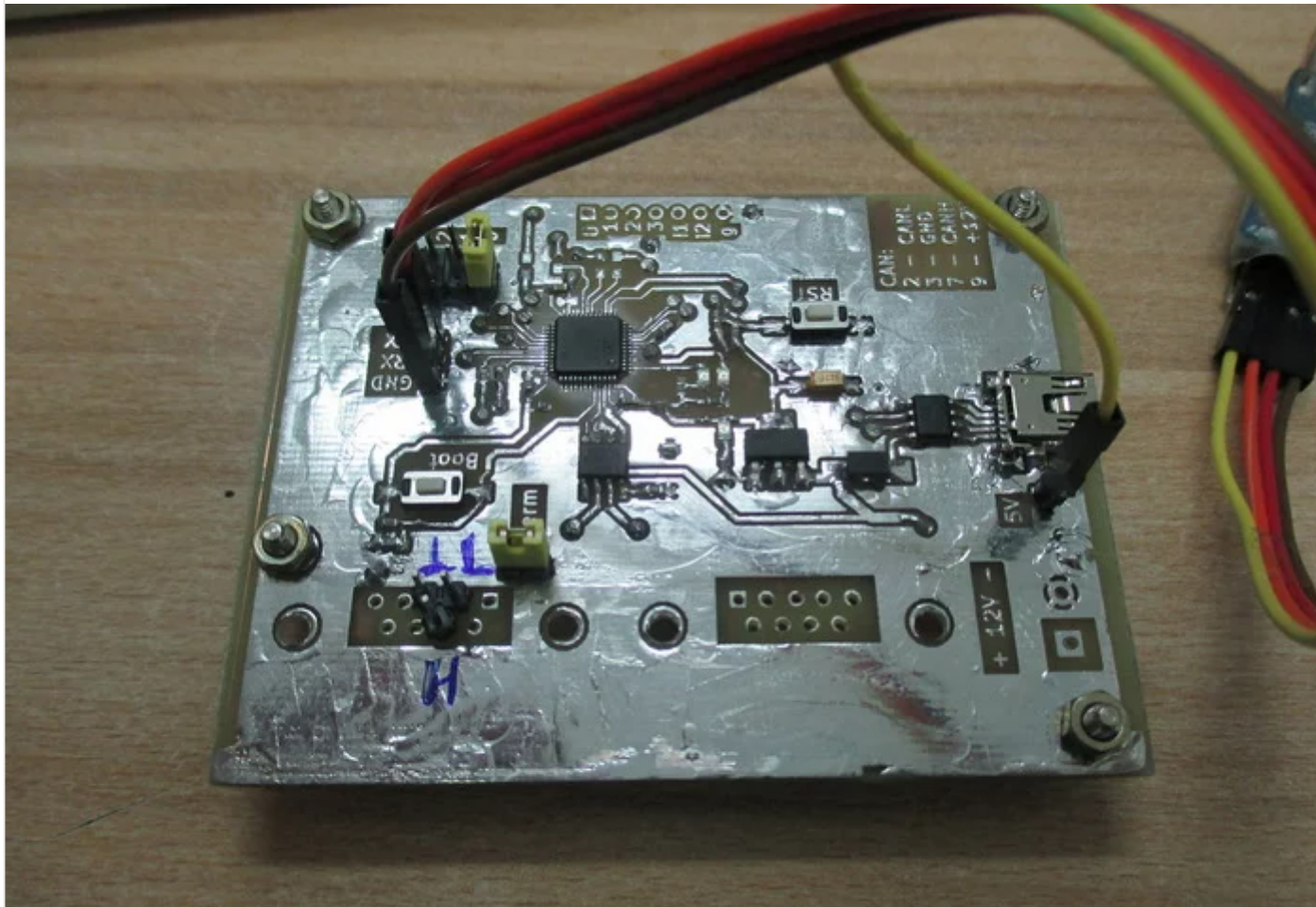
 madlinuxoid 1 год назад  Лига Радиолюбителей

CAN-сниффер на STM32F0x2 [↗](#)



Горячее Лучшее Свежее Подписки Сообщества

1

(заранее извиняюсь за отсутствие кусков кода в тексте: редактор пикабу не позволяет воткнуть достаточно длинные посты, да и нет никаких тегов для оформления исходников, оригинал можно прочитать у меня в ЖЖ; и вообще, на пикабу крайне неудобный редактор постов, в ЖЖ все намного удобней).

Саму основу для сниффера я сделал еще давно — когда необходимо было разработать девборду для "тренировок" с USB и CAN (под контроллеры термодатчиков):

Протокол работы железки я уже описал, теперь опишу исходники ее прошивки.

Итак, первое, что нам нужно для работающего CAN-сниффера — это возможность одновременной работы и CAN, и USB. В дешевой нише STM32 это умеют STM32F0x2 (072 и 042; на работе для термодатчиков я закупал 042, для дома же купил на али 2 десятка 072, тогда это было примерно 60 рублей за штучку).

В отличие от 103-х, где служебный буфер в памяти CAN и USB делить не "по-братски", и он может принадлежать лишь одной из периферий, в 0x2 CAN забирает лишь 256 последних байт буфера USB. Поэтому для USB доступно 768 байт буфера (в принципе, этого вполне достаточно, если не делать сложное составное устройство).

Начнем с USB. Сниффер будет "выдавать" себя за PL2303 (мне нравится, что модуль ядра выделяет для него устройство /dev/ttyUSBx, а не позорный /dev/ttyACMx, как под обычный USB-CDC; кроме того, в некоторых некошерных дистрибутивах вроде бубунты могут быть проблемы с USB-CDC: при подключении запускается modemd и захватывает файл устройства в свое личное распоряжение).

В заголовочном файле `usb_defs.h` определяем `USB_BTABLE_SIZE` как 768 байт. Там же определяем размеры буферов конечных точек 0 и 1 (конечная точка 1 — interrupt IN — использоваться при работе не будет, и в принципе можно было бы ее не определять). Там же нам понадобится определить структуры служебных регистров и регистров описания конечных точек. В файлах `usb_lib.c` и `usb_lib.h` разместим "низкоуровневые" функции USB. В принципе, эту иерархию не я придумал: когда я только начал заниматься USB, вменяемой реализации на просторах интернета не нашел. Один из пользователей easyelectronix выложил простую реализацию USB-HID. Собственно, на основе ее я и сделал свои экземпляры USB-HID, CDC и эмуляцию PL2303.

Авторизация

[Забыли пароль?](#)

ВОЙТИ

РЕГИСТРАЦИЯ

или



Рассылка Пикабу: отправляем лучшие посты за неделю 🔥

Подписаться

ДОБАВИТЬ ПОСТ

О сообществе

сообщества



Лига Радилюбителей

[Горячее](#) [Лучшее](#) [Свежее](#) [Подписки](#) [Сообщества](#)

или исходящим и есть ли там данные SETUP. Для работы с регистрами конечных точек нам понадобится определить пару макросов: KEEP_DTOG_STAT и KEEP_DTOG.

Здесь сразу скажу: надо помнить, что в регистрах EPnR некоторые флаги имеют свойство toggle. Поэтому не повторяйте моих ошибок: держите это всегда в уме, и когда нужно лишь какой-то флаг установить/сбросить, обнуляйте биты всех ненужных toggle-флагов! Эти макросы собственно и занимаются тем, что оставляют нетронутыми флаги DTOG (они нам не нужны, т.к. мы не пользуемся двойной буферизацией) и STAT (а это важно, когда мы получаем данные, но не хотим сразу же отправлять ACK, пока буфер не будет обработан). Еще для работы конечного автомата состояния USB понадобится определить его состояния. Там же определяем макросы для задания строковой информации (_USB_STRING_, _USB_LANG_ID_) и вспомогательные структуры для разбора конфигурационного пакета (config_pack_t), настройки конечной точки (ep_t) самого USB(usb_dev_t — эта структура используется, чтобы поменьше глобальных переменных заводить). Из /usr/include/linux/usb/cdc.h копируем определение структуры usb_LineCoding (в принципе, можно обрабатывать SET_LINE_CODING, меняя скорость, скажем, USART1; но в данном случае это не нужно, а USART1 у меня использовался исключительно для отладочных сообщений).

В файле usb_lib.c определяем дескрипторы устройства (достать их несложно, если "натравить" на "настоящий китайский" PL2303 утилиту lsusb -v). Здесь же как WEAK определены заглушки для обработчиков стандартных запросов SET_LINE_CODING (изменение параметров последовательного порта: скорости, четности и т.п.), SET_CONTROL_LINE_STATE (аппаратное управление потоком) и SEND_BREAK (конец связи).

В отличие от "обычного" USB CDC у pl2303 есть еще и vendor-запросы. Нафиг они нужны — непонятно, однако, благодаря тому, что кто-то уже отреверсил подобную железяку и написал для нее модуль ядра, в исходниках /usr/src/linux/drivers/usb/serial/pl2303.c можно посмотреть, как оно работает. Собственно, оттуда я и утащил функцию-обработчик vendor_handler(config_pack_t *packet).

Возможно, в оригинале эти запросы таили что-то эдакое, но в ядре они используются лишь для идентификации — что на том конце действительно PL2303. Во всяком случае, с таким минимумом устройство работает и под android (возможно, будет работать и под игровыми

Сообщество для радиолюбителей и людей, которым интересна подобная тематика)

Правила сообщества ▾

Управление сообществом



Beteljuice Администратор



akosh9206 Модератор

Комментарий дня

ТОП 50

Чо кактам пацаны? торвпндаркнет? инкогнита?

Нормальные преступники через пикабу переписываются. Два куба дров кленовых нужно к выходным.

+1112  CatcherRye 17 часов ...

Объявление скрыто **CRITEO**

Пожаловаться
на объявление

При работе со строковыми дескрипторами иногда может оказаться, что дескриптор не влезает в стандартный объем 64-байтной посылки (особенно большими размерами славятся HID-дескрипторы), поэтому такие вещи надо разбить на несколько посылок. Отправляются такие дескрипторы только в начале коннекта, поэтому я решил не заморачиваться с конечным автоматом, а сделать блокирующую запись: `static void wr0(const uint8_t *buf, uint16_t size)`. Здесь еще надо было учесть, что если мы отправляем в последней посылке ровно 64 байта, то нужно еще и ZPL — посылку нулевой длины — отправить.

Работать с USB мы будем на прерываниях, поэтому все самое интересное начинается в обработчике прерывания `usb_isr()`.

Здесь мы обрабатываем такие прерывания, как RESET — хост говорит устройству, что нужно заново инициализировать USB, CTR (correct transfer) — прерывание по приему или передаче данных, а также вспомогательные SUSP и WKUP — "засни" и "проснись".

В обработке OUT-запросов (т.е. входящих для устройства) пришлось пойти на хитрость: т.к. некоторые вещи (как тот же LINECODING) передаются в "два захода", необходимо отдельно заполнять данными `setup_packet` и вспомогательный `ep0databuf` (где и лежат эти данные по установке LINECODING). Данные для LINECODING приходят так: сначала без флага SETUP приходит нужная информация, а потом уже с этим флагом — команда запроса SET_LINECODING. И, соответственно, вызывается процедура обработчика запроса.

Чтобы USB корректно работало, сначала нам надо настроить все конечные точки: выдать им адреса и размеры буферов, определить направление, задать функцию-обработчик. Это делается в `EP_Init`.

Кому-то нравится руками задавать адреса буферов данных, я же решил все упростить — в переменной `lastaddr` хранится последний свободный адрес в буфере (при RESET эта переменная реиницируется на начало буфера). Дальше все по коду понятно.

Для разбора данных, поступающих на управляющую точку EP0, вызывается функция `EP0_Handler`. Из остальных конечных точек, как я уже говорил, точка EP1 у нас хоть и определена, но не используется. А для приема/передачи заводим односторонние точки EP2 и EP3 (все эти данные хранятся в дескрипторе устройства, поэтому размеры буферов и направление передачи конечных точек нужно согласовывать с данными в дескрипторе). Их обработчики будут уже в

Рекомендуемое сообщество



Всё о часах

216 постов • 1 516 подписчик...

ПОДПИСАТЬСЯ



Часы и все, что с ними связано - выбор, обслуживание, ремонт, аксессуары.

У STM32F0x2 регистры данных USB_TX пишутся по 16 бит (в STM32F103 эмулируется 32-битное хранилище, т.е. писать надо как 32-битный блок, но активны там только 16 бит), а USB_RX вообще можно читать побайтно (у 103 они тоже эмулируют 32-битные блоки). В общем, здесь все проще. EP_WriteIRQ отличается от EP_Write тем, что вызывается внутри обработчиков прерывания, поэтому в ней флаги EPnR не меняются.

В файле `usb.c` лежат сравнительно высокоуровневые функции (правда, я и `USB_setup` зачем-то здесь оставил).

Собственно, настройка USB и начинается с `USB_setup`. Тактируем USB от HSI48, что позволяет не цеплять внешний кварц. Используем автокоррекцию HSI48 от USB SOF. Для начала разрешаем только прерывания RESET и WKUP (остальное будем разрешать уже после настройки конечных точек).

IN/OUT запросы данных обрабатываем функциями `transmit_Handler()` и `receive_Handler()`. Обработчик IN (выходящие данные) очищает флаг `CTR_TX` и выставляет внутренний флаг `tx_succesfull` (говорящий о том, что предыдущая посылка отправлена и можно отправлять следующую). А в обработчике OUT (входящие данные) мы выставляем внутренний флаг `rxNE` (говорящий, что в буфере есть данные и их можно считать) и очищаем флаг `CTR_RX`. Выставлять ACK мы будем лишь после того, как данные из буфера будут обработаны!

Для того, чтобы не блокировать МК на время отправки мелких объемов данных (а в основном они значительно меньше 64 байт), в функции `USB_send` есть возможность "отложенной" отправки данных (последняя строчка просто помещает маленькие объемы данных во вспомогательный буфер, который будет отправлен в последующем при помощи функции `send_next()`).

Из основного цикла `main` на каждом проходе надо запускать функцию `usb_proc`. Здесь анализируются состояния КА USB. Скажем, в состоянии `USB_STATE_CONFIGURED` (EP0 настроена, все дескрипторы отправлены) нужно настроить остальные конечные точки. В состояниях `USB_STATE_DEFAULT` (начальное) и `USB_STATE_ADDRESSED` (только прошла процедура адресации) USB еще пользоваться нельзя — поэтому снимаем флаг `usbON`. А в состоянии `USB_STATE_CONNECTED` вызываем ту самую `send_next()`.

Более высокоуровневую функцию `USB_receive` вызываем уже откуда-нибудь извне. Она возвращает количество считанных данных и заполняет ими буфер. Ну, а т.к. данные теперь надежно сохранены, можно отправлять хосту ACK.

**Пикабу в Telegram**

235K подписчиков

@pikabu

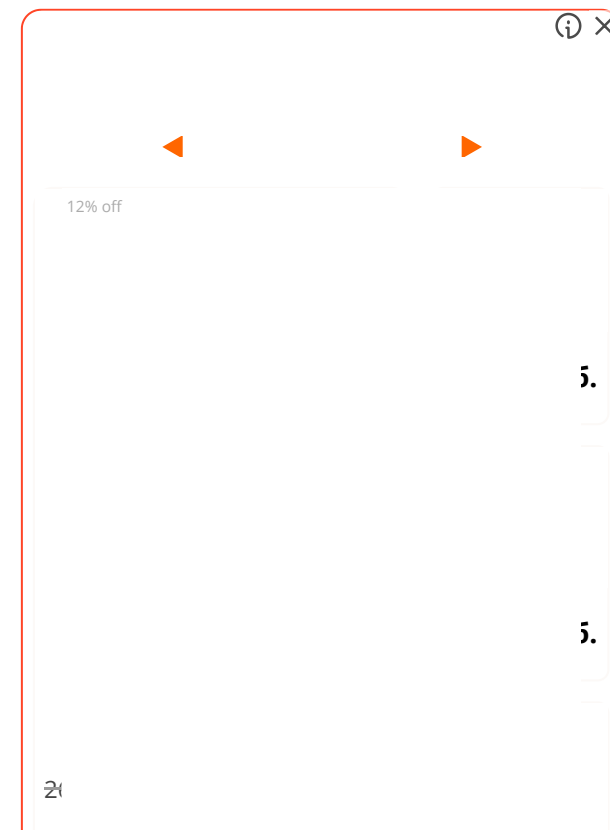
**Развлекательный канал**

45K подписчиков

@pikabu_fun

**Пикабу в Viber**

364K подписчиков

Вступить

Уже в `main.c` размещаем функцию буферизованного чтения из USB с выставлением флага готовности по `'\n'`. Здесь происходит эхо введенных символов и обработка `backspace` (чтобы можно было удалять неправильно набранные символы).

OK, с USB закончили. Пора переходить к CAN.

В `can.h` определим структуру `CAN_message`. В ней содержатся данные (до 8 байт) для стандартной CAN-посылки, длина данных и идентификатор получателя этих данных. И определим флаги состояния CAN.

В `can.c` определяем основные функции для работы с CAN. Входящие данные буферизуются в массиве `messages[CAN_INMESSAGE_SIZE]`.

Настройку CAN делаем по сниппетам. Разве что аргументом этой функции является скорость в кбод, поэтому проверяем в начале, допустимым ли является значение скорости.

Поначалу фильтр позволяет принимать абсолютно все сообщения: нечетные в `FIFO0` и четные в `FIFO1`. Далее фильтры можно будет перенастроить.

В отличие от USB, у CAN в прерывании анализируются лишь ошибки, а вот для работы с данными из `main` постоянно запускается `can_proc()`. Здесь рассматриваются различные флаги и обрабатываются ошибки линии: если на шине нет никого, либо накапливается много ошибок, CAN переинициализируется.

Функция `can_send(uint8_t *msg, uint8_t len, uint16_t target_id)` отправляет сообщение `msg` длиной `len` с идентификатором `target_id`. В принципе, можно было бы уменьшить количество аргументов этой функции, если поместить эти данные в обертку — структуру `CAN_message`. Функция находит первый свободный "почтовый ящик", заполняет в нем регистры данных и инициализирует посылку.

Функция `can_process_fifo(uint8_t fifo_num)` запускается при наличии данных в соответствующем буфере `FIFO`. Все данные помещаются в массив-буфер, откуда впоследствии их можно считать. Если буфер полон, то функция "надеется", что к следующему запуску можно будет его опустошить (иначе происходит переполнение `FIFO` — ничего с этим не поделать).

Как все это работает.

21 893,63 руб.

7 497,82 руб.

271,42 руб.

[О компании](#)

Активные сообщества ?

все



Чёрный юмор



Фабрика Мемов ↑1



Видеохостинг на Пикабу ↑1



Специфический юмор ↓1



Реальные истории из жизни



Домашняя.Ламповая.Наша.



Комиксы 18+



Авторские медицинские посты



Всё о кино



Лига историков

[Создать сообщество](#)

Тенденции

теги

Сообщества Пикабу 12

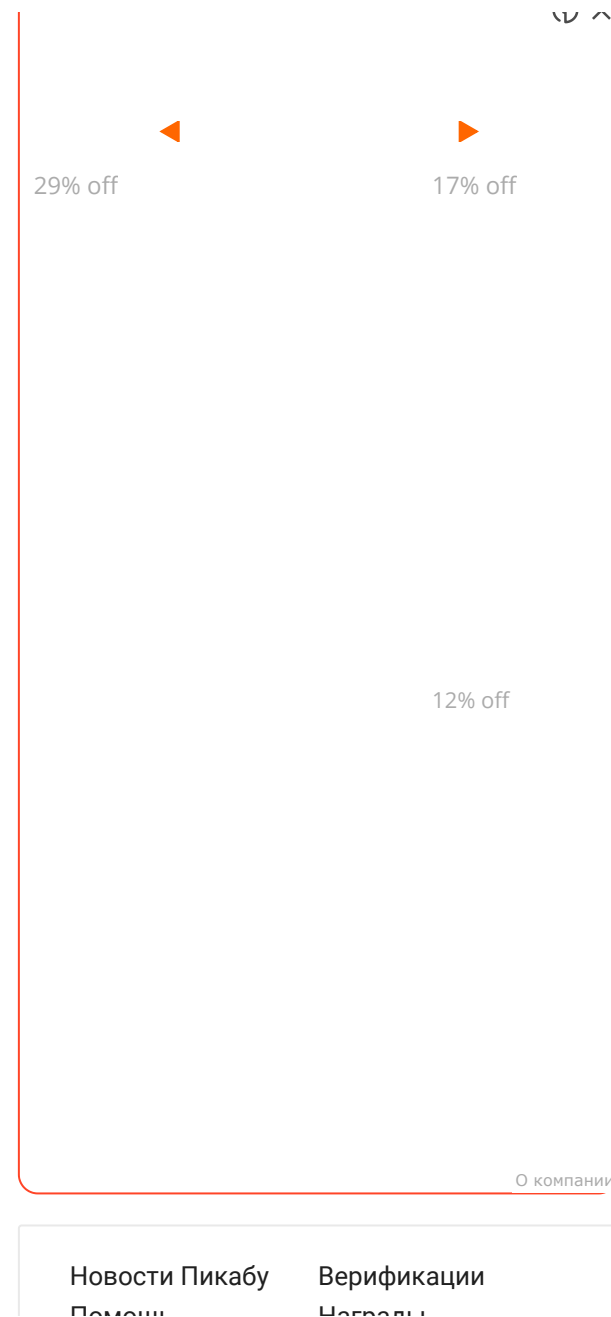
При получении определенной команды по USB, она анализируется (`proto.c`), и если это команда на передачу данных (`s`), запускается функция `sendCANcommand`. В ней происходит парсинг введенных пользователем данных, и если все ОК, то отправляется сообщение.

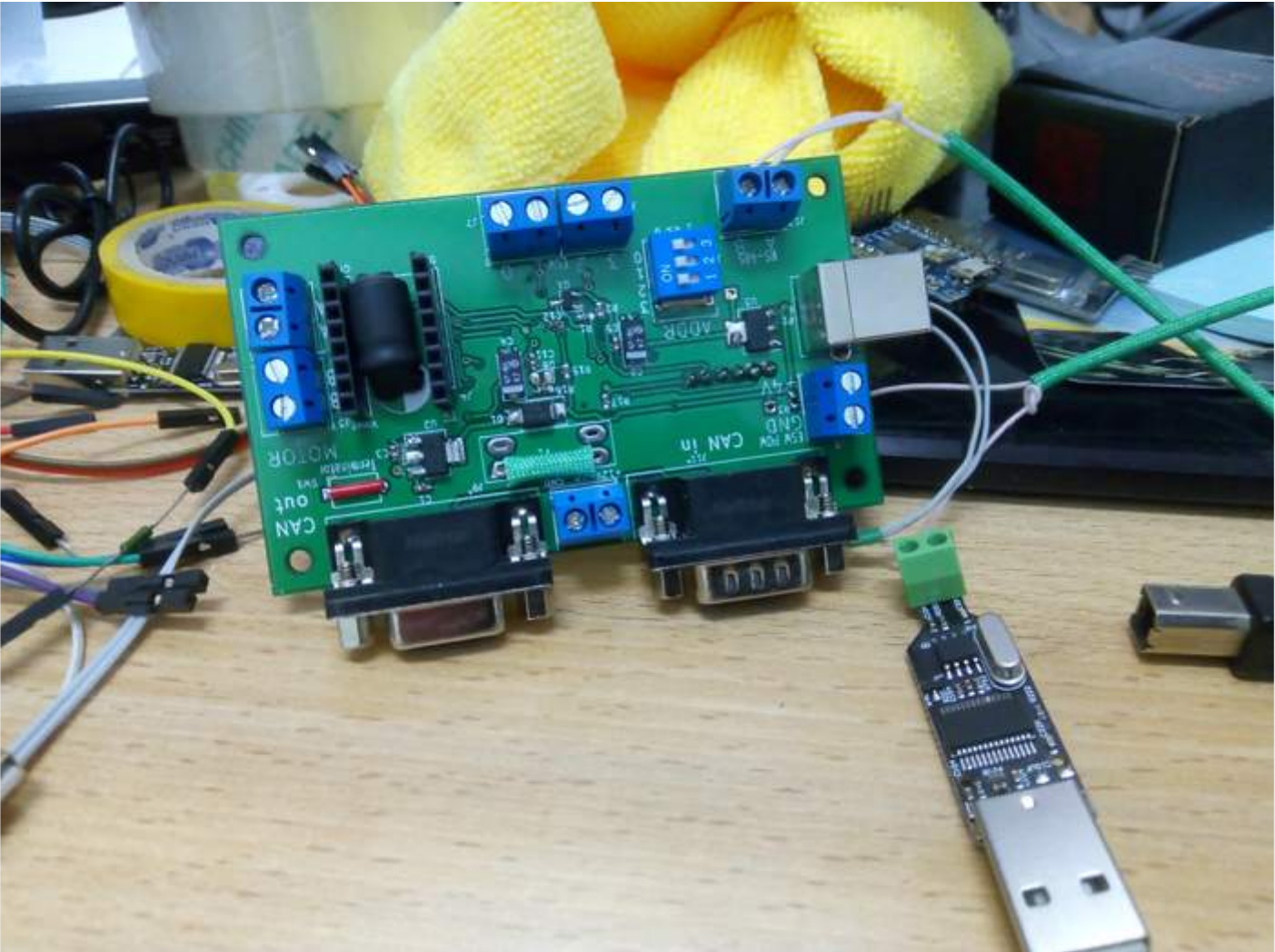
При поступлении сообщения с CAN, его содержимое выводится на терминал.

Для упрощения создания фильтров я добавил еще и софтовый фильтр — для возможности отклонения сообщений с ID из списка (иначе пришлось бы лепить аппаратный фильтр с нужной маской и ID, а считать-то лень!).

Функция `add_filter` позволяет добавить или удалить (если фильтр не содержит данных) фильтр с номером от 0 до 27. Т.е. можно удалить фильтры по умолчанию и заменить их своими. В функции идет довольно-таки длинный парсинг текстовых данных: разбирается, в каком режиме фильтр (список или маска), а далее заполняются сами данные фильтра.

При разработке сниффера я тестировал его на платах `термодатчиков` и разрабатываемой управлялки шаговыми двигателями.





[моё] Stm32 USB Длиннопост

29 16K

Эмоции

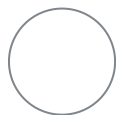
Промокоды	Магазин
Android	iOS



Модуль
беспроводной...

271,42 руб

~~330,65 руб~~



Microphse FPGA
плата с ядром...

8 247,60 руб



TLSR8266 Тесты
доска Bluetooth...

397,38 руб



Макетная
Lichee Pi Z

547,08 руб

~~576,15 руб~~

Aliexp



Лига Радиолюбителей

523 поста 6K подписчика

ДОБАВИТЬ ПОСТ

ПОДПИСАТЬСЯ



Правила сообщества

Соблюдайте правила Пикабу. Посты выкладывать лишь касаемо нашей тематики. Приветствуется грамотное изложение. Старайтесь не использовать мат.

Постарайтесь не быть снобами в отношении новичков. Все мы когда-то ничего не знали и ничего не

[Подробнее](#) ✓

Горячее

Лучшее



Свежее

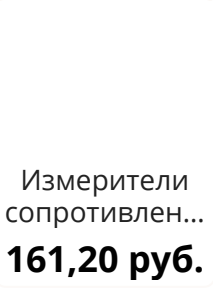
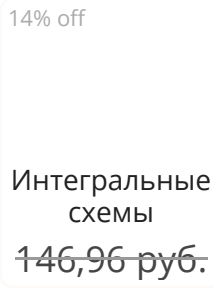
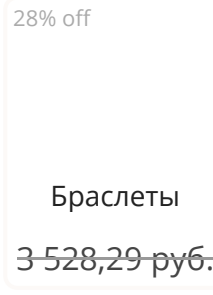
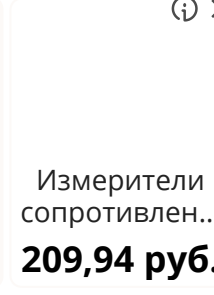

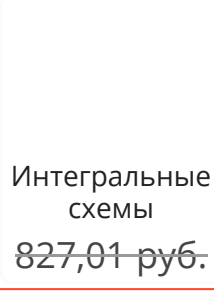

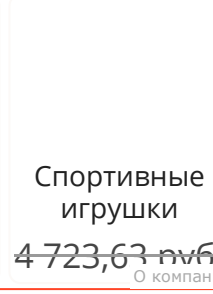
Подписки

Сообщества

[РАСКРЫТЬ 29 КОММЕНТАРИЕВ](#)

Чтобы оставить комментарий, необходимо [зарегистрироваться](#) или [войти](#)



 Измерители сопротивлен... 161,20 руб.	 Интегральные схемы 146,96 руб. 14% off	 Браслеты 3 528,29 руб. 28% off	 Измерители сопротивлен... 209,94 руб.
 Двигатели постоянного... 5 464,41 руб.	 Интегральные схемы 827,01 руб.	 Измерители сопротивлен... 121,46 руб.	 Спортивные игрушки 4 723,63 руб.

О компании

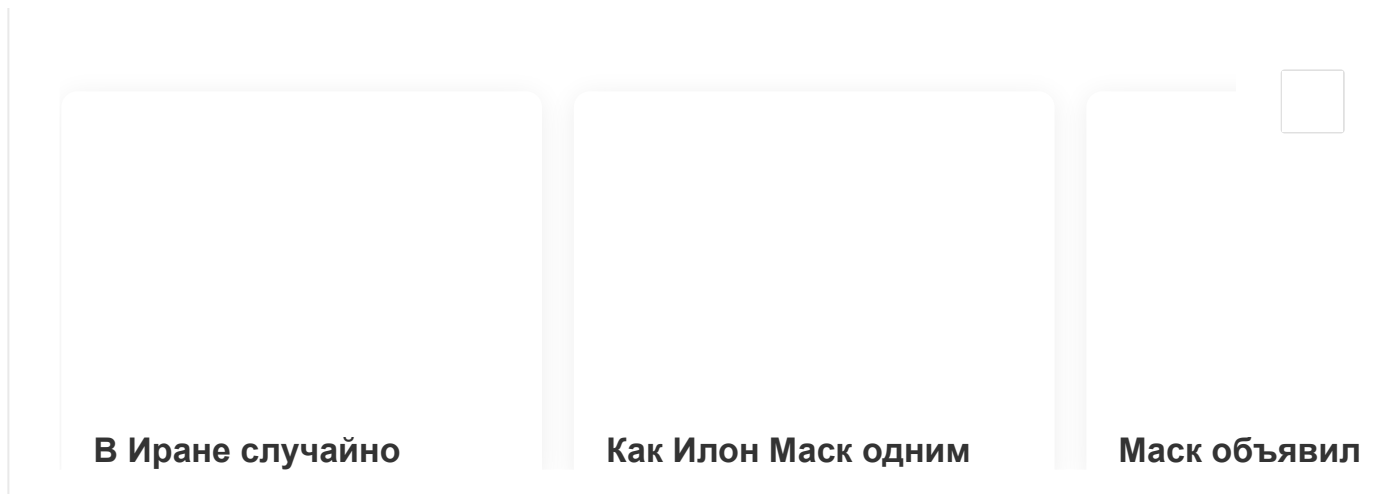
ПОХОЖИЕ ПОСТЫ

Похожие посты не найдены. Возможно, вас заинтересуют другие посты по тегам:

[Stm32](#) [USB](#) [Длиннопост](#)

Пультс

[Горячее](#) [Лучшее](#) [Свежее](#) [Подписки](#) [Сообщества](#)



Рассылка Пикабу: отправляем лучшие посты за неделю 🔥

Укажите E-mail

Подписаться

О ПИКАБУ

[О проекте](#)
[Контакты](#)
[Реклама](#)
[Сообщить об ошибке](#)
[Предложения по Пикабу](#)
[Новости Пикабу](#)
[Магазин](#)

ИНФОРМАЦИЯ

[Помощь](#)
[Правила](#)
[Награды](#)
[Верификации](#)
[Бан-лист](#)
[RSS](#)
[Конфиденциальность](#)

MOBILE

[Android](#)
[iOS](#)

ПАРТНЁРЫ

[Fornex.com](#)

[Промокоды](#)

защита от спама reCAPTCHA
Конфиденциальность - Условия
использования

