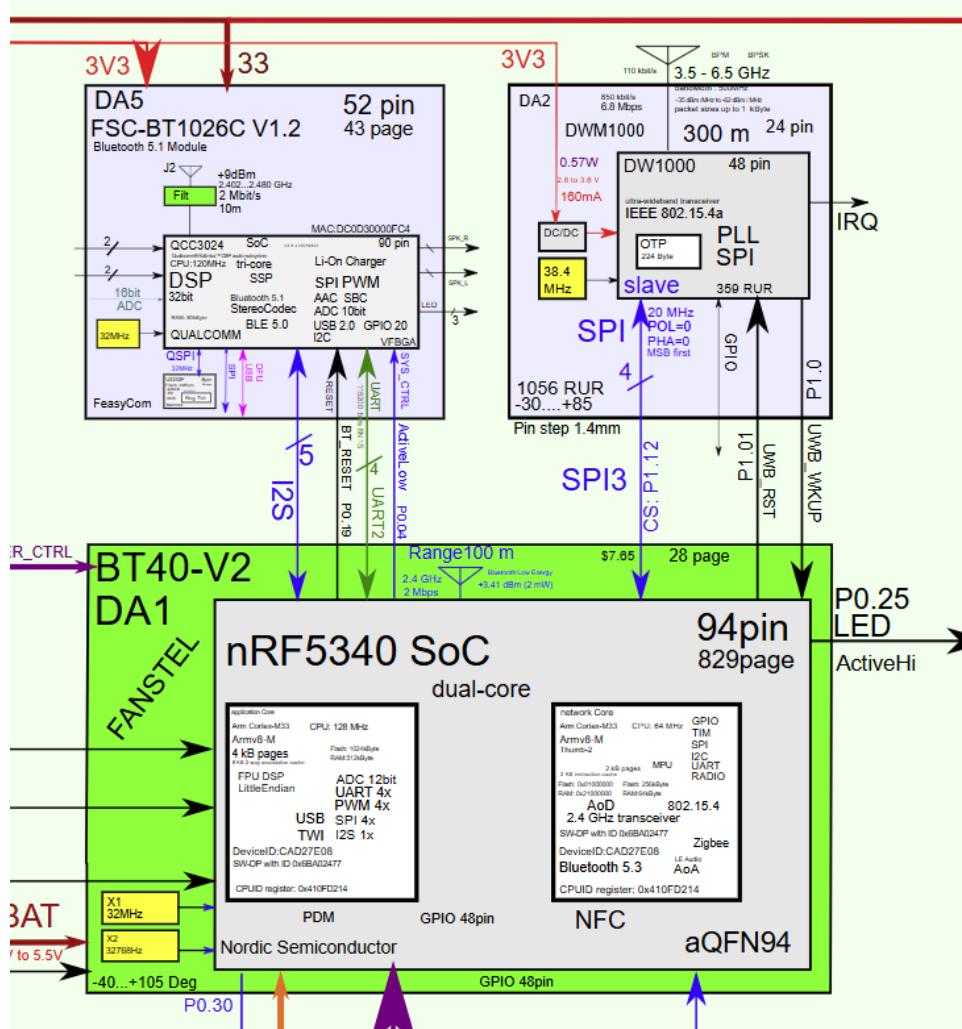


Рис. 24.5: BT1026+nRF5340

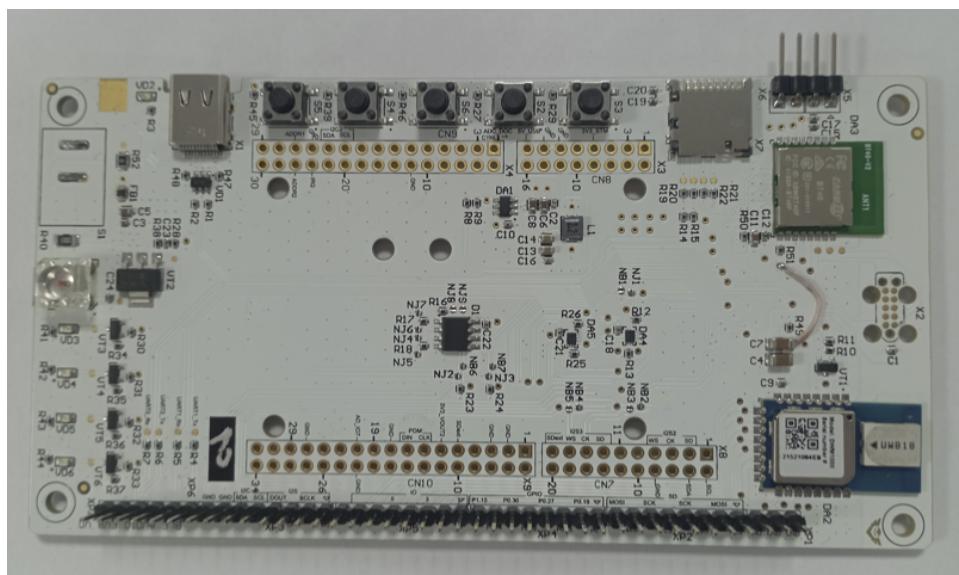


Рис. 24.6: nRF5340 EVB

24.2.10 Ток которого нет

Существует чип VNQ7E100AJ - это драйвер микросхемы 4x канального high-side. High-side - это когда провод коммутируют на шину силового питания. Проблема в том, что чип VNQ7E100AJ при отключённой нагрузке и скважности PWM на входе менее 100 нет (ибо ничего не подключено).

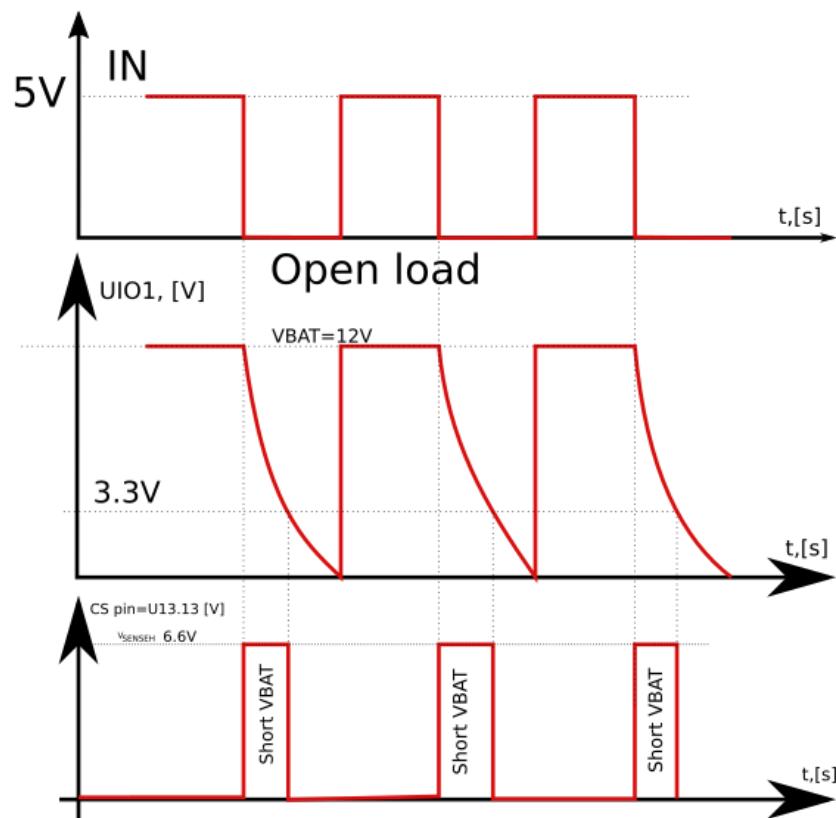


Рис. 24.7: open-load fault

При внимательном изучении спеки выяснилось, что это оказывается такая функция чипа VNQ7E100AJ. Просто VNQ7E100AJ так своеобразно сообщает об ошибке. В случае ошибки выставляет напряжение V_{SENSEH} . По факту происходит подключение Pull-up к pinu CS. На pinе CS получалось 5....6.6V. Ошибкой являлось событие Open-Load.

А прошивка глядя на показания ADC думала, что это ток в силовой цепи так подскочил. Решением стало отказаться от чипов VNQ7E100AJ в следующей ревизии схемотехники ECU в пользу другого драйвера.

24.2.11 Заклинивший CAN трансивер

У нас в организации коллега написал CAN драйвер для микроконтроллера YTM32B1ME05G0MLQ. При интеграции этого драйвера в сборку выяснилось что CAN не работает на повторный приём.

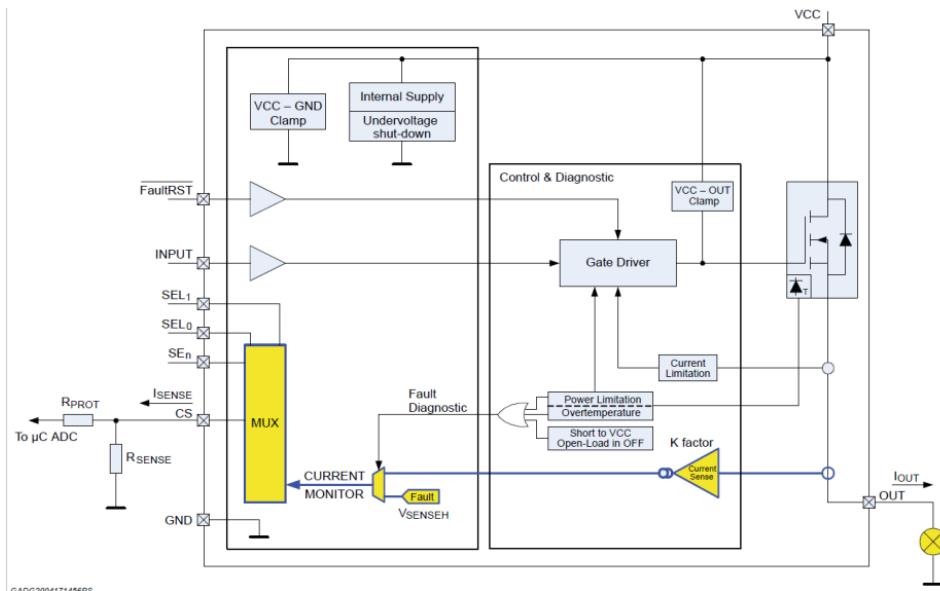


Рис. 24.8: блок-схема VNQ7E100AJ

На железо тут пенять бессмысленно, так как драйвер FlexCAN из SDK от вендора на этом же железе работает безупречно.

Соединил перемычками CAN0 - CAN5. Инициировал отправку пакета в CAN0. Первый вызов CANwriteFrame()1.CAN5.CANwriteFrameCAN0,CAN0,.CAN5CAN0.ACK.

Оказывается что после срабатывания прерывания CAN5Interrupt1631HandlerCANFreeze

Листинг 24.1: Сброс состояния Freeze

```
static RESULT CAN_ReloadLowLevel(CanHandle_t* const Node) {
    RESULT ret = RESULT_NOT_OK;
    if (Node) {
        if (Node->CANx->MCR.B.HALT) {
            /* The CPU should clear HALT after initializing the message buffers and
               the control registers CAN_CTRL1 and CAN_CTRL2 */
            Node->CANx->MCR.B.HALT = 0;
            ret = RESULT_OK;
        }
    }
    return ret;
}
```

24.2.12 Упс! Микроконтроллер перепутали...

Собранная прошивка для платы с MK STM32F413ZGJ6 зависает при прыжке из загрузчика в приложение. При пошаговой отладке всё стартует отлично. При выяснении причины выяснилось, что перед прыжком в приложение загрузчик смотрел на указатель стека и не запускал прошивку, если в таблице векторов прерываний указатель на верхушку стека ссылается на размер RAM больше, чем 128kByte. В таких случаях загрузчик просто прыгал в бесконечный цикл. Загрузчик, к слову, писал коллега из другого дивизиона. При этом напомню на MCU STM32F413ZGJ6 320k Byte RAM! Оказывается коллега разработчик загрузчика

по ошибке всегда собирали прошивку вообще для другого MCU: STM32F205VC. И generic приложение он тоже собирали для STM32F205VC, а записывал на STM32F413ZGx. Поэтому раньше у него это зависание не проявлялось. Его прошивка работала чисто только благодаря обратной совместимости между ядрами ARM Cortex-M4 и ARM Cortex-M3. Оказалось, что ему просто лень было посмотреть, что написано на корпусе микросхемы DD14. Упс!, Микроконтроллер перепутали...

Самое тупое то, что они обнаружили, что программируют не тот микроконтроллер только на пятый год разработки! Это как?

Программисты порой как туземцы какие-то вообще не понимают с чем работают. Попалась им в руки европейская игрушка и давай её вертеть не по назначению.

"Техника в руках дикаря - кусок железа."

24.2.13 Ошибка внутри sprintf()

Прошивка для ARM Cortex-M33 неожиданно сваливается в исключение Default Handler внутри функции sprintf(). А конкретно на функции svfprintf и ассемблерной команде VSTMDB. Оказывается sprintf использует вычисления с действительными числами. При этом код прошивки был собран с активированным аппаратным FPU одинарной точности. Это опции компилятора -mfloat-abi=hard -mfpu=fpv5-sp-d16, одновременно с этим startup код не включал сопроцессоры FPU в регистре SCB->CPACR. Пришлось либо отрубать аппаратный FPU (-mfloat-abi=soft), либо включать сопроцессоры FPU.

Листинг 24.2: Активация сопроцессоров FPU

```
#ifdef ENABLE_FPU
/* Enable CP10 and CP11 coprocessors */
SCB->CPACR |= (3UL << 20 | 3UL << 22);
#endif
```

24.2.14 Загадочные прерывания

Почему-то при инициализации CAN трансивера прошивка зависает в DefaultISR. Сначала я думал, что нет тактирования на CAN трансивере, ибо SoC весьма сложно раздавал тактирование. Однако нет. Тактирование настроено нормально. Это показал вывод частоты "на улицу". Оказывается, при инициализации CAN вызываются прерывания, на которые нет обработчика. Причём это были зарезервированные для данного MCU номера прерываний, для которых для данного MCU по спеке вообще ничего не предусмотрено. И тем не менее, иногда эти номера выстреливают, что приводит к сваливанию прошивки в бесконечный цикл default_ISR. Чтобы починить инициализацию CAN трансивера пришлось определить обработчики прерываний по умолчанию для всех 196 ISR. И определить эти обработчики, как weak функции.

24.2.15 Ошибка 71-ой минуты

Прошивка всегда стабильно зависает на 71-ой минуте работы (4291 секунде). Светодиоды перестали мигать, UART-CLI перестала отвечать на команды. Оказывается переполнилась переменная $up_time_us = 4294967296 \text{ us} = 4\ 294\ 967 \text{ ms} = 4\ 294 \text{ s} = 71 \text{ m}$. И программный компонент limiter просто ждет, когда переменная up_time_us превысит значение 4294967297 us и не вызывает никаких функций. А это и не происходит, так как 32-х битный счетчик up_time_us пошел по второму кругу. Пришлось добавить обработку на переполнение типа данных `uint32_t`.

24.2.16 Низкая дальность LoRa link-a

У нас в компании спроектировали PCB с дальнобойной радиосвязью LoRa. Однако почему-то уже на расстоянии 30 метров обрывается сеанс связи. Радио-команды не поступают. При этом тот же код драйвера SX1262 на покупной плате T-Beam работает аж на 15 километров. Это было специально проверено на полигоне. Оказывается на нашей плате схемотехник сильно отошел от reference дизайна и заложил ключ BGS12WN6E6327XTSA1 с инверсными логическими уровнями. Проще говоря, когда LoRa трансивер передавал сигнал, антенна была просто отключена!

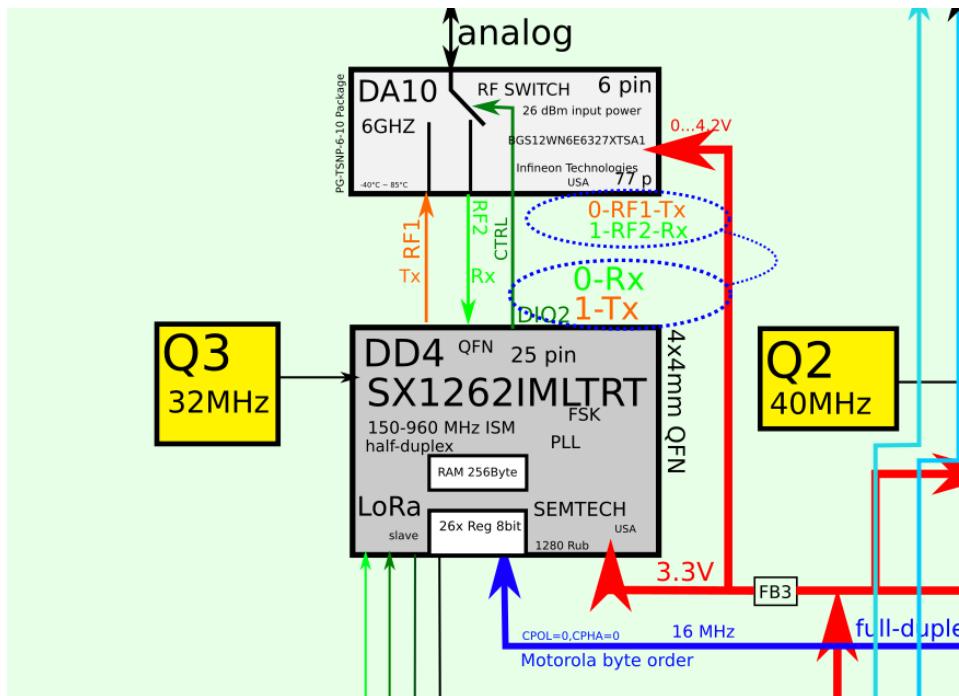


Рис. 24.9: SX1262 и BGS12WN6E6327XTSA1 не совместимы по логическим уровням

Когда трансивер принимал, мультиплексор DA10 подключал антенну на передатчик и ASIC ничего не принимал. Вернее антенна была в виде дорожки PCB длиной 7 миллиметров. Вот так...

24.3 В чём проблема починки ошибок?

1. Проблема в том, что все баги они уникальные! Поиск причины каждого конкретного бага это искусство.
2. В программировании микроконтроллеров часто надо уметь отличить программную ошибку от аппаратной ошибки. Обычно как железо, так и прошивку делает одна и та же организация. Поэтому ошибки в электронной плате - это вообще норма жизни.
3. Бывают ошибки, которые редко воспроизводятся: Гейзенбаги.

24.4 Как Чинить Программные Ошибки?

24.5 Терминология

Баг (bug) - ошибка в компьютерной программе. Несоответствие между реальным поведением программы и ожидаемым поведением программы.

Гейзенбаг (плавающая ошибка) - программная ошибка, которая исчезает или меняет свои свойства при попытке её обнаружения.

24.6 Классификация программных ошибок

Вот классификация программных ошибок.

1. Печатаешь текст и на экране ничего не происходит. Через 20сек разом вываливается большой текст.
2. Прошивка свалилась в Hard Fault ISR
3. Прошивка постоянно непрерывно перезагружается
4. Не проходит инициализация какого-то программного компонента. Например FatFs.
5. Из интерфейса поступают испорченные данные. Например зашумленный сигнал.
6. На выходе расшифровщика не понятный текст, а кракозябры
7. Программный компонент не отвечает на команды
8. Подали питание и Heart Beat LED не мигает
9. Не отвечает UART-CLI

24.7 Классификация причин программных ошибок

Вот некоторые типичные причины программных багов из реальной повседневной разработки программного обеспечения:

1. Произошло переполнение типа данных
2. Неверно подобранные приоритеты потоков в RTOS
3. Переполнение стековой памяти
4. Попытка прописать read only память (.rodata)
5. Произошло деление на ноль
6. Произошло обращение к нулевому указателю
7. Сработало прерывание для которого нет обработчика
8. Произошел выход за границы массива. Код перетер чужую RAM память за пределами массива. Глобальные переменные больше не валидные. Последствия непредсказуемы...
9. Неверный конфиг для программного компонента
10. Кто-то закомментировал кусок кода, который должен был исполняться
11. Программа застряла в нежелательном бесконечном цикле while(1)
12. Функция вычисления формулы вернула NaN вместо double
13. Неверный порядок инициализации
14. Использование неинициализированной переменной в которой случайное значение
15. Сработало прерывание SysTick таймера до инициализации планировщика RTOS
16. Возникло состояние гонки (race condition). Например, два потока пытаются одновременно прописать в on-chip NOR Flash.

24.8 Универсальная методичка починки багов

Я считаю, что можно в какой-то степени обобщить алгоритм поиска программной ошибки. Надо просто следовать вот по такому алгоритму:

1. Подробно описать программную ошибку.
2. Научиться стабильно воспроизводить баг.
3. Выдвинуть несколько предположений.
4. Делать эксперименты.
5. Активировать более глубокий уровень логирования.
6. Прогнать модульные тесты.
7. Пройти код пошаговым GDB отладчиком.
8. Прогнать код через статический анализатор.
9. Задать вопрос в профессиональных форумах и сообществах.

А теперь обо всём по порядку...

24.8.1 Фаза № 1. Подробно опишите программную ошибку

Подробно напишите, что именно не так. Не жалейте слов. В чем именно несоответствие. Что должно быть, а что происходит на самом деле. Добавьте скриншот. Добавьте кусок лога программы в этом месте. Заведите про эту ошибку текстовый файл или заметку в трекере задач. Это подобно тому, как полиция делает фотографии на месте преступления. Составьте дефектную ведомость.

24.8.2 Фаза № 2. Научиться стабильно воспроизводить баг

Это очень важно. Баг надо воспроизвести. Надо составить подробный алгоритм, который приводит к воспроизведению программного бага. Если нет возможности локально воспроизвести баг, то дальше и говорить не о чём. Без этого Вы будете просто ходить кругами и фантазировать.

Чтобы отлаживать код, его надо исполнять. Если в коде баг и нет возможности исполнить код, то баг не исправить.

Это как писал философ Э.В. Ильенков, если найти в лесу отдельно лежащую человеческую ногу, то не ясно что это и зачем. Понятно только, когда эта нога видна в действии: в составе танцора на сцене или просто идущего по улице человека. Аналогично в понимании причины программных багов. Надо запускать код...

Чтобы лёд тронулся надо сперва научиться регулярно воспроизводить баг, желательно автоматически из какого-нибудь скрипта. Если Вам сказали, что обнаружен баг, то первый вопрос, который вы должны задать себе это: "Настолько стабильно баг воспроизводится?".

24.8.3 Фаза № 3. Выдвинете предположения (гипотезы)

Выдвинете предположения. Составьте гипотезы. С опытом Ваши предположения будут всё более и более точными. Тут надо выдвинуть как можно больше предположений. Три, четыре, восемь. Запишите все возможные предположения почему происходит этот баг.

24.8.4 Фаза №4. Делайте эксперименты

Для каждой гипотезы придумайте эксперимент, который доказывает или опровергает гипотезу (предположение). Эксперименты это компиляция и запуск кода с разными конфигами с последующей попыткой воспроизведения бага. Сразу обнаружится, что некоторые гипотезы несостоятельны. Тут же надо отразить эти наблюдения в текстовом описании бага (дефектной ведомости).

24.8.5 Фаза №5. Активировать более глубокий уровень логирования

Включите уровни логирования того компонента в котором возникла ошибка до уровня Debug или Paranoid. В хорошей прошивке, должен быть UART-CLI. CLI

позволяет включать/отключать уровни логирования для каждого программного компонента. Вероятно баг вызван тем, что не выполняется какой-то важный код, который должен выполняться.

24.8.6 Фаза №6. Прогоните модульные тесты

Покройте функционал модульными тестами. Это не формальность. Модульные тесты являются скрепами, которые сдерживают корректный функционал при перестройке программы. Очень вероятно, что отчет модульных тестов в UART-CLI сузит место поиска причины бага, либо явно укажет на причину бага. Модульные тесты это отличный способ отладить большой кусок кода, который невозможно или проблематично пройти пошаговым отладчиком.

24.8.7 Фаза №7. Пройдите код пошаговым GDB отладчиком

Это крайняя мера. Проходите код пошаговым отладчиком. Каждую строчку. Анализируйте по какой ветви происходит исполнение кода. Правильные ли значения лежат в локальных и глобальных переменных? Вероятно не выполняется нужная ветвь кода. Пошаговая отладка это весьма утомительное мероприятие. Особенно когда приходится делать её из CLI.

24.8.8 Фаза №8. Прогоните код через статический анализатор.

Это скорее профилактическая мера. Есть такие утилиты, которые анализируют код без его исполнения. Это например CppCheck.exe. Они часто находят нелепые опечатки, которые не замечает компилятор. Минимум раз в месяц надо прогонять код репозитория через статический анализатор. Сами вы всё не уследите.

24.8.9 Фаза №9. Задайте вопрос на профессиональных форумах и сообществах.

Это скорее фаза отчаяния. Эта мера абсолютно не гарантирует результата. Никто вам ничего там не должен. Но как попытка вполне легальное действие.

24.9 Итог

Как видите, проблем может быть просто море и они поджидают вас абсолютно везде. И чтобы понять, что происходит, порой нужно воистину нешаблонное мышление.

При хорошем DevOps можно отловить много багов на стадии статического анализатора, или отработки скриптов сборки (Make, CMake). Потом можно отловить баги на фазе препроцессора (утилита cpp.exe), затем отловить баги на фазе компилятора (gcc.exe). Ошибки также показывает компоновщик (ld.exe). Часть

ошибок могут отловить модульные тесты уже в run-time(e). Последний рубеж обороны от ошибок это интеграционные тесты.

Однако тем не менее ошибки просачиваются в релизные артефакты (*.elf, *.bin, *.hex) и обнаруживаются только на фазе исполнения программы (run-time(e)).

Каждый программный баг он уникальный. Однако есть одно общее правило буравчика, которое справедливо для разбора любого программного сбоя.

"Чтобы найти причину программной ошибки надо научиться стабильно воспроизводить ошибку на фазе исполнения."

По крайней мере с этим можно будет дальше работать. А просто разглядывать код с умным видом в надежде найти причину бага - это бессмысленное занятие.

Компилировать сорцы в уме, а потом фантазировать на тему, как этот код будет исполняться - это не наши методы. Код с багом надо запускать на target устройстве и анализировать ситуацию по горячим следам.

В программировании микроконтроллеров до того, как вы дойдете до всяких так алгоритмов и структур данных пройдет месяц, а то и два колупания с электропитанием, макетированием, тактированием, прерываниями и прочим. А проблемы отсутствия всяческого Link(a) в программировании микроконтроллеров и вовсе красной нитью прошиваю всю мою карьеру. Особенно в случае беспроводных интерфейсов.

На сам процесс программирования уходит максимум 10...30 процентов времени. В основном приходится что-то бесконечно ремонтировать.

Суммируя аппаратные баги, это либо ошибка в схемотехнике или ошибка в топологии.

Баги в разработке на микроконтроллерах это совершенно нормальное явление. Главное чтобы из них делали выводы. Надо уметь выявлять их причину.

Если у вас тоже случались культовые неочевидные баги в разработке на MCU и вы нашли причину и решение, то напишите мне про это пожалуйста.

24.10 Гиперссылки

1. Модульное тестирование для микроконтроллеров
2. 16 Способов Отладки и Диагностики FirmWare
3. Пошаговая GDB отладка ARM процессора из консоли в Win10
4. Модульное Тестирование в Embedded
5. Почему Нам Нужен UART-Shell?

Глава 25

Литература

"Из толстых книг нельзя узнать ничего нового. Толстые книги — это кладбище, в котором погребены отслужившие свой век идеи прошлого"
(Лев Давидович Ландау)

1. Эффективное использование GNU Make, Владимир Игнатов, 2000
2. Автоматное программирование, Поликарпова Н. И., Шалыто А. А., 2008
3. Программирование введение в профессию, 4 Тома, А.В. Столяров, 2016
4. "Как стать специалистом по встраиваемым системам"пособие для тех, кто хочет заниматься интересным и хорошо оплачиваемым делом, Левин Э.
5. Искусство схемотехники. Часть первая. Аналоговая Издание 3-е, часть первая , Хилл У., Хоровиц П.
6. Джозеф Ю. Ядро Cortex-M3 Компании ARM. Полное руководство.
7. Архитектура встраиваемых систем, Даниэле Лакамера, ДМК-Пресс, 2023,332 стр,ISBN: 978-5-93700-206-8
8. The Power of Ten—Rules for Developing Safety Critical Code, Gerard J. Holzmann, 6 pages
9. Цифровая обработка сигналов. Практическое руководство для инженеров и научных работников, Стивен Смит,2018
10. Ядро Cortex-M3 компании ARM Полное руководство, Joseph Yiu, 2015
11. Extreme C, Kamran Amini, 2019
12. Язык С в XXI веке, Бен Клеменс , 2015
13. Абстракция данных и решение задач на C++, Фрэнк М Каррано, Джанет Дж. Причард, 2003
14. Структуры данных и алгоритмы , Ахо, Хопкрофт, Ульман, 2021
15. Алгоритмы Руководство по разработке , Стивен С. Скиена , 2011
16. Алгоритмические трюки для программистов, Генри Уоррен-мл., 2014
17. Современные операционные системы, Эндрю Таненбаум,Херберт Бос, 2016

18. Test Driven Development for Embedded C (James W. Grenning), 2011, ISBN-10: 1-934356-62-X
19. Introduction to Microcontrollers and Embedded Systems, Tyler Ross Lambert, Auburn University