

# Фундаментальные Основы Программирования Микроконтроллеров

aabzel

4 декабря 2024 г.



# Оглавление

<b>1</b>	<b>Об авторе</b>	<b>5</b>
<b>2</b>	<b>Атрибуты Хорошей Прошивки (Firmware)</b>	<b>7</b>
2.1	Микросекундный UpTime счетчик (камертон)	7
2.2	Сторожевой таймер (сторожевой пёс)	7
2.3	Загрузчик (BootLoader)	7
2.4	NVRAM (числохранилище)	8
2.5	Модульные тесты (скрепы)	8
2.6	Health Monitor (медбрат)	8
2.7	Full-Duplex Command Line Interface (CLIшка)	8
2.8	Диагностика	9
2.9	Аутентификация бинаря (optional)	10
2.10	Black-Box Recorder (Чёрный ящик)	10
2.11	Переход в энергосбережение	11
2.12	Итоги	11
<b>3</b>	<b>Архитектура Хорошо Поддерживаемого драйвера для I2C/SPI/MDIO Чипа</b>	<b>13</b>
3.1	xxx_drv.c/ xxx_drv.h с функционалом	13
3.2	xxx_isr.c/xxx_isr.h	14
3.3	Файл xxx_types.h с типами*	14
3.4	Отдельный xxx_const.h файл с перечислением констант*	15
3.5	xxx_param.h файл с параметрами драйвера*	16
3.6	config_xxx.c/config_xxx.h файл с конфигурацией по умолчанию*	17
3.7	xxx_commands.c/xxx_commands.h файл с командами CLI*	17
3.8	xxx_diag.c/xxx_diag.h файлы с диагностикой*	18
3.9	test_xxx.c/test_xxx.h Файлы с модульными тестами диагностикой*	19
3.10	Make файл xxx.mk для правил сборки драйвера из Make*	19
3.11	xxx_dep.h файл с проверками зависимостей	20
3.12	Должен быть файл xxx_preconfig.mk	22
3.13	Функционал драйвера	23
3.14	Итоги	26
3.15	Гиперссылки	26

<b>4</b>	<b>Почему важно собирать код из скриптов?</b>	<b>27</b>
4.1	Пролог . . . . .	27
4.2	Как отлаживаться . . . . .	34
4.3	Достоинства сборки кода из Make файлов . . . . .	34
4.4	Аналогии . . . . .	37
4.5	CMake или GNU Make? . . . . .	38
4.6	Кто собирает код из скриптов? . . . . .	40
4.7	Вывод . . . . .	40
4.8	Гиперссылки . . . . .	42
<b>5</b>	<b>Литература</b>	<b>43</b>

# 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. Далее работал стартап проектах, которые разработали серию IoT приборов для умных домов. Последние 7 лет разрабатывает system software для электроники в автопроме. Автор написал firmware более чем для шестидесяти электронных плат на основе микроконтроллеров с архитектурами AVR, ARM и Power PC. С 2012 года автор обладает 12-летним опытом и экспертизой в проблемах разработки программного обеспечения для встраиваемых система на основе современных микроконтроллеров.



## Глава 2

# Атрибуты Хорошей Прошивки (Firmware)

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 издѣвке судьбы в 9 из 10 российских embedded компаний нет и не знают ни одного из перечисленных атрибутов.

### 2.1 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, LoadDetect. Реализовать этот счетчик можно на SysTick таймере или на периферийном Timer.

### 2.2 Сторожевой таймер (сторожевой пѣс)

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

### 2.3 Загрузчик (BootLoader)

Программатор есть далеко не всегда. Программатор часто не видит микроконтроллер из-за статического электричества или из-за длинного шлейфа. Часто программатор в одном единственном экземпляре на всю компанию в целом здании. Загрузка программатором это чисто developer(ская) прерогатива. У Customer нет и

никогда не будет отладчика и особого шлейфа для PCB. 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 плате (PCB) USB, UART, RS485, CAN, LIN, BLE, 100BaseTX, WiFi, LoRa и пр.

## 2.4 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 допрограммировать уже в run-time(e) просто исполнив несколько команд в CLI. Также FlashFS позволяет передать сообщение от приложения загрузчику и наоборот.

## 2.5 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.

## 2.6 Health Monitor (медбрат)

Это отдельная задача, поток или периодическая функция, которая периодически проверяет все компоненты, счетчики ошибок и в случае, если возникли какие-то сбои, НМ как-то сообщает об этом пользователю. НМ 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.

## 2.7 Full-Duplex Command Line Interface (CLIшка)

CLI наверное самое полезное. Как её только не называют: "CUI" "TUI" "командный интерфейс" "Shell" "Bash" подобная консоль Real Time Terminal (RTT), "Терминал" "Printf отладка" "CLI" "Консоль UART Debug Termianl. Это уже намекает на её супер полезность. Однако суть одна. Это текстовый интерфейс командной строки



поверх UART(или UDP, TCP, RS232). Подойдет любой Full/Half Duplex интерфейс. Чтобы общаться с устройством на человеческом языке. Запрос-ответ. Как в Linux только в случае с микроконтроллером.

С помощью CLI можно запускать модульные тесты софта и железа, просматривать куски памяти, отображать диагностические страницы, управлять GPIO, пулять пакеты в SPI, UART, I2C, 1Wire, I2S, SDIO, CAN, испускать PWM(ки), включать/выключать таймеры. В общем CLI для тотального управления гаджетом. Стоит заметить, что CLI это, к слову, единственный способ отлаживать прошивку в микроконтроллерах без JTAG (AVR, ESP32).

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. Только с CLI можно делать Non Invasive Debug. Когда есть CLI(шка), команды установки уровней логирования и чтения памяти по адресу, то отпадает даже необходимость в пошаговой отладке по JTAG. Вернее JTAG/SWD понадобится только для отладки UART и запуска CLI. Далее отладка только через CLI.

С помощью CLI вы сможете до-программировать поведение устройства уже после записи самой прошивки во Flash. Можно хранить CLI-скрипты на SD карте и запускать их подобно \*.bat файлам. Как OS устанавливает и запускает утилиты. Просто подключившись к UART через TeraTerm/Putty/HTerm и отправив несколько команд. Также CLI выступает как интерпретатор команд. А без CLI вам бы пришлось варить еще кучу сборок с какой-то специфической функцией на 1 раз. А потом поддерживать на плаву этот зоопарк проектов.

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

Посмотрите какая классная CLIшка у швейцарского U-Blox ODIN C099-F9P

Посмотрите какая классная CLIшка у китайского STM32 устройства NanoVNA V2 Zephyr RTOS из коробки со своей собственной CLI.

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

Даже авторитетные авторы FreeRTOS, рекомендуют использовать CLI.

## 2.8 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?

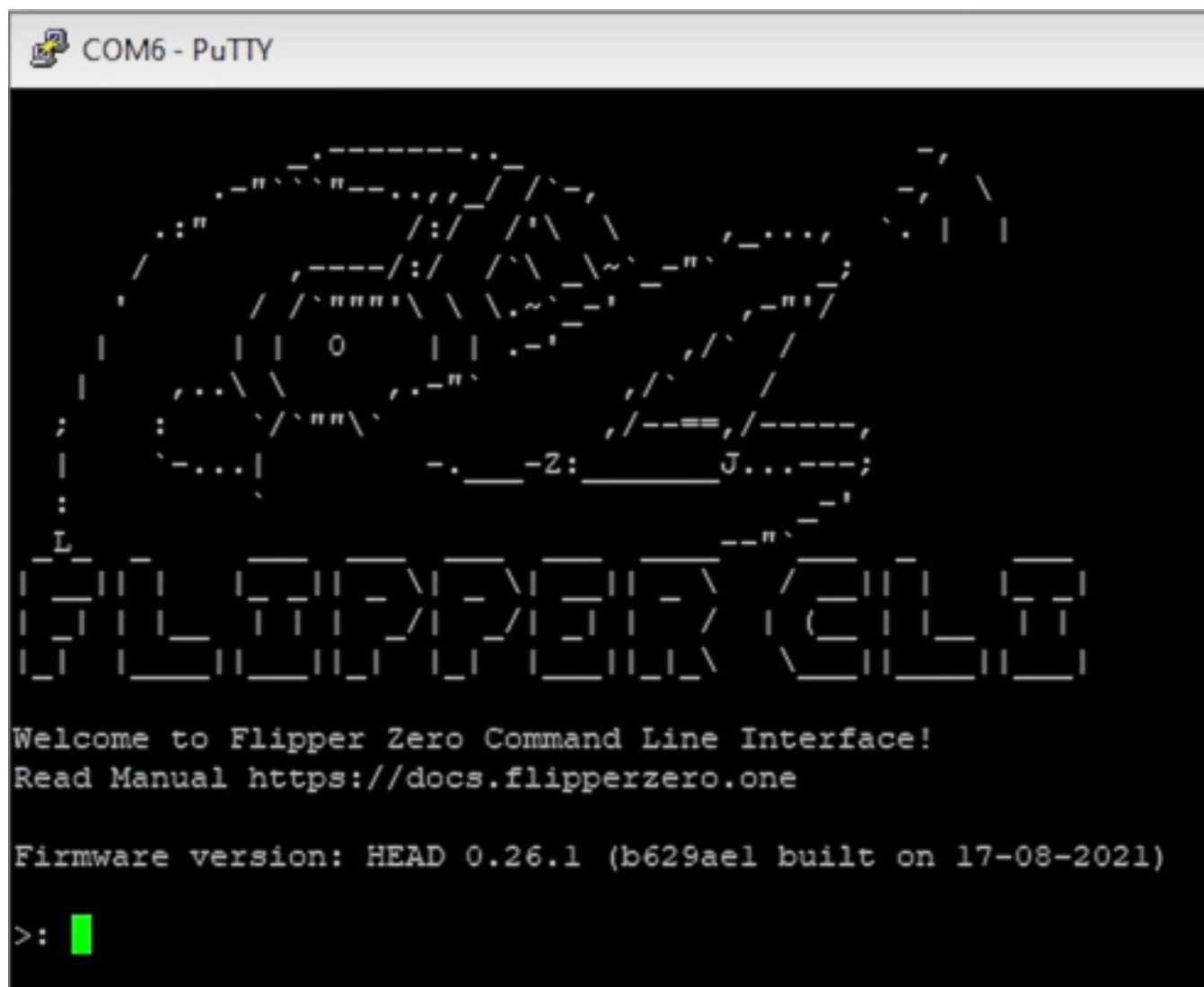


Рис. 2.1: CLI у Flipper Zero

## 2.9 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.

## 2.10 Black-Box Recorder (Чёрный ящик)

У него много имен: черный ящик. LogBook, BlackBox, Прошивка может записывать логи не только в UART(который может никто не смотреть) но и в NorFlash 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.



Рис. 2.2: CLI у U-Blox ODIN C099-F9P

## 2.11 Переход в энергосбережение

Можно по команде из UART-CLI понизить частоту системной шины до минимума. Отключить тактирование от ненужной периферии и таким образом микроконтроллер перейдет в режим низкого энергопотребления. Это особенно важно если устройство работает от батареи.

## 2.12 Итоги

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

Это сборка собранная из GNU Make скриптов, без RTOS, в которой есть UART-CLI, NVRAM, heart beat LED, (кнопка) и модульные тесты вызываемые из CLI. Также следует сразу подготовить отдельную сборку загрузчика через UART-CLI.

Только на этой почве можно полноценно начинать наращивать любой функционал.

Без этого минимума-минимумов будет не разработка, а стрельба из лука с завязанными глазами.

Как по мне эти 11 атрибутов являются просто джентльменским набором 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 не следует. Как ни крути, но сначала надо поднять систему. Писать прошивку без этих 7-10 свойств - это как ходить по улице без одежды.

```

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 U2 info
SN:                  usage: NanoUNA-F U2 Serial Numbel

```

Рис. 2.3: CLI y NanoVNA V2

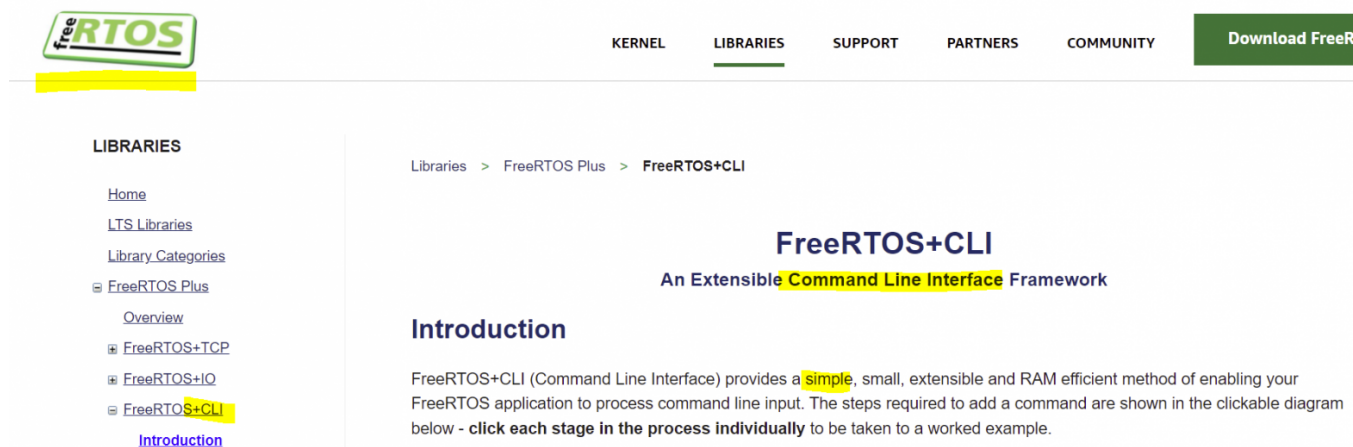


Рис. 2.4: CLI y FreeRTOS

## Глава 3

# Архитектура Хорошо Поддерживаемого драйвера для I2C/SPI/MDIO Чипа

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:

### 3.1 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.

## 3.2 xxx\_isr.c/xxx\_isr.h

Отдельные файлы с кодом драйвера, который должен обрабатывать 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.

## 3.3 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 3.1: 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. Этому же принципа логично придерживаться и в разработке кода.

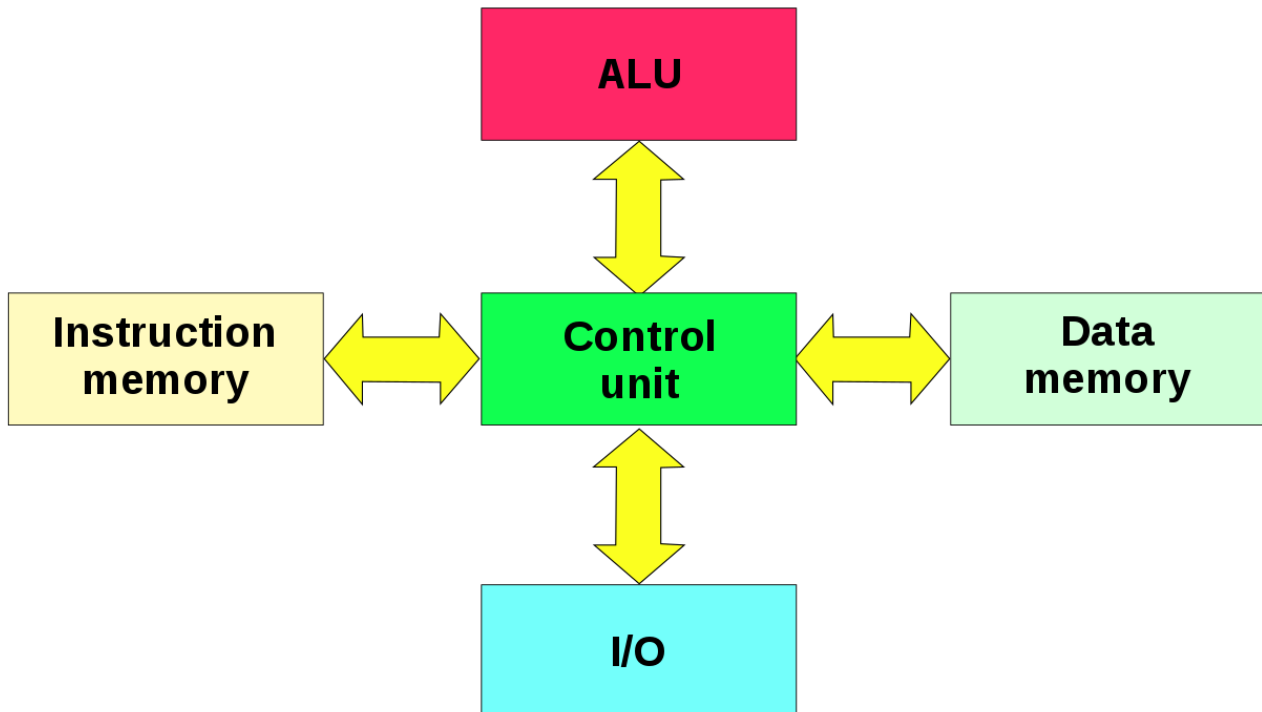


Рис. 3.1: Harvard architecture

### 3.4 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 3.2: 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;
  
```

## 3.5 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 позволяют 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 3.3: 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, 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_UINT
{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

endif /* SX1262_PARAMS_H */

```



## 3.6 config\_xxx.c/config\_xxx.h файл с конфигурацией по умолчанию\*

config\_xxx.c/config\_xxx.h файл с конфигурацией по умолчанию\* После старта питания надо как-то проинициализировать драйвер. Особенно при первом запуске, когда FlashFs/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 3.4: 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.

## 3.7 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 197х) 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, пулять пакеты в I2C, SPI, UART, MDIO и т. п.

Листинг 3.5: 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 */

```

### 3.8 xxx\_diag.c/xxx\_diag.h файлы с диагностикой\*

У каждого драйвера есть куча всяческих констант. Значения подобраны вендором обычно случайно. 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 строки. Эти Hash функции как раз вызывает CLI(шка) и компонент логирования при лог-инициализации board(ы).

Листинг 3.6: Интерпретатор констант

```

const char* DacLevel2Str(uint8_t code){
    const char *name="?";
    switch(code){
        case DAC_LEV_CTRL_INTERNALY: name="internally"; break;
        case DAC_LEV_CTRL_LOW:        name="low"; break;
        case DAC_LEV_CTRL_MEDIUM:     name="medium"; break;
        case DAC_LEV_CTRL_HIGH:       name="high"; break;
    }
    return name;
}

```

### 3.9 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.

### 3.10 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. Спекта GNU Make всего навсего это 224 страницы.

Листинг 3.7: mk файл сборки программного компонента

```

ifneq ($(SI4703_MK_INC),Y)
    SI4703_MK_INC=Y

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

```

```

INCDIR += -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:

### 3.11 xxx\_dep.h файл с проверками зависимостей

\*9—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 3.8: Проверка зависимостей препроцессором

```

ifndef DWM3000_DEPENDENCIES_H

```













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. 3.2: Содержимое папки с файлами

```
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 */

```

## 3.12 Должен быть файл 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 3.9: Преконфиг

```

define KEEPASS_COMPONENT_VERSION "1.2"

$(info AT24CXX_PRECONFIG_MK_INC=$(AT24CXX_PRECONFIG_MK_INC) )

ifneq ($(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
          xxx.gvi                                     Graphviz

\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

## 3.13 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. Она может работать как в суперцикле, так и в отдельном 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.
4. Если ваш чип с I2C, то вам очень-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.
5. 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`

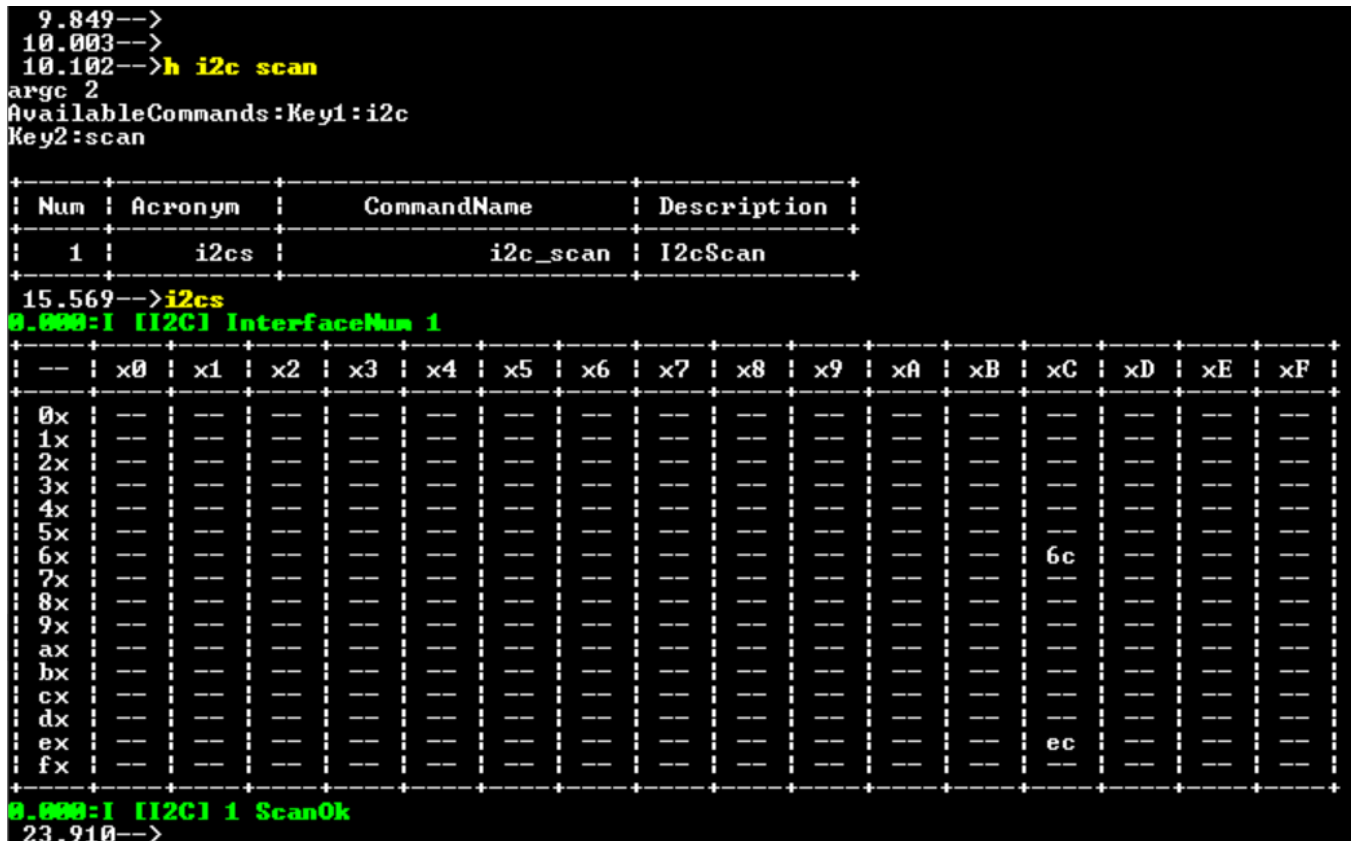


Рис. 3.3: Сканирование шины I2C

- blob(a) позволит визуально сравнить конфигурацию с тем, что прописано в datasheet(e) и понять в каком режиме чип работает прямо сейчас.
- 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 ChipID именно по этой причине, что их иначе невозможно протестировать.
  - 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).
  - \*У каждого компонента должна быть версия. Должна быть поддержка чтения версии компонента в run-time

Листинг 3.10: Версия компонента

```
define KEEPASS_COMPONENT_VERSION "1.2"
```

- (Advanced) Должна быть диагностика чипа. В идеале даже встроенный интерпретатор регистров каждого битика, который хоть что-то значит в карте регистров



```
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	RDSA
14	0x0d	0000_1101	0x0000	0000_0000_0000_0000	RDSB
15	0x0e	0000_1110	0x0000	0000_0000_0000_0000	RDSC
16	0x20	0010_0000	0x0000	0000_0000_0000_0000	RDSB

Рис. 3.4: Значения 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. Типа такой: <https://github.com/aabzel/tja1101-register-value-blob-parser> 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.

При каждом изменении в коде драйвера версию надо увеличивать.

10. \* 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 к в математике у каждой функции есть обратная функция.
11. 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.
12. \*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 посылать 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.
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.

Суммируя вышесказанное получается вот такой список необходимых файлов

№	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	<a href="#">xxx_preconfig.mk</a>	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	скрип сборки для CMake
10	9	<a href="#">xxx.mk</a>	mk	скрип 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 зафисимостей 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. 3.5: список необходимых файлов программного компонента

## 3.14 Итоги

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

Остальной код зависит уже от конкретного ASIC чипа, будь это чип управления двигателем, беспроводной трансивер, RTC или простой датчик давления. Как видите, чтобы написать адекватный драйвер чипа надо учитывать достаточно много нюансов и проделать некоторую инфраструктурную работу.

Не стесняйтесь разбивать драйвер на множество файлов. Это потом сильно поможет при custom(мизации), переносе и упаковке драйвера в разные проекты с разными ресурсами.

Если есть замечания на тему того, какими ещё атрибутами должен обладать обобщенный драйвер периферийного I2C/SPI/MDIO чипа, то пишите в комментариях.

## 3.15 Гиперссылки

- (a) Архитектура Хорошо Поддерживаемого драйвера для I2C/SPI/MDIO Чипа
- (b) Эффективное использование GNU Make

## Глава 4

# Почему важно собирать код из скриптов?

### 4.1 Пролог

В период с 199х по 202х на территории РФ развелось порядка двадцати тысяч программистов-микроконтроллеров, которые никогда в своей жизни не вылазили 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.



Рис. 4.1: boenig vs airbus

Дело в том, что GUI IDE появились в 199х...201х, когда не было расцвета DevOps(a),

программист работал один и все действия выполнялись вручную. Мышкой. В то время работа в GUI казалась программистам-микроконтроллеров веселее, ведь в IDE много страшиков.

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

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  ...

 igorp1024 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  ...

Рис. 4.2: 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?

- IDE монолитные и неделимые. Если Вы захотите поменять какую-то часть ToolChain(a): препроцессор, компилятор или компоновщик, а остальные фазы ToolChain(a) оставить как есть, то ничего из этого у вас не выйдет, так как капот IDE наглухо закрыт на замок.
- IDE стоят дорого, порядка 3500 EUR на один компьютер. Это лишняя дань и оброк для вашей компании. Оно вам надо?
- IDE(шные) xml очень слабо документированы или не документированы вовсе. У всех вендоров xml имеет свой особенный снобский xml-like язык разметки

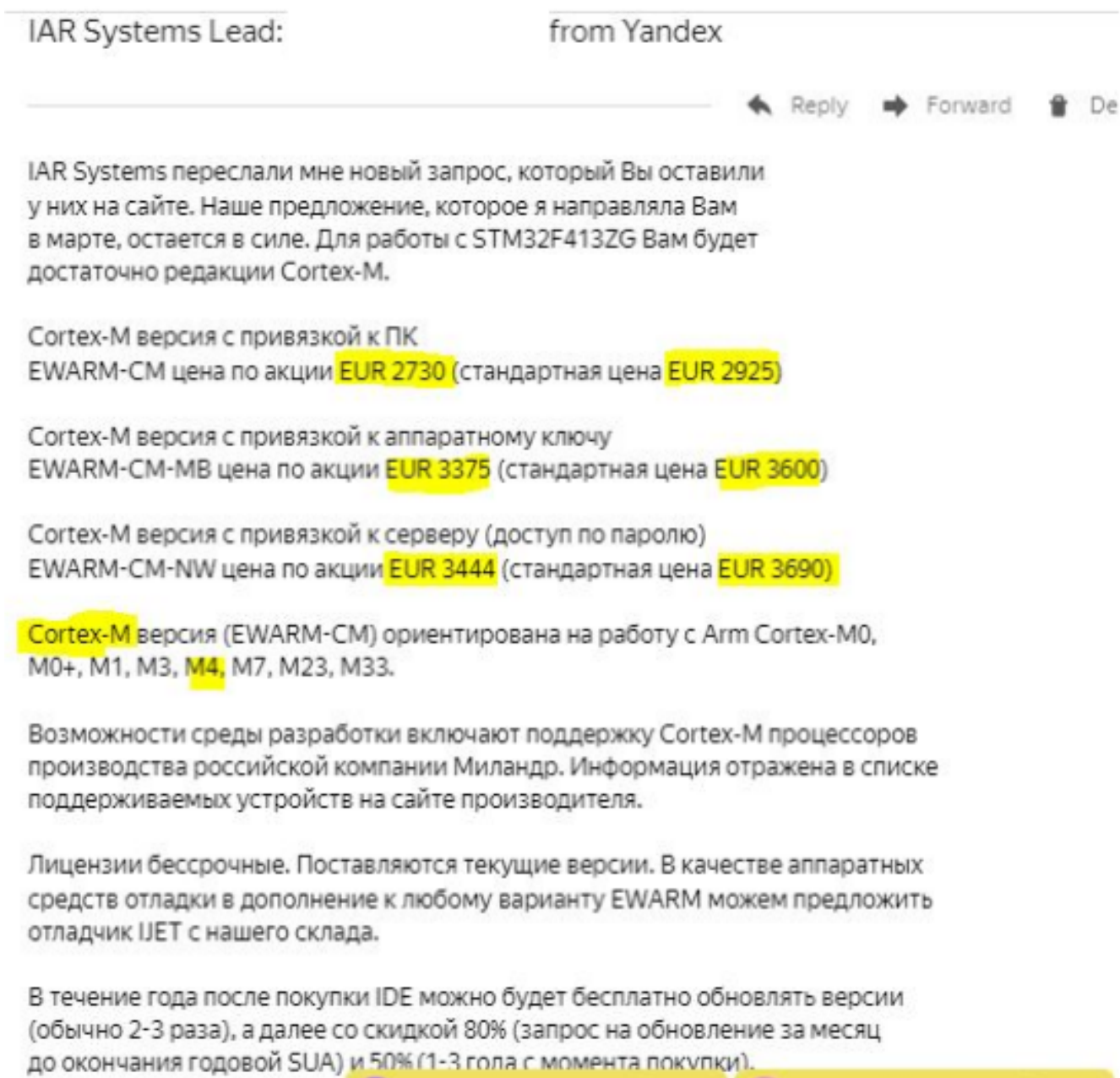


Рис. 4.3: Стоимость IAR

для конфигурации проекта. При внесении незначительных изменений появляется огромный git diff.

- (d) Затруднена сборка из консоли. В основном инициировать сборку в IDE можно мышкой или горячими клавишами. Либо надо делать refresh мышкой из-под IDE.
- (e) Обратная несовместимость с новыми версиями IDE.
- (f) 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
- (g) 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 напроць зависают, что



[illegible]

Рис. 4.4: Добавил в дерево IDE ссылку на одну папочку. Получился такой diff. Как думаете сколько часов его будут review(вить) ? Ответ: в среднем две недели

помогает лишь перезагрузка розеткой.

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 нелепы 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).

Понятное дело, что разработчики IDE во всю пользуются ситуацией и добавляют всяческие программные закладки, такие как слив 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.



Рис. 4.5: иллюстрация программирования микроконтроллеров в IDE

А когда запад нас в очередной раз кинул пришлось начать думать как теперь дальше жить... Хорошим решением оказалось сделать шаг назад в 197х 198х когда на компьютерах всё делали из консоли. Даешь сборку сорцов из скриптов! Можно вообще \*.bat файл написать и он, в общем-то, иницирует 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 197х 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 200х...202х.

Когда вы собираете из Make вы можете не только собирать исходники, но и

- (a) Что можно делать из скриптов сборки
- (b) сортировать конфиги
- (c) запускать статический анализатор
- (d) собирать документацию (вызвать Latex, Doxygen)
- (e) строить графы зависимостей на graphviz
- (f) автоматически архивировать артефакты

- (g) можете прямо из make отправлять файлы прошивок потребителям
- (h) Вызывать процедуру пере прошивки консольными утилитами программатора
- (i) автоматически генерировать функцию инициализации программы
- (j) автоматически генерировать конфигурации
- (k) автоматически обновлять версию прошивки
- (l) подписывать артефакты
- (m) автоматически выравнивать отступы в исходниках
- (n) генерировать список макросов для подсветки синтаксиса
- (o) и многое другое

Утилите make всё равно какие консольные утилиты вызывать. Понимаете? 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.

Для каждой сборки надо самим писать крохотный Makefile

Листинг 4.1: Makefile

```

MK_PATH:=$(dir $(realpath $(lastword $(MAKEFILE_LIST))))
WORKSPACE_LOC:=$(MK_PATH)../.. /

```

```

INCDIR += -I$(MK_PATH)
INCDIR += -I$(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 4.2: 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 4.3: mk файл для UWB трансивера

```
ifneq ($(DWM1000_MK_INC),Y)
    DWM1000_MK_INC=Y

    DWM1000_DIR = $(DRIVERS_DIR)/dwm1000

    $(info + DWM1000)

    INCDIR += -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
endif

```

## 4.2 Как отлаживаться

Сейчас же в 201х...202х какой-нибудь 43-летний Junior-embedded программист микроконтроллеров, привыкший к GUI-IDE, может логично провозгласить:

Я вообще не представляю, как без помощи IDE и зелёного треугольника вверху делать пошаговую отладку программы?

Тут есть 4+ ответа:

- (a) Использовать связку GDB Client + GDB Server и отлаживать код из командной строки.
- (b) Отлаживать код через интерфейс командной строки CLI поверх UART.
- (c) В каком-то веке покрывать свой код модульными тестами
- (d) 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

## 4.3 Достоинства сборки кода из Make файлов

В чем достоинства сборки C-кода из Make файлов?

- (a) Makefile это самый гибкий способ управлять модульностью кодовой базы. Можно буквально одной строчкой добавлять или исключать один конкретный программный компонент (десятки файлов) из десятков сборок. В случае же сборки из-под IDE вам бы пришлось вручную мышкой редактировать .xml для каждой отдельной сборки.
- (b) 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. Нарботка артефактов.

А сборку из Makefile очень легко автоматизировать. Достаточно в консоли выполнить `make all` и у вас иницируется 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 собрались пока я сплю.

- (c) 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`.
- (d) После сборки из скриптов вы получите полный лог сборки, в то время как IDE обычно показывают последние 3-4 экрана сообщений компилятора.
- (e) Делов том, что GNU Make скрипты пишутся только один раз. Потом только лишь так косметически меняются при добавлении очередных программных компонентов.
- (f) 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
- (g) Параллельное написание Make файлов и C-кода стимулирует придерживаться модульности, изоляции программных компонентов и к прослеживанию зависимостей между компонентами. Если вы пишете код и make файлы примерно параллельно, то очень вероятно, что у вас получится чистый, аккуратный репозиторий сам собой.
- (h) 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.
- (i) 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!
- (j) 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 `grep` в той же консоли от GIT-bash даже при работе в Windows 10.
- (k) 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 `cpr` (препроцессора).

- (l) 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. 4.6: эмулятор прошивки в консольном приложении

- (m) Внутри makefile вы можете выполнить какой-нибудь полезный скрипт. Например подписать прошивку состоянием репозитория

#### Листинг 4.4: Конфиг для сборки

```

GIT_SHA := $(shell git rev-parse --short HEAD)
OPT += -DGIT_SHA=0x0$(GIT_SHA)

```

таким образом 3мя строчками вы сделаете то, что отдельными скриптами заняло бы 50+ строк.

## Листинг 4.5: Конфиг для сборки

```
LOG_INFO(SYS, " GitSha: 0x%08x", GIT_SHA);
```

- (n) 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 мышкавозней и дублированием конфигов!



Рис. 4.7: скриптами проще добавить новую сборку в сравнении с IDE

## 4.4 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!

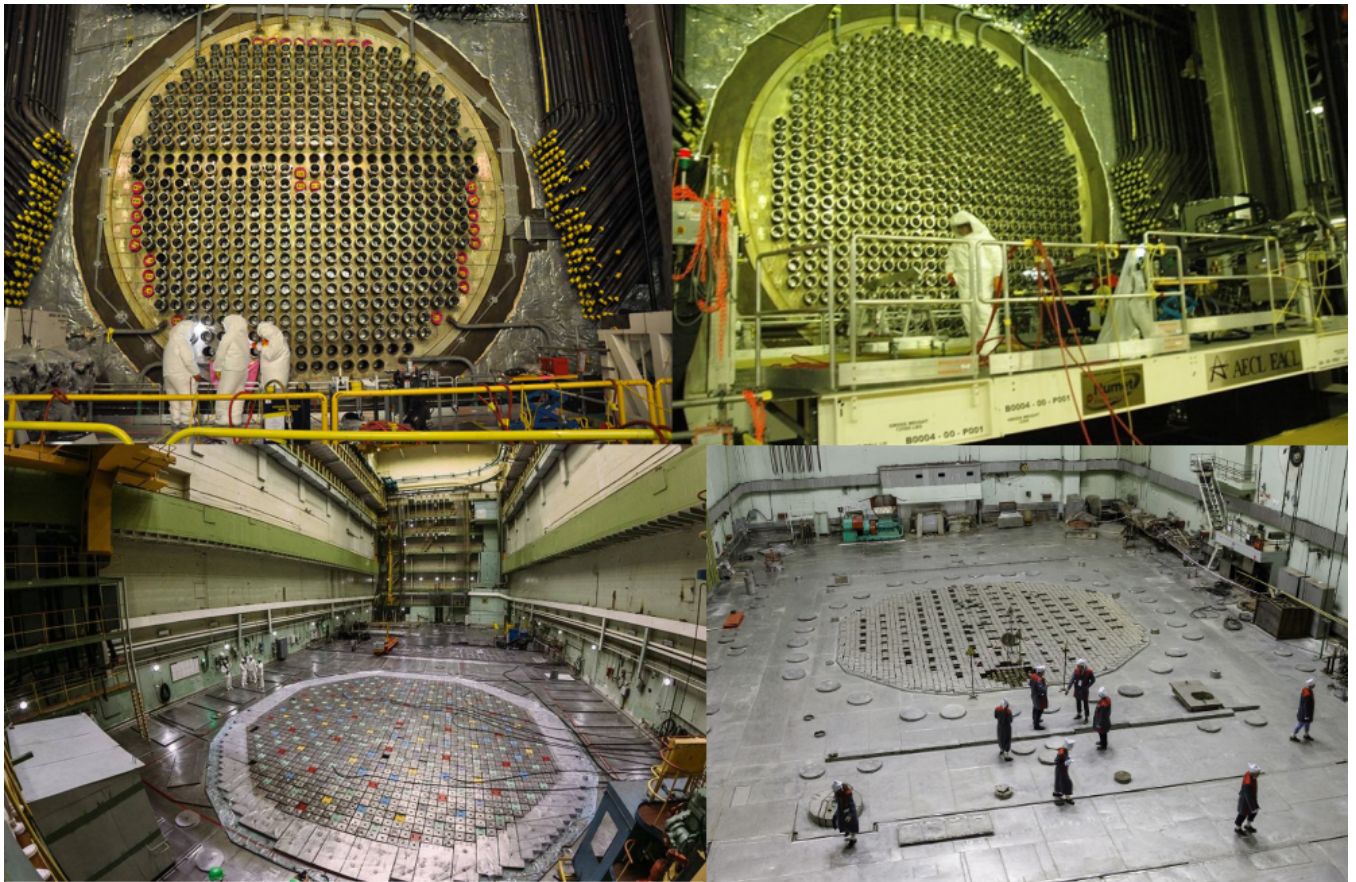


Рис. 4.8: Горизонтальные ТВЭЛы труднее загружать чем вертикальные ТВЭЛы

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.

## 4.5 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

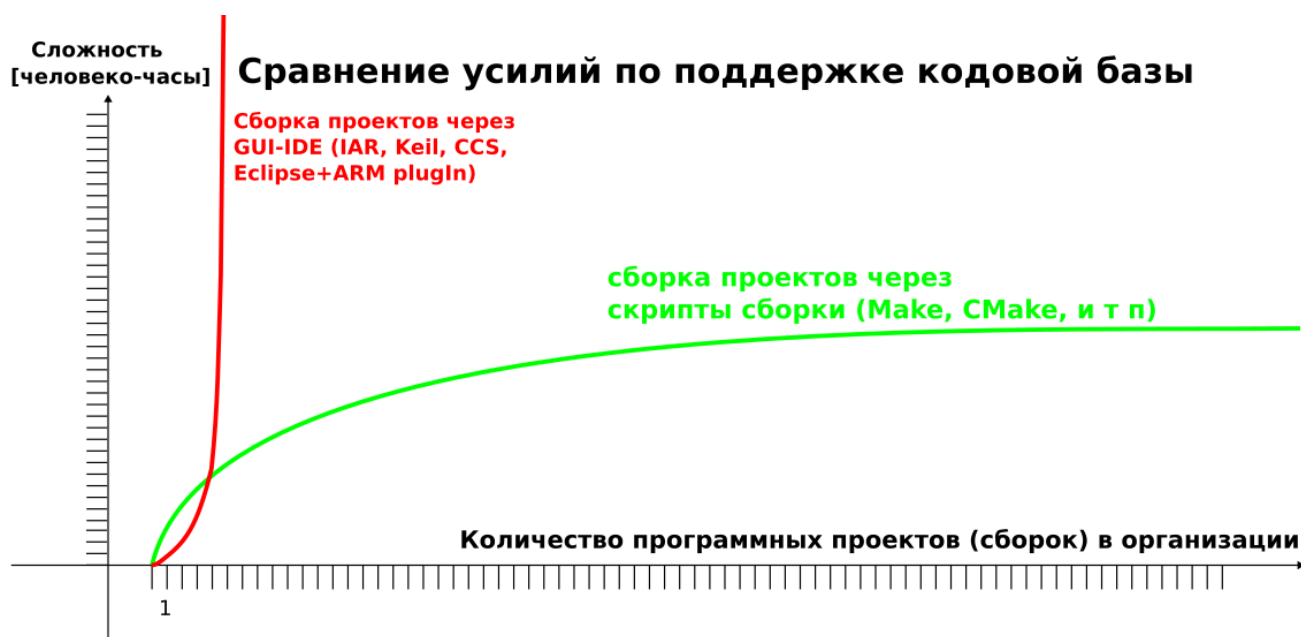


Рис. 4.9: скриптами проще добавлять новую сборку чем в IDE

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 скрипы 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 вырабатывает 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(o) 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

## 4.6 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.

## 4.7 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(y). 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 ещё 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 IT конференциях, которые поясняют откуда взялся этот график. Находятся по названиям по первым ссылкам.

(a) CI/CD прошивок для микроконтроллеров в Wiren Board ( начало на 25:20)

(b) 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 ну просто немерено. Океан информации.

## 4.8 Гиперссылки

- (a) Настройка ToolChain(a) для Win10+GCC+C+Makefile+ARM Cortex-Mx+GDB
- (b) Пример Makefile
- (c) Эффективное использование GNU Make
- (d) Обновление Прошивки из Make Скрипта
- (e) CI/CD прошивок для микроконтроллеров в Wiren Board ( начало на 25:20)
- (f) Конвеерум 30: Эволюция рабочего окружения для embedded разработки
- (g) GNU Make может больше чем ты думаешь

## Глава 5

# Литература

- (a) Эффективное использование GNU Make, Владимир Игнатов, 2000
- (b) Автоматное программирование, Поликарпова Н. И., Шалыто А. А., 2008
- (c) Программирование введение в профессию, 4 Тома, А.В. Столяров, 2016
- (d) "Как стать специалистом по встраиваемым системам" пособие для тех, кто хочет заниматься интересным и хорошо оплачиваемым делом, Левин Э.
- (e) Искусство схемотехники. Часть первая. Аналоговая Издание 3-е, часть первая, Хилл У., Хоровиц П.
- (f) Цифровая обработка сигналов. Практическое руководство для инженеров и научных работников, Стивен Смит, 2018
- (g) Ядро Cortex-M3 компании ARM Полное руководство, Joseph Yiu, 2015
- (h) Extreme C, Kamran Amini, 2019
- (i) Язык C в XXI веке, Бен Клеменс, 2015
- (j) Абстракция данных и решение задач на C++, Фрэнк М Каррано, Джанет Дж. Причард, 2003
- (k) Структуры данных и алгоритмы, Ахо, Хопкрофт, Ульман, 2021
- (l) Алгоритмы Руководство по разработке, Стивен С. Скиена, 2011
- (m) Алгоритмические трюки для программистов, Генри Уоррен-мл., 2014
- (n) Современные операционные системы, Эндрю Таненбаум, Херберт Бос, 2016
- (o) Архитектура встраиваемых систем, Даниэле Лакамера, 2023