REx OS

Realtime Exokernel OS

v. 0.4.4

Оглавление

1. Введение

На данный момент существует огромнейшее множество операционных систем реального времени для микроконтроллеров. Большинство из них являются наследниками `Project TRON`, практически не привнося ничего нового, а именно:

- микроядро со стандартным набором, включающим переключение контекста и объектами синхронизации: мьютекс, событие, очередь сообщений, реже семафор.
- системный таймер на базе тиков, который заставляет просыпаться микроконтроллер несколько сот раз (а иногда и тысяч, если требуется б*о*льшая отзывчивость) в секунду для выполнения задач ядра.
- абстрактную архитектуру, не использующую возможностей современных 32-битных микроконтроллеров, а нередко и вообще выполняющихся в пользовательском контексте.

Все вышеописанное заставило полностью пересмотреть все базовые принципы создания ОСРВ и разработать принципиально новую архитектуру.

2. Описание возможностей

- Экзоядро. На данный момент это единственная в мире экзоядерная ОСРВ уровня промышленной эксплуатации. Экзоядро позволяет создавать процессы со значительной экономией программного кода, что критично для микроконтроллеров.

- Возможность независимой компиляции ядра и процессов. В случае компиляции переносимого процесса (`relocated process`) возможен запуск процессов непосредственно во время работы ОСРВ, в том числе из RAM или внешнего носителя.
- LGPL лицензия. Все изменения, вносимые в ядро, и/или драйверы/стеки, входящие в состав REx, открыты. Все проприетарные решения могут быть скомпилированы отдельным процессом без необходимости открытия кода. Также возможна коммерческая лицензия.
- Безтиковая система. Нет необходимости будить процессор постоянно. Только по действительному событию с высокой точностью до теоретической в 1us. Также это значительно снижает энергопотребление и загрузку процессора.
- Небольшой размер ядра. Со значительно лучшими характеристиками, чем у микроядерных систем. Достигается, благодаря использованию отдельного уровня библиотек.
- Система и драйверы в пользовательском контексте. В целом это стандартная микроядерная модель. Однако, благодаря отдельному уровню библиотек, код изолирован и использует

значительно меньше места.

- Объекты синхронизации. Наиболее востребованным объектом синхронизации остается IPC, максимально адаптированный под 32-битную архитектуру.
    - IPC — очередь сообщений
    - Stream — байтовый поток данных;
    - IO — блок данных.
- Программные таймеры. Используют в качестве аппаратного таймеры ядра и позволяют сэкономить на аппаратных таймерах микроконтроллера.
- Менеджер динамической памяти. Реализованный на уровне библиотек, позволяет выдать каждому процессу собственный менеджер памяти, без необходимости обращения к ядру.
- Максимальная адаптация к железу. Все системные вызовы реализованы через `swi/svc`. Переключение контекста происходит на уровне прерывания `PendSV (cortex-m)`. Все объекты организованы таким образом, чтобы максимально соответствовать требованиям MPU.
- Аппаратная абстракция. Реализована через правила построения пользовательских хэндлов с выделением классов аппаратной абстракции. Использование общих правил работы с классом устройства. Что, естественно, не ограничивает обработку специфичных для конкретного железа запросов.
- Системные библиотеки. Уровень библиотек содержит необходимую коллекцию функций, которые одновременно используются ядром и множеством процессов, без необходимости дублирования:
    - `uStdio: printf, sprintf`. Весь размер библиотеки порядка 2 килобайта.
    - `uStdlib: malloc, free, realloc`.
    - Функции для работы с временем.
    - Динамические массивы. Позволяют реализовать сложные динамические структуры, где это необходимо. Особенно — в стеках протоколов.
    - `SO` — библиотека однотипных объектов. Используется для минимизации дефрагментации оперативной памяти при использовании динамически выделяемых однотипных объектов.
- Прикладные библиотеки. Могут использоваться приложениями для встраивания в свое адресное пространство.
    - Односвязные списки, двусвязные списки, кольцевые буферы.
    - Программная генерация случайных чисел.
    - `Libusb.` Работа с `USB` дескрипторами. Позволяет генерировать дескрипторы.
    - `GPIO bitbang`. Стандартизированный и доступный для пользователя набор библиотек, абстрагированный от железа для работы с `GPIO`. Позволяет просто и без

привязки к конкретному железу реализовать модули интерфейсов, работающих программно поверх GPIO.

- GUI. Базовый набор библиотек для работы с графикой: холст, работа со шрифтами (включая поддержку UTF-8), отрисовка графических примитивов.

- USB Device. Полная поддержка стека устройств USB 2.0, включая управление энергопотреблением, high speed, композитных устройств, vendor specific request. Поддержка классов устройств с возможностью автоконфигурирования по информации из дескрипторов. На базе стека USB device реализованы следующие классы устройств:
  - CDC ACM
  - HID Boot Keyboard
  - CCID
  - MSC. Поддержака нескольких хранилищ, реализация DVD-ROM интерфейса.
  - RNDIS
- SCSI Поддержка основных команд (SPC5, SPC3, MMC6)
  - TCP/IP. Поддержка IP фрагментации, а также:
  - MAC level 802.3
  - RFC768, RFC791, RFC792, RFC793, RFC826
- Криптография: AES, AES-CBC, SHA1, SHA256, HMAC-SHA1, HMAC-SHA256
- Возможности, которые находятся в стадии бета тестирования:
  - HTTP Server
  - TLS 1.2 Server
- REx предоставляет огромные возможности для отладки и обработки ошибок, в их числе:
  - Собственные обработчики ошибок в каждом процессе. Ошибка может быть установлена как ядром, так и неудачным вызовом IPC.
  - Минидампы при критических ошибках уровня ядра. При необходимости система может быть перезапущена.
  - Обработка системных исключений, включая hard fault. В зависимости от степени ошибки, рабочий процесс может быть перезапущен, либо ошибка поднята до уровня критической. Система распознает адрес источника ошибки, что позволяет быстро ее исправить.
  - Маркировка системных объектов. Каждый создаваемый системный объект может быть промаркирован для предотвращения случаев использования неинициализированных хэндлов.
  - Проверка адресов пользовательских и системных объектов за выход адресного

диапазона, доступного процессу/ядру.

- Профилирование процессов. Позволяет проводить статистику загрузки процессов и микроконтроллера в целом, включая: время работы, использование стека, включая максимальное и текущее состояние динамической памяти. Статистика приводится в привязке к имени процесса.
- Профилирование динамической памяти. Проверка выхода переменных за область выделенной памяти как снизу, так и сверху. Проверка наличия конфликтов между стеком и динамической памятью. Статистика использования выделенных и свободных слотов, включая степень фрагментации. Возможность внутренней проверки динамического пула.
- Любая из вышеописанных опций отладки может быть выключена для экономии размера кода.

- Поддерживаемая архитектура устройств:

- `cortex-m0/m0+`
- `cortex-m3`
- `cortex-m4`
- `ARM7`

- Доступные драйверы.

- База. `GPIO, UART, TIMER, POWER: STM32F1, STM32F2, STM32F4, STM32L0, LPC11Uxx, LPC18xx`
- `RTC: STM32F1, STM32F2, STM32F4, STM32L0`
- `WDT: STM32F1, STM32F2, STM32F4, STM32L0`
- `EEPROM: LPC11Uxx`
- `I2C: LPC1Uxx, LPC18xx`
- `ADC, DAC: STM32F1`
- `USB: STM32F1_CL, STM32L0, LPC11Uxx, LPC18xx`
- `ETH: LPC18xx`
- `SD/MMC: LPC18xx`
- Битбанг: `STM32F1, STM32F2, STM32F4, STM32L0, LPC11Uxx, LPC18xx`
- МЭЛТ mt12864j LCD

- Системные требования. Минимальные системные требования: `24КБ flash, 2КБ RAM`. 32 битная архитектура.

2.1. Экзоядро

Чтобы понять преимущества экзоядра, необходимо пройти весь путь от монолитного ядра.

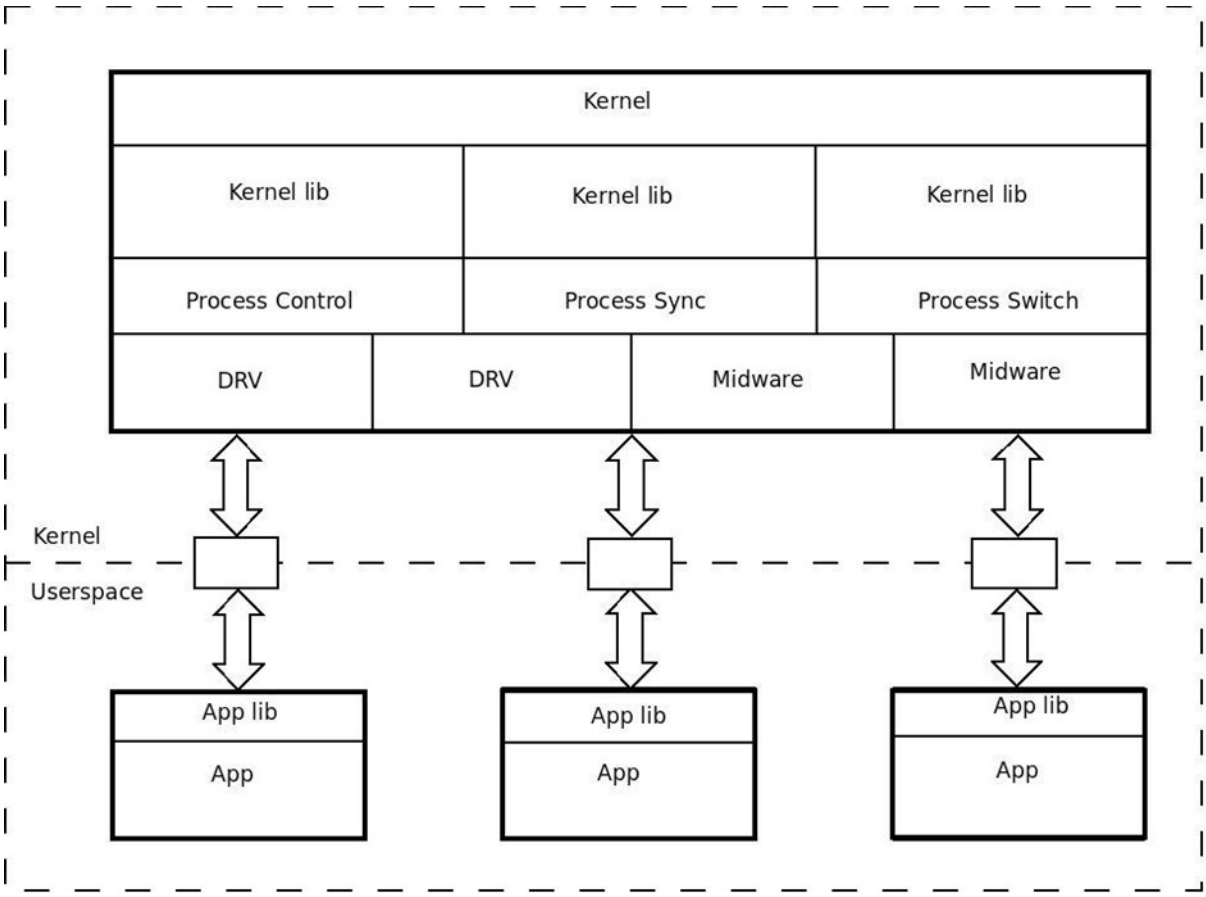


Рис. 1. Монолитное ядро

Основным преимуществом монолитного ядра, как видно из рис. 1, является простота реализации. Глобальным же недостатком является его неустойчивость — ошибка в каком-либо драйвере приведет к падению всей системы.

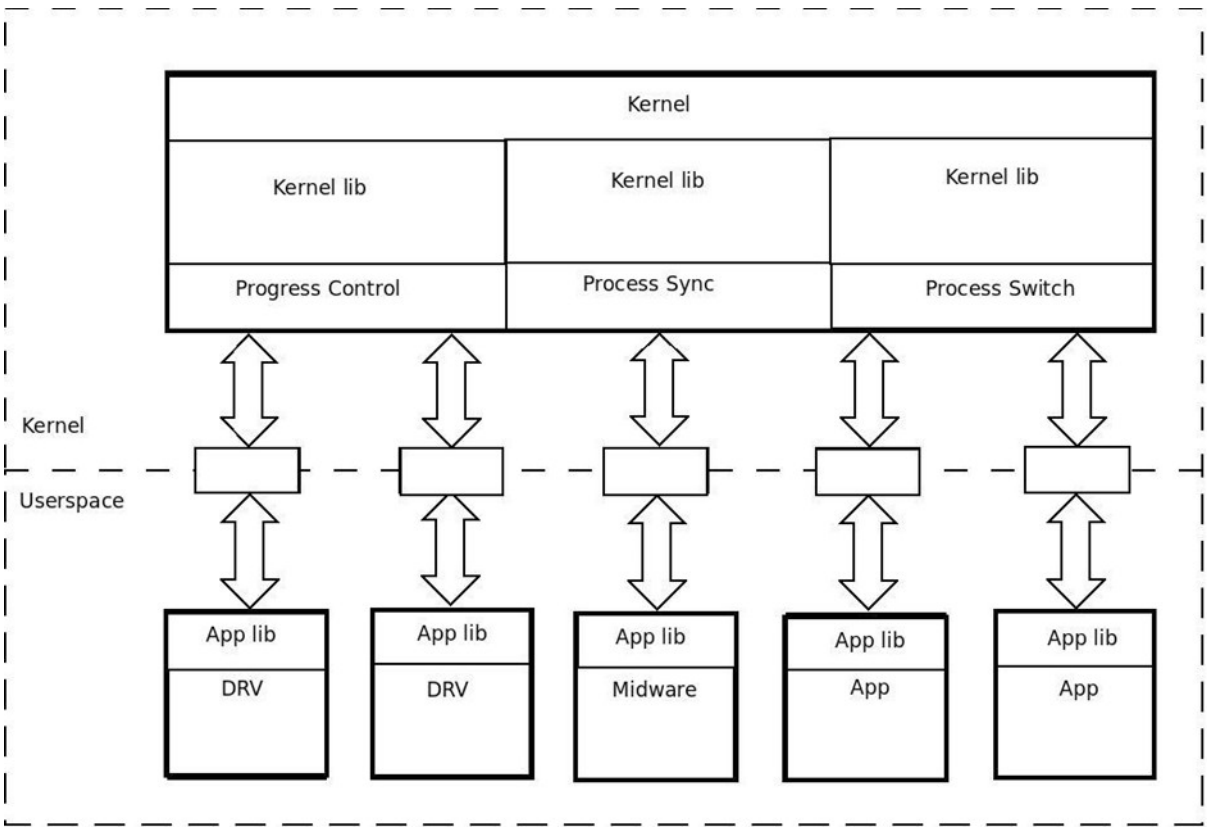


Рис. 2. Микроядро

Микроядро лишено недостатков монолитного ядра и позволяет создавать отказоустойчивые системы, что является критичным для микроконтроллеров. Однако если посмотреть внимательно на рис.2, можно увидеть, что в микроядре происходит дублирование библиотек — непосредственно для ядра и для каждого из процессов. Речь в первую очередь о таких базовых библиотеках, как `stdlib, stdio`. Для персональных компьютеров это не является проблемой, однако для микроконтроллера лишние 10 килобайт в каждом процессе могут стать существенной проблемой.

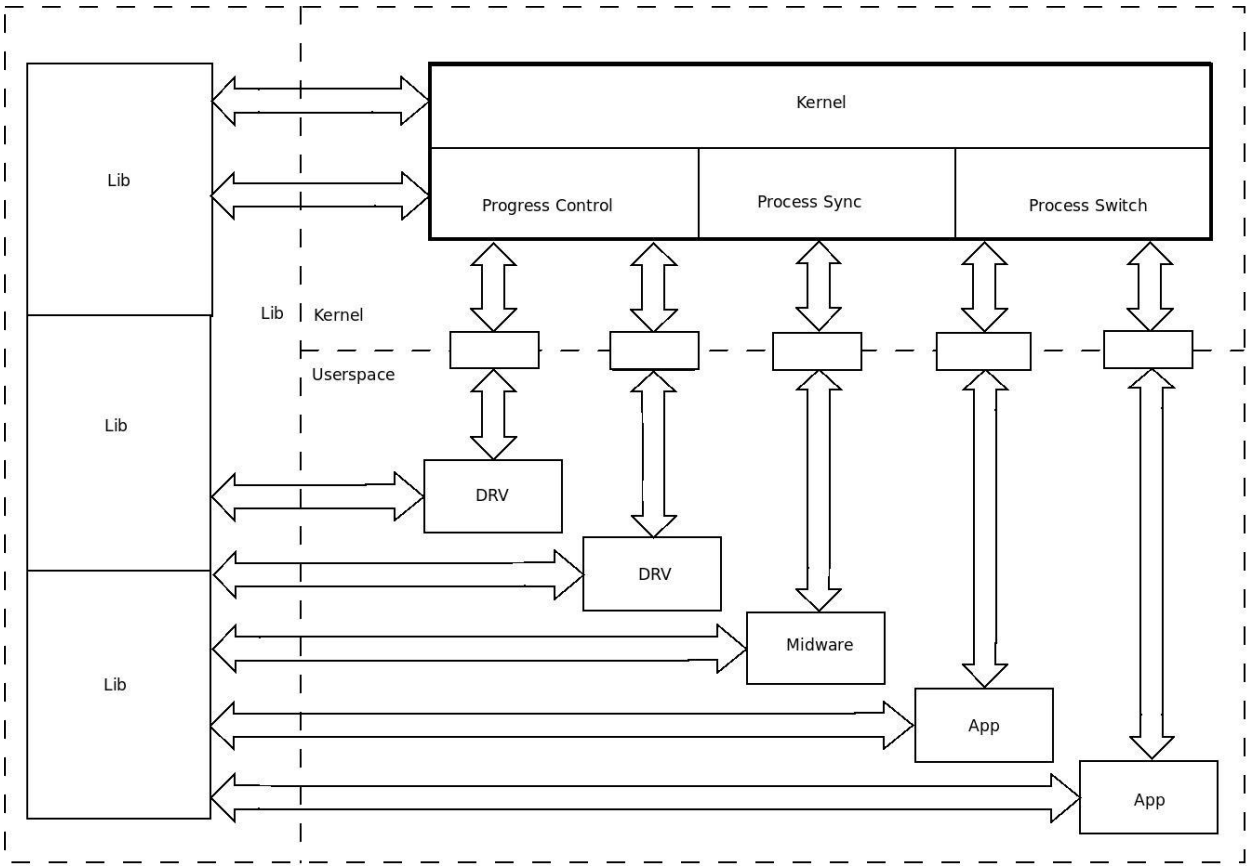


Рис. 3. Экзоядро

Экзоядро решает все вышеперечисленные проблемы, позволяя процессам быть размером хоть в сотню байт, имея при этом всю мощь системных библиотек и не перегружая систему лишними вызовами ядра. Также это позволяет реализовывать тяжелые библиотеки (например, графику), без создания дополнительных механизмов динамической линковки.

2.2. Высокоточная архитектура таймера

Стандартная модель реализации системного таймера в ОС РВ представлена на рис.4.

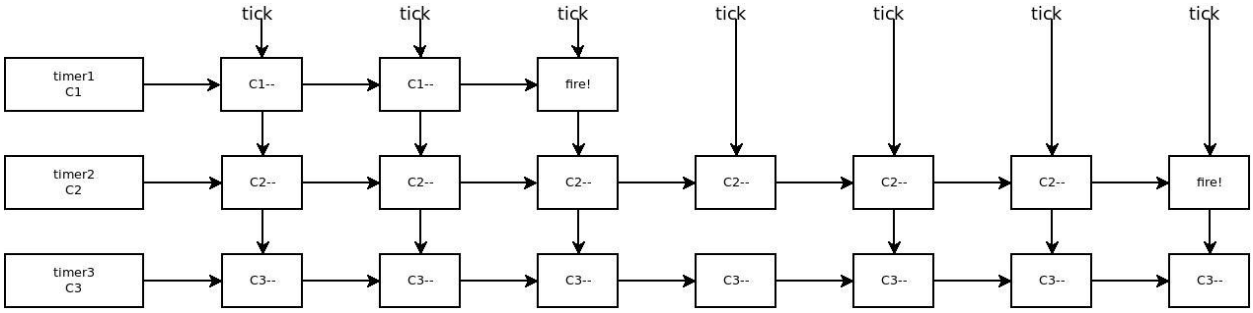


Рис. 4. Системный таймер

Модель стала настолько стандартом, что для нее создан специальный таймер в `Cortex-M`. Тем не менее, у нее есть огромное количество недостатков, а именно:

- Необходимость будить процессор каждые несколько миллисекунд для обработки тика.
- При каждой обработке тика проводить декремент всех активных таймеров
- Все таймеры привязаны к абстрактной величине — тику. При изменении продолжительности тика все таймеры изменят свою продолжительность.

И самый главный недостаток — высокая гранулярность таймера, вследствие большой продолжительности системного тика. Это не позволяет использовать системные тики для точного отсчета аппаратных задержек. В итоге, чтобы реализовать аппаратную задержку, приходится крутить пустые циклы, либо реализовывать аппаратным таймером с отдельной синхронизацией.

Подобная реализация не только ведет к повышенному энергопотреблению и потреблению дополнительного процессорного времени, но и полностью списывает большинство преимуществ ОСРВ.

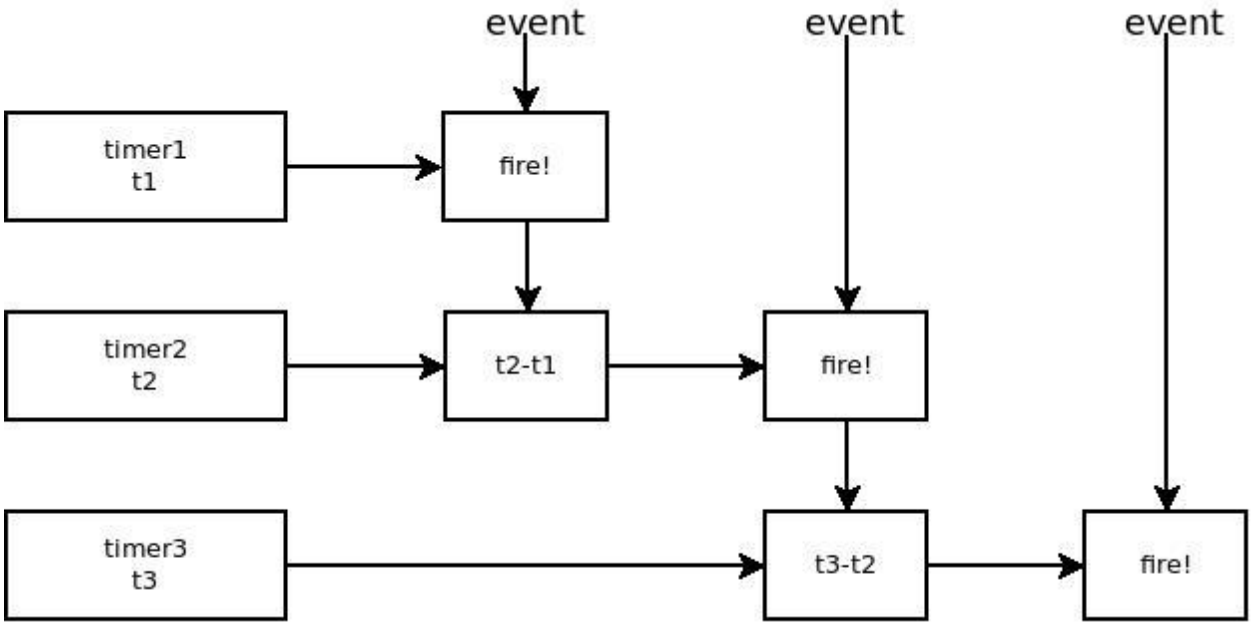


Рис. 5. Системный таймер `REx`

В `REx` используется системный таймер на базе событий. Это позволяет значительно снизить нагрузку на ядро и получить точность времени срабатывания таймера до 1 микросекунды (1 us). С максимальным периодом до 136 лет.

Фактическая реализация требует два таймера — ежесекундный импульс, и высокоточный таймер, которому задается дельта времени между двумя событиями в пределах диапазона секунды. Диапазон в секунду выбран не случайно — как правило, во многих микроконтроллерах уже есть `RTC`, в реализации которых есть прерывание секундного импульса.

Также подобная реализация позволяет без дополнительных накладных расходов

реализовывать аптайм системы с точностью до 1 микросекунды, а также оценивать время выполнения задач с аналогичной точностью.

2.3. Безмьютексная система

Второй глобальной проблемой производительности после системных таймеров в TRON-подобных ОСРВ являются мьютексы. Рассмотрим на примере.

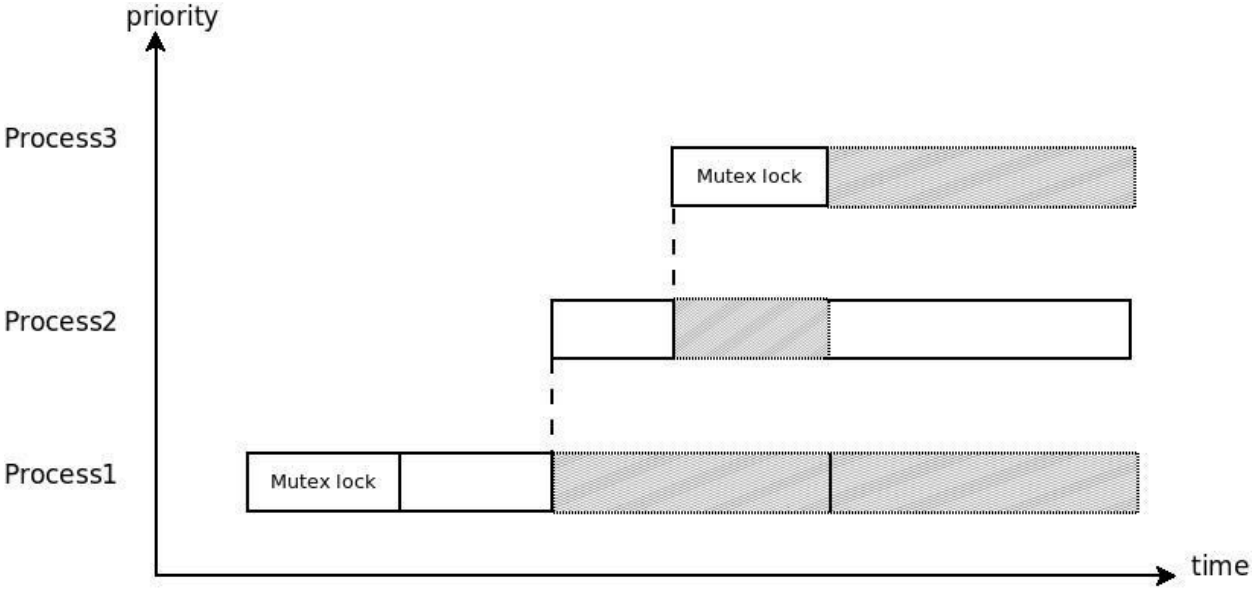


Рис. 6. Блокировка мьютексов

Время выполнения высокоприоритетного процесса №3 зависит от менее приоритетного процесса №2. Если процесс №3 работает непосредственно с железом, из-за удлиненного тайминга работоспособность может быть нарушена. Это классическая проблема мьютекса и имеет два стандартных полурешения:

а) `Ceiling`. Процесс, захватывающий мьютекс, поднимается до максимального. В результате при длительных задержках встанет вся система, а не только процессы, прямо или косвенно участвующие в захвате мьютекса. Большинство систем на рынке сейчас имеет именно эту реализацию в силу своей простоты.
б) `Nested mutex priority inheritance`. Процессу, захватившему мьютекс, присваивается приоритет наивысшего из всех процессов, ожидающих мьютекс. Эта реализация более корректна, но сильно медленнее предыдущей из-за того, что может проходить несколько вытеснений контекста за одну простую операцию. К тому же, она более сложна в реализации и занимает больше программного места.

Из вышесказанного видно, что проблема не имеет оптимального решения. А,

следовательно, необходимо строить архитектуру системы без использования мьютексов. В REx это реализовано следующим образом:

a) Блоки данных (IO) не принадлежат адресному пространству процесса и могут быть переданы посредством хэндлов и открытия окна MPU только для одного процесса. Копирование данных при этом не происходит.

b) Потоки (stream) используются тогда, когда несколько процессов одновременно записывают небольшие порции данных для одного получателя. Ярким примером такого использования может служить stdio.

В итоге единственным случаем одновременного использования одного блока данных является процесс с использованием прерываний. В таком случае используются критические секции, на базе запрета прерываний. В cortex-m это составляет одну ассемблерную инструкцию. В большинстве данный функционал выделен в процесс драйвера, что значительно упрощает первоначальное изучение REx.

3. Карта памяти

Карту памяти системы можно представить следующей схемой

Рис. 7. Карта памяти

3.1. Global

Окно, доступное для процесса в режиме чтения. Всегда находится по фиксированному адресу (обычно — младший адрес RAM). Необходимо для функционирования процесса и прерываний. Состав:

- heap. Указатель на адресное пространство текущего процесса.
- svc_irq. Адрес процедуры обработчика svc ядра. Необходим для прямого вызова обработчика, когда повышать контекст не требуется — из прерываний.
- lib. Указатель на структуру библиотек.

3.2. Kernel

Окно, доступное только ядру. Состав:

- код последней ошибки. Для взаимодействия с библиотеками.

- `stdout`. Отладочный вывод.
- Список выполняемых процессов. Хранится как указатель. Может быть пустым, если все процессы ждут события.
- Активный процесс. Процесс, чей контекст выполняется в данный момент. Может быть `NULL`.
- Контекст исполнения. Вектор прерывания или -1, если выполняется процесс.
- Прерывания. Пользовательский список указателей на структуры обработчиков векторов прерываний. По умолчанию — все заглушки.
- Переменные подсистемы таймера.
- Структура основного пула памяти.
- Объекты ядра. Пользовательские глобальные объекты.

3.3. Main pool
Основной пул, в котором создаются все системные объекты, и выделяется память для адресного пространства процессов.

3.4. Kernel stack(s)
Стек супервизора. Для ранних архитектур ARM — также отдельные стеки для других контекстов — например, IRQ.

4. Архитектура ядра

4.1. Аппаратно-зависимая часть
Находится в каталоге kernel/core. В составе:
- сценарий первоначальной загрузки;
- обработчики системных исключений;
- обработчик вызова ядра;
- обработчик переключения контекста;
- низкоуровневый маршрутизатор прерываний;
- `pend_switch_context`. Процедура запроса переключения контекста;
- `process_setup_context`. Процедура инициализации процесса;
- `disable_interrupts, enable_interrupts`. Процедуры запрета/разрешения прерываний;
- `fatal`. Низкоуровневый обработчик фатальных ошибок.

4.2. Ядро ядра

kernel/kernel.c:

- startup. Первоначальная инициализация системы;
- svc. Машрутизатор вызовов ядра;
- stdout_stub. Заглушка для stdout ядра;
- panic. Высокоуровневый обработчик фатальных ошибок.

4.3. Остальные подсистемы ядра

Более детальное описание работы с каждой подсистемой из userspace дано ниже. Здесь приведено лишь базовое описание для понимания, что и как работает. Все файлы ядра располагаются в каталоге kernel.

- kprocess. Реализация работы с процессами, переключение контекста;
- kirq. Высокоуровневый маршрутизатор прерываний. Программный VIC с вытесняющими приоритетами для систем, не имеющих аппаратной реализации (например, ARM7);
- ktimer. Реализация системного таймера и программных таймеров (если включены);
- dbg. Вывод отладочной информации для ядра. Работает при запрещенных прерываниях;
- kmalloc. Обертки для работы с менеджером памяти уровня ядра;
- kipc. Менеджер IPC. Может также использоваться непосредственно из ядра для отправки сообщения процессу (IPC_STREAM_WRITE, IPC_TIMEOUT).
- kio, kevent, kmutex, kobject, ksem, kstream. Реализация прочих объектов синхронизации. Используются только процессами.

5. Системные вызовы

Нельзя напрямую вызвать функцию супервизора из процесса. Это приведет к нарушению работы всей системы. Чтобы вызвать супервизора, нужно прервать выполнение процесса. По сути, это аналогично аппаратному прерыванию. В REx это осуществляется с помощью функции, svc_call. Фактическая реализация архитектурно-зависимая, детали реализации можно найти в каталоге userspace/core.

```
void svc_call(unsigned int num, unsigned int param1, unsigned int
param2, unsigned int param3)
```

num: номер системного вызова. Список, как и сама функция — в файле userspace/svc.h

`param1, param2, param3`: параметры системного вызова. Специфичны для каждого из вызовов.
Функция не возвращает значение (и не может возвращать, учитывая, что вызов и возврат не обязательно атомарны) — если необходимо вернуть какие-то данные в процесс, параметры могут быть использованы как указатель на структуру в адресном пространстве процесса.

Все описанные ниже функции работы с супервизором являются лишь обертками к этой функции. Также 4 параметра используются неслучайно — это максимальное количество параметров в `ARM`, которые можно передать, не используя стек.

5.1. Обработка прерываний

Аппаратные прерывания маршрутизируются через ядро. Это позволяет переопределить их в процессе выполнения, а также передать параметры из процесса, связанного с прерыванием. Дополнительные накладные расходы составляют порядка 10 тактов. С целью безопасности после того как прерывание определено, оно блокируется на процесс, его определивший.
`typedef void (*IRQ)(int vector, void* param)`
Прототип функции прерывания.
`vector` — номер вектора прерывания
`param` — пользовательский параметр, передаваемый в прерывание

`void irq_register(int vector, IRQ handler, void* param)`
Регистрация прерывания.
`vector` —номер вектора прерывания.
`handler` — процедура обработчика прерывания
`param` — пользовательский параметр

`void irq_unregister(int vector)`
Освобождение ранее зарегистрированного прерывания
`vector` — номер вектора прерывания

Хорошей практикой является работа с прерываниями лишь в драйверах процессов. Но требование жесткого реалтайма может накладывать свои исключения.

5.2. Хэндлы

Хэндлы имеют два разных значения. Системные хэндлы — указатели на системные объекты, доступные лишь для ядра. Пользовательские хэндлы — идентификаторы объектов,

специфичные для конкретного процесса или представляющие собой аппаратную абстракцию (см. ниже). Для использования супервизора зарезервированы следующие хэндлы:
`INVALID_HANDLE` — неверный хэндл
`ANY_HANDLE` — любой хэндл. Используется как маска.
`KERNEL_HANDLE` — хэндл, идентифицирующий ядро.

5.3. Объекты ядра

Объекты ядра — глобальные хэндлы, зарегистрированные в ядре и доступные всем процессам для чтения. По сути, представляют собой аналог `ROOT FS` в условиях очень ограниченной памяти.

```
void object_set(int idx, HANDLE object)
```

Зарегистрировать глобальный объект. После того как хэндл зарегистрирован, только владелец может его изменить.
`idx`: индекс объекта
`object`: значение хэндла

```
void object_set_self(int idx)
```

Зарегистрировать хэндл собственного процесса как глобальный объект.
`idx`: индекс объекта.

```
HANDLE object_get (int idx)
```

Получить хэндл глобального объекта.
`idx`: индекс объекта.

6. Процессы

6.1. Создание процесса

Для помощи супервизору в процессе создания процесса используется структура `REx`:

```
typedef struct {
    const char* name;
    unsigned int size;
    unsigned int priority;
    unsigned int flags;
```

```
    void (*fn) (void);
}REX;
```

name: указатель на строку с именем процесса;

size: размер оперативной памяти процесса;

priority: базовый приоритет процесса. Меньше — выше;

flags: список флагов процесса:

PROCESS_FLAGS_ACTIVE — запустить процесс после создания.

REX_HEAP_FLAGS(HEAP_PERSISTENT_NAME) — имя процесса находится в постоянной памяти, нет необходимости его резервировать в оперативной памяти.

fn: точка входа процесса

```
HANDLE process_create(const REX* rex)
```

Создание процесса. Возвращает хэндл процесса или INVALID_HANDLE в случае ошибки

rex: указатель на вышеописанную структуру.

Для создания процесса нужно лишь знать указатель на структуру процесса, поэтому процессы можно загружать из внешних носителей либо запускать из любого места во flash, скомпилированных отдельно.

6.2. Жизненный цикл процесса

Жизненный цикл процесса — это инициализация и бесконечный цикл, в котором он получает, обрабатывает и отправляет IPC.

```
void process1()
{
IPC ipc;
MY_STRUCT struct;
process_init(&struct);
bool need_post;
for(;;)
{
    error(ERROR_OK);
    ipc_read_ms(&ipc, 0, ANY_HANDLE);
    need_post = false;
    switch (ipc.cmd)
    {
    case CMD1:
```

```
            need_post = process_cmd1(&struct, ipc.param1);
            break;
        case CMD2:
            need_post = process_cmd2(&struct, ipc.param1);
            break;
        default:
            break;
        }

        if (need_post)
            ipc_post_or_error(&ipc);
        }
}
```

6.3. Завершение процесса

Процесс не может выйти за пределы бесконечного цикла. Чтобы завершить процесс, нужно вызвать специальную команду.

`void process_destroy(HANDLE process)`

Завершить процесс.

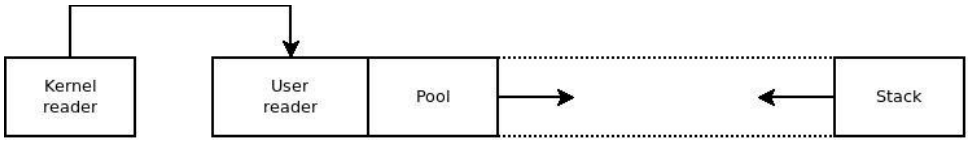
`process`: хэндл процесса

`void process_exit()`

Завершить текущий процесс.

6.4. Карта памяти процесса

Процесс состоит из системного объекта `PROCESS` и адресного пространства, выделенного ему супервизором.



Заголовок процесса состоит:

- из `error`. Код последней ошибки;
- `flags`. Пользовательские флаги процесса;
- `pool`. Структура динамической памяти процесса;
- `stdio`. Хэндлы;

- имя или указатель на имя процесса.

6.5. Остальное API процесса

`HANDLE process_get_current()`

`HANDLE process_iget_current()`

Получить хэндл текущего процесса. Здесь и далее префикс i обозначает необходимость вызова команды из контекста прерывания.

`unsigned int process_get_flags (HANDLE process)`

Получить флаги процесса.

`process`: хэндл процесса

`void process_set_flags (HANDLE process, unsigned int flags)`

Установить флаги процесса. На данный момент только `PROCESS_FLAGS_ACTIVE` может быть установлено, что равносильно вызову `freeze/unfreeze`

`process`: хэндл процесса

`flags`: флаги для установки

`void process_unfreeze (HANDLE process)`

Возобновить исполнение процесса.

`process`: хэндл процесса

`void process_freeze (HANDLE process)`

Остановить исполнение процесса.

`process`: хэндл процесса

`unsigned int process_get_priority (HANDLE process)`

Получить приоритет процесса.

`process`: хэндл процесса

`unsigned int process_get_current_priority()`

Получить приоритет текущего процесса.

`void process_set_priority(HANDLE process, unsigned int priority)`

Установить приоритет процесса.

process: хэндл процесса

priority: значение приоритета

```
void process_set_current_priority (unsigned int priority)
```

Установить приоритет текущего процесса.

priority: значение приоритета

```
void sleep (SYSTIME* time)
void sleep_ms (unsigned int ms)
void sleep_us (unsigned int us)
```

Приостановить текущий процесс до тайм-аута. Супервизор переходит при этом к исполнению следующего процесса или останавливает ядро. Здесь и далее параметры времени указываются в трех видах:

time: указатель на структуру SYSTIME;

ms: относительно время в миллисекундах;

us: относительное время в микросекундах.

6.6. Профилирование процессов

Для использования профилирования процессов предварительно должна быть включена опция в настройках ядра.

```
void process_switch_test()
```

Тестирование переключения процессов. Может использоваться для оценки производительности ядра.

```
void process_info()
```

Вывод отладочной информации о списке процессов и используемой памяти.

7. Синхронизация процессов

7.1. Межпроцессные сообщения.

Межпроцессные сообщения, или IPC, позволяют отправлять сообщения от одного процесса другому. В своем принципе это аналог системного вызова с той лишь разницей, что сообщение происходит с другим процессом, а не ядром системы.

```
typedef struct {
    HANDLE process;
    unsigned int cmd;
```

```
    unsigned int param1;
    unsigned int param2;
    unsigned int param3;
} IPC;
```

process: хэндл процесса, которому мы отправляем сообщение либо от которого сообщение пришло. Отправить сообщение можно только другому процессу, прийти сообщение может от другого процесса или ядра, в последнем случае значение этого параметра будет KERNEL_HANDLE.

cmd: код сообщения.

param1, param2, param3: Параметры, специфичные для кода запроса. В общем случае рекомендуется придерживаться правила: param1 — хэндл объекта, param2 — объект данных, param3 — размер или код ошибки. Однако данное правило не является обязательным.

IPC условно можно разделить на два типа:

- Сообщения с запросом результата выполнения.
- Сообщения без запроса результата выполнения.

Тип сообщения задается при формировании кода cmd установкой флага HAL_REQ_FLAG или с помощью макроса HAL_REQ(group, item). Также может быть установлен флаг HAL_IO_FLAG, но он предназначен для передачи больших объемов информации с помощью блоков IO, о который будет рассказано позже.

Код сообщения состоит из двух частей. Старшие 2 байта кода содержат HAL группу сообщения, а младшие 2 байта содержат тип IPC сообщения. В заголовочном файле userspace/ipc.h определены следующие макросы, для упрощения формирования и разбора кода сообщений:

HAL_CMD(group, item) — макрос формирует команду cmd, где group — группа HAL сообщения, a item — тип IPC сообщения.

HAL_REQ(group, item) — аналогичен предыдущему макросу, но помечает команду как сообщение с запросом результата выполнения.

HAL_GROUP(cmd) — макрос извлекает HAL группу из cmd IPC сообщения.

HAL_ITEM(cmd) — макрос извлекает IPC группу из cmd IPC сообщения.

Отправка и вызовы IPC

Для отправки сообщения другому процессу используется функция ipc_post, для вызова с ожиданием результата — функция call. Функция ipc_post работает в асинхронном режиме, другими словами, процесс-отправитель после вызова может параллельно заниматься другими задачами. Функция call работает в синхронном режиме. Небольшой пример кода:

```
//Отправка без ожидания ответа.
IPC ipc;
ipc.cmd = cmd;
ipc.process = usbd->user;
ipc.param1 = USBD_IFACE(iface, num);
    ipc.param2 = param2;
    ipc.param3 = param3;
ipc_post(&ipc);
```

```
//Отправка с ожиданием ответа.

IPC ipc;
    ipc.cmd = HAL_REQ(HAL_ETH, ETH_GET_MAC);
    ipc.process = eth;
    ipc.param1 = eth_handle;
call(&ipc);

//Дождались ответа и продолжаем работу.
mac->u32.hi = ipc.param2;
mac->u32.lo = ipc.param3;
```

Рабочий цикл приема/отправки сообщений.

Для приема сообщения и обработки "вызова" от другого процесса используется функция `ipc_read` и `ipc_read_ex`, а для отправки результата выполнения вызова используется функция `ipc_write`.

Функция `ipc_read` ожидает любое сообщение, адресованное данному процессу, а функция `ipc_read_ex` строго определенное сообщение. Небольшим примером использования `ipc_read_ex` может быть следующая ситуация: при инициализации процесса рабочий цикл приема/отправки сообщений не будет запущен до тех пор, пока мы не дождемся определенного сообщения от другого процесса.

Пример.

```
void app()
{
    ..............


    IPC ipc;

    // Рабочий цикл процесса
    for (;;)
    {
        ipc_read(&ipc); // Ждем сообщение.
        // Разбор команды IPC.
        switch (HAL_GROUP(ipc.cmd))
        {
        case HAL_USBD:
            comm_request(&app, &ipc);
            break;


        ..............
        default:
            error(ERROR_NOT_SUPPORTED);
            break;
        }
        ipc_write(&ipc); // Записываем ответ.
    }
}
```

В данном примере хотелось бы обратить внимание на функцию ipc_write. Она записывает результат выполнения только в том случае, если команда в IPC сообщении отмечена как сообщение с запросом результата выполнения (HAL_REQ_FLAG), в противном случае ответа отправителю не последует. Хотелось бы добавить, что в REx введена специфическая ошибка ERROR_SYNC. Представим такую ситуацию, что ответ на IPC сообщение не может быть сформирован сразу из-за того, что результат зависит от работы таймера. А рабочий цикл приема/отправки сообщений работает параллельно с таймером, и из-за этого может возникнуть ситуация, что рабочий цикл вышлет преждевременный ответ, и именно для этих целей была введена ошибка ERROR_SYNC. Процедура обработчик сообщений отправляет ошибку синхронизации, которая говорит функции ipc_write, что не нужно отправлять ответ на пришедшее сообщение сразу, ответ будет выслан позднее.

Сообщения «геттеры»

Сообщения «геттеры» используются в случае, когда нам необходимо отправить запрос другому процессу на получение какой-либо информации или данных. К функциям данного типа относятся функции get, get_handle, get_size. Функция get, как правило, используется для запроса получения некоего объекта данных. Функция get_handle используется для запроса хэндла некоего объекта, а функция get_size для получения размера объекта. Все три функции принимают одни единственный параметр IPC* ipc и возвращают значение, а также записывают результат своего выполнения в param2 и код ошибки в param3, если она произошла.

Справочная информация

Все представленные константы, функции, макросы можно найти в заголовочном файле userspace/ipc.h.

Ядром зарезервированы следующие HAL группы сообщений:

```
HAL_SYSTEM

HAL_PIN, HAL_POWER, HAL_TIMER, HAL_RTC, HAL_WDT, HAL_UART,
HAL_USB, HAL_ADC, HAL_DAC, HAL_I2C, HAL_LCD, HAL_ETH, HAL_FLASH,
HAL_EEPROM, HAL_SDMMC

HAL_USBD, HAL_USBD_IFACE, HAL_TCPIP, HAL_MAC, HAL_ARP, HAL_ROUTE,
HAL_IP, HAL_ICMP, HAL_UDP, HAL_TCP, HAL_HTTP, HAL_TLS,
HAL_PINBOARD

HAL_APP
```

Ядром зарезервированы следующие типы IPC сообщений:

IPC_PING — проверка жизнеспособности процесса. Все процессы обязаны отвечать на это сообщение таким же.

IPC_TIMEOUT — тайм-аут программного таймера.

Также некоторые сообщения зарегистрированы системой (система не связана с ядром и не обязательно может быть его частью, являясь лишь одним из вариантов реализации):

IPC_READ, IPC_WRITE, IPC_CANCEL_IO, IPC_FLUSH, IPC_SEEK, IPC_OPEN, IPC_CLOSE. IPC_GET_RX_STREAM, IPC_GET_TX_STREAM — получить хэндл потока. Используется для процессов, осуществляющих обмен данными с помощью двунаправленных потоков данных — например: uart, CDC.

IPC_USER — все пользовательские сообщения не должны быть меньше значения данного

кода.

API "userspace/ipc.h"

`void ipc_post(IPC* ipc)`
Отправляет сообщение другому процессу без синхронизации ответа.

`void ipc_post_inline(HANDLE process, unsigned int cmd, unsigned int param1, unsigned int param2, unsigned int param3)`
Аналогична предыдущей функции за тем лишь исключением, что принимает сообщение в виде параметров, а не указателем на структуру.

`void ipc_ipost(IPC* ipc)`
`void ipc_ipost_inline(HANDLE process, unsigned int cmd, unsigned int param1, unsigned int param2, unsigned int param3)`
Должна использоваться из прерываний.

`void call(IPC* ipc)`
Отправляет сообщение другому процессу с синхронизацией ответа.

`void ack(HANDLE process, unsigned int cmd, unsigned int param1, unsigned int param2, unsigned int param3)`
Аналогична предыдущей функции за тем лишь исключением, что принимает сообщение в виде параметров, а не указателем на структуру.

`void ipc_read(IPC* ipc)`
Ожидает сообщение.

`void ipc_read_ex(IPC* ipc, HANDLE process, unsigned int cmd, unsigned int param1)`
Ожидает строго определенное сообщение от определенного процесса `process` с определенной командой `cmd` и, как правило, определенным хэндлом объекта `param1`, если следовать договоренности.

`void ipc_write(IPC* ipc)`
Отправляет ответ на сообщение.

`unsigned int get(HANDLE process, unsigned int cmd, unsigned int param1, unsigned int param2, unsigned int param3)`
Запрашивает объект данных. Результатом возврата будет указатель на объект.

`unsigned int get_handle(HANDLE process, unsigned int cmd, unsigned int param1, unsigned int param2, unsigned int param3)`
Запрашивает хэндл объекта. В случае ошибки возвращает `INVALID_HANDLE`.

`int get_size(HANDLE process, unsigned int cmd, unsigned int param1, unsigned int param2, unsigned int param3)`
Запрашивает размер объекта данных. Результатом возврата будет размер в байтах или код ошибки (отрицательное значение).

7.2. Потоки

Потоки используются для записи данных из одного или нескольких источников в один приемник. Владелец потока создает объект потока и раздает другим процессам хэндлы потока. Процесс создает хэндл экземпляра потока и пишет в поток через экземпляр потока данные или читает данные из него. Если в потоке недостаточно данных при чтении или недостаточно свободного места при записи, процесс засыпает до тех пор, пока операция не завершится успешно. Зачастую потоки используются парой — один для чтения, другой для записи. Внутренняя реализация потоков основана на кольцевых буферах.

`HANDLE stream_create(unsigned int size)`

Создать поток. Возвращает хэндл потока или `INVALID_HANDLE` в случае ошибки.

`size`: размер потока в байтах

`HANDLE stream_open(HANDLE stream)`

Открыть экземпляр потока. Возвращает хэндл потока или `INVALID_HANDLE` в случае ошибки.

`stream`: хэндл ранее созданного потока командой `stream_create`.

`void stream_close(HANDLE handle)`

Закрыть экземпляр потока.

`handle`: хэндл ранее созданного экземпляра потока командой stream_open.

`void stream_destroy(HANDLE stream)`

Уничтожает поток. Все экземпляры потока при этом закрываются.

`stream`: хэндл потока

`unsigned int stream_get_size(HANDLE stream)`

Возвращает размер данных потока. При использовании правила одного получателя данное значение гарантированно не уменьшится.

`stream`: хэндл потока

`unsigned int stream_get_free(HANDLE stream)`

Возвращает размер свободного места в потоке. Команда имеет смысл только при условии, что у потока один отправитель. Например, драйвер uart.

`stream`: хэндл потока

`bool stream_listen (HANDLE stream, void* param)`

Начать слушать поток. Как только в потоке появятся данные, процесс получит от ядра сообщение о заполнении потока следующего вида — process: KERNEL_PROCESS, cmd: `IPC_STREAM_WRITE, param1: param, param2: stream, param3:` size.

После отправки данного сообщения мониторинг потока ядром останавливается. Возвращает true в случае успеха.

stream: хэндл потока

param: пользовательский хэндл, идентифицирующий данный поток

`bool stream_stop_listen(HANDLE stream)`

Остановить чтение потока.

stream: хэндл потока

`bool stream_write (HANDLE handle, const char* buf, unsigned int size)`

Записать данные в поток. True в случае успеха.

handle: хэндл экземпляра потока

buf: указатель на буфер данных

size: размер буфера данных

`bool stream_read(HANDLE handle, char* buf, unsigned int size)`

Прочитать данные из потока. True в случае успеха.

handle: хэндл экземпляра потока

buf: указатель на буфер данных

size: размер буфера данных

`void stream_flush(HANDLE stream)`

Очистить поток. Все операции чтения и/или записи при этом завершаются.

stream: хэндл потока

Для процессов, работающих с потоками, могут также использоваться запросы get для получения хэндла потока:

```
get (process, IPC_GET_RX_STREAM, handle, 0, 0);
get (process, IPC_GET_TX_STREAM, handle, 0, 0);
```

где:

process: процесс-владелец потока;

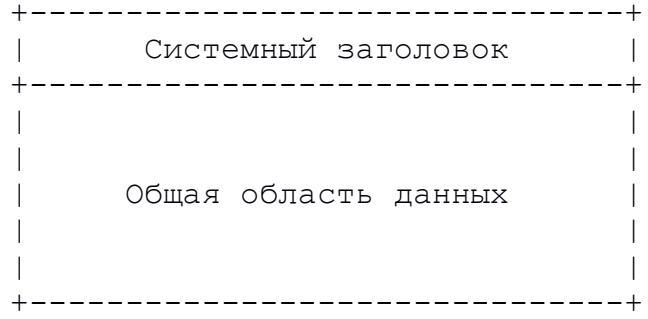
handle: пользовательский хэндл, идентифицирующий поток.

7.3. Блоки IO

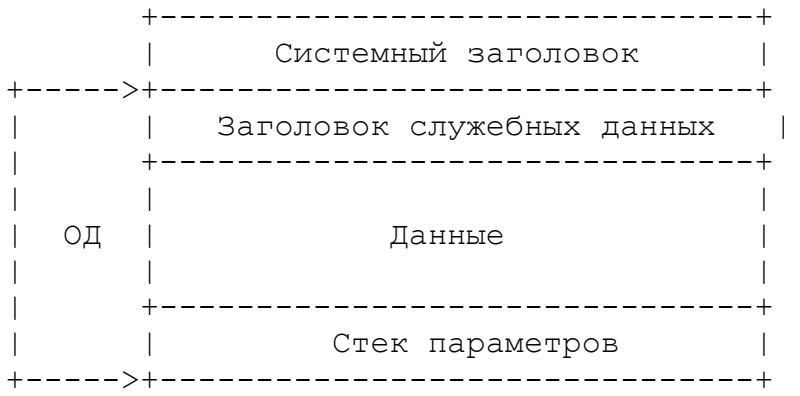
Когда необходима передача больших объемов данных между процессами, в REx используются блоки IO.

Структура и предназначение

Блок IO можно представить следующим рисунком.



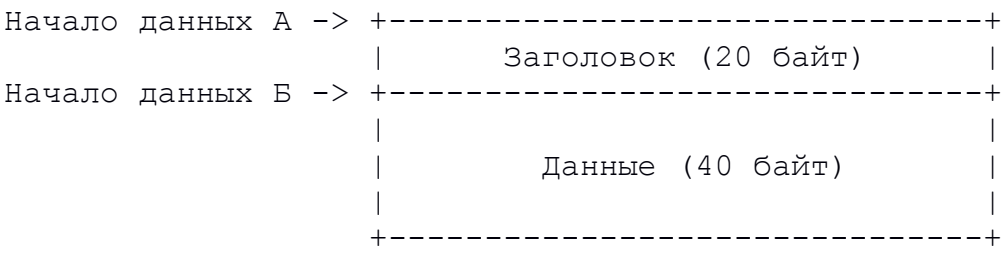
Но при необходимости общая область данных (для сокращения назовем ее ОД) может представлять собой три логические группы: заголовок служебных данных (СД), стек параметров (СП) и непосредственно сами данные (ДН), и тогда IO будет иметь следующий вид.



Представим ситуацию. У нас есть некий поток пакетов данных размеров в 60 байт, где 20 первых байт — это служебная информация, а оставшиеся 40 байт непосредственно сами данные. У нас есть процесс (отправитель), который принимает эти пакеты. В его задачу входит простое действие — принять этот пакет данных, отделить служебную информацию и отправить непосредственно сами данные в другой процесс (приемник). Одно из решений (и возможно часто встречающееся во многих проектах) данной ситуации — это скопировать 40 байт данных и отправить другому процессу. К сожалению, такое решение не выглядит оптимальным для ОСРВ, потому что оно может повлечь многократное копирование данных, особенно если данный пакет данных будет передаваться по более длинной цепочке, и задача у процессов будет одинаковой (условно говоря) — отделять служебную информацию и передавать остаток данных другому.

Именно для решения данного рода задач были введены блоки IO в REx. Они позволяют избавиться от лишних операций копирования и тем самым уменьшить фрагментацию памяти

в самой ОС. Рассмотрим все тот же пример с пакетом данных в 60 байт.



Для процесса (отправителя) данные начинаются с отметки А, а для процесса (приемника) с отметки Б. Отправитель создает блок IO размером в 61 байт, записывает в него данные и перед отправкой блока смещает указатель начала данных (который стоит на отметке А) на 20 байт вперед (до отметки Б) и выделяет 1 байт под стек параметров, в который записывает размер смещения. Когда приемник получит IO, то он не будет знать, где "реально" начинаются данные, он будет считывать/записывать данные в IO с учетом смещения (прозрачно), которое задал отправитель. Процесс-отправитель, условно говоря, прячет часть данных от процесса приемника.
Блок IO представляет собой структуру:

```
typedef struct {
    HANDLE kio;
    unsigned int size, data_offset, data_size, stack_size;
} IO;
```

`HANDLE` — хэндл блока. Системный заголовок.
`size` — общий размер блока, включая в себя заголовок служебных данных, данные и стек параметров.

`data_offset` — смещение указателя начала данных ДН.

`data_size` — размер общих данных.

`stack_size` — размер стека параметров.

Создание и первые действия

За создание нового блока IO отвечает функция `io_create`, в качестве своего единственного аргумента она принимает размер создаваемого блока в байтах, она выделяет из общей памяти место для блока, и возвращает указатель на созданный блок. Перед отправкой блока можно поместить часть данных (спрятать) в служебный заголовок с помощью функции `io_hide`, обратное действие можно выполнить с помощью функции `io_unhide`, `io_show`.

Отправка и ответ

Отправка блока представляет собой отправку IPC сообщения/вызов. В команде IPC (`cmd`) перед отправкой нужно добавить флаг HAL_IO_FLAG. Добавить флаг можно как вручную, так и воспользоваться макросами (которые находятся в `userspace/ipc.h`) `HAL_IO_CMD(group, item)` или `HAL_IO_REQ(group, item)`. За отправку отвечают следующие функции: `io_write`, `io_read`, `io_complete`, `io_complete_ex`, `io_write_sync`, `io_read_sync`. С помощью `io_write` мы отправляем запрос другому процессу на запись данных в переданный ему блок, функция `io_read`, напротив, отправляет запрос на чтение, другими словами, процесс (приемник) должен записать некие

данные в переданный ему блок.

```
// Показать данные блока.
io_show(session->io);

// Спрятать часть данных блока, другими словами
// увеличить служебный заголовок данных.
io_hide(session->io, session->data_off);

.............

io_write(hss->process, // Хэндл процесса.
    // Команда IPC
    HAL_IO_REQ(HAL_HTTP, (IPC_USER + session->method_idx)),
    // Хэндл объекта
    (unsigned int)session,
    // Блок io
    session->io);
```

Функции `io_complete` и `io_complete_ex` очень похожи на функции `io_wirte`, `io_read`, но с той лишь разницей, что при формировании *IPC* команды `cmd` для `io_complete` и `io_complete_ex` нужно использовать макрос `HAL_IO_CMD`. Для функций `io_write`, `io_read` нужно воспользоваться макросом `HAL_IO_REQ`. Функция `io_complete` должна использоваться для ответа на запрос `io_write`, `io_read`. Функции `io_write_sync`, `io_read_sync` работают так же, как и `ipc_write`, `ipc_read`, но возвращают размер записанных или прочитанных данных или код ошибки, который меньше нуля.

Функция `io_async_wait` будет ждать результата нашего запроса на чтение или запись до тех пор, пока не получит какой-либо ответ или код ошибки. Небольшой пример:

```
io_write(hss->process, // Хэндл процесса.
    // Команда IPC
    HAL_IO_REQ(HAL_HTTP, (IPC_USER + session->method_idx)),
    // Хэндл объекта
    (unsigned int)session,
    // Блок io
    session->io);

.............

// Ждем (процесс спит) ответа.
result = io_async_wait(hss->process,
  HAL_IO_REQ(HAL_HTTP, (IPC_USER + session->method_idx)),
  (unsigned int)session
);
```

API "userspace/io.h"

`IO* io_create(unsigned int size)`

Создает новый блок, принимает размер в байтах и возвращает указатель на созданный блок.

`void* io_data(IO* io)`

Возвращает указатель на начало данных с учетом смещения служебного заголовка данных.

`void* io_stack(IO* io)`

Возвращает указатель на начало стека параметров.

`void *io_push(IO* io, unsigned int size)`

Резервирует или увеличивает размер блока `size` данных для стека параметров из ОД и возвращает указатель на начало СП. В случае ошибки (например, недостаточно свободного места для резервирования) вернет NULL.

`void io_push_data(IO* io, void* data, unsigned int size)`

Аналогична предыдущей функции, но копирует данные `data` на вершину стека, `size` содержит количество копируемых данных.

`void* io_pop(IO* io, unsigned int size)`

Уменьшает размер блока стека параметров до размера `size`.

`unsigned int io_get_free(IO* io)`

Возвращает оставшееся количество свободного места.

`unsigned int io_data_write(IO* io, const void *data, unsigned int size)`

Копирует `data` на вершину стека ДН, `size` содержит количество копируемых данных. В случае успеха вернет количество копируемых данных, равное размеру, указанному в `size`, в противном случае вернет размер свободного места.

`unsigned int io_data_append(IO* io, const void *data, unsigned int size)`

Аналогична предыдущей функции, но копирует данные `data` в конец блока ДН.

`void io_reset(IO* io)`

Сбрасывает состояние блока, до первоначального.

`void io_hide(IO* io, unsigned int size)`

«Прячет» часть данных от начала блока до `size`.

`void io_unhide(IO* io, unsigned int size)`

Открывает часть скрытых данных. Другими словами уменьшает заголовок служебных данных на размер `size`.

`void io_show(IO* io)`

Показывает все данные блока.

`int io_async_wait(HANDLE process, unsigned int cmd, unsigned int handle)`

Ожидает результат ответа на ранее отправленный запрос.

`void io_destroy(IO* io)`

Уничтожает блок IO.

`io_write(process, cmd, handle, io)`

Отправляет запрос на запись другому процессу.

`io_read(process, cmd, handle, io, size)`

Отправляет запрос на чтение другому процессу.

`io_complete(process, cmd, handle, io)`

Отправляет ответ на запрос `io_write`, `io_read`.

`io_complete_ex(process, cmd, handle, io, param3)`

Отправляет ответ на запрос `io_write`, `io_read`, но с той лишь разницей, что в `param3` в основном передается код ошибки или размер прочитанных/записанных данных.

`iio_complete(process, cmd, handle, io)`

Аналогична `io_complete`, но должна использоваться из прерываний.

`iio_complete_ex(process, cmd, handle, io, param3)`

Аналогична `io_complete_ex`, но должна использоваться из прерываний.

`int io_write_sync(process, cmd, handle, io)`

Отправляет запрос на запись с ожиданием ответа от процесса. В случае неудачи вернет код ошибки.

`int io_read_sync(process, cmd, handle, io, size)`

Отправляет запрос на чтение с ожиданием ответа процесса. В случае неудачи вернет код ошибки.

8. Таймеры

После первичной инициализации драйверов приложение обязано инициализировать системный таймер и запустить ежесекундный импульс. После инициализации процедуры системных вызовов будут заблокированы с целью безопасности. Обычно данные функции уже реализованы в базовых драйверах таймера и RTC, данная информация приводится лишь для понимания общих принципов работы REx.

8.1 Системные таймеры

Для инициализации системного таймера используется структура с набором функций-каллбэков, которые будут вызываться ядром по необходимости. Следует подходить с особой осторожностью к написанию этих функций, так как они, с целью высокой производительности, будут вызываться из контекста супервизора напрямую. Также функции обязаны работать даже при условии запрета прерываний.

```
typedef struct {
    void (*start) (unsigned int us, void* param);
    void (*stop) (void* param);
    unsigned int (*elapsed) (void* param);
} CB_SVC_TIMER;
```

`start`: Запустить системный таймер с указанным значением микросекунд. Таймер должен также поддерживать режим `free run`, когда нет системных событий, в таком случае значение us будет > 1 секунды.
`stop`: Остановить системный таймер.

`elapsed`: Возвращает количество микросекунд, прошедших со времени запуска таймера.

```
void systime_hpet_setup(CB_SVC_TIMER* cb_svc_timer, void*
cb_svc_timer_param)
```

Функция проводит первичную инициализацию таймера. Функция может быть вызвана только один раз после запуска ОС, все дальнейшие вызовы вернут ошибку.

```
void systime_second_pulse()
```

Информирование ядра о том, что прошел секундный импульс. Вызывается только из контекста прерывания (таймера или RTC).

```
void systime_hpet_timeout()
```

Информирование ядра о том, что произошел тайм-аут высокопрецизионного таймера. Вызывается только из контекста прерывания (таймера). Срабатывание тайм-аута в режиме free run, информирует систему об отсутствии или некорректных настройках секундного импульса.

Для хранения информации таймеров используется тип `SYSTIME`, который представляет собой следующую структуру:

```
typedef struct _SYSTIME{
    unsigned int sec;
    unsigned int usec;
} SYSTIME;
```

`sec`: количество секунд.
`usec`: количество микросекунд.

Пример:

```
SYSTIME uptime;
int i;
```

```
unsigned int diff;

get_uptime(&uptime);
for (i = 0; i < TEST_ROUNDS; ++i)
    svc_test();
diff = systime_elapsed_us(&uptime);
printf("average kernel call time: %d.%dus\n", diff / TEST_ROUNDS,
(diff / (TEST_ROUNDS / 10)) % 10);
```

API

```
int systime_compare(SYSTIME* from, SYSTIME* to)
```

Сравнивает два результата таймера. Возвращает - 1, если `from` больше `to`, возвращает 1, `from` меньше `to`, возвращает 0, если `from` равно `to`.

```
void systime_add(SYSTIME* from, SYSTIME* to, SYSTIME* res)
```

Складывает два результат таймера, сумму в `res`.

```
void systime_sub(SYSTIME* from, SYSTIME* to, SYSTIME* res)
```

Вычитает результат from из результата to, разность записывает в res.

```
void us_to_systime(int us, SYSTIME* time)
```

Переводит микросекунды us в формат `SYSTIME`. Результат записывает в `time`.

```
void ms_to_systime(int ms, SYSTIME* time)
```

Переводит микросекунды `us` в формат `SYSTIME`. Результат записывает в `time`.

```
int systime_to_us(SYSTIME* time)
```

Переводит time формата `SYSTIME` в микросекунды.

```
int systime_to_ms(SYSTIME* time)
```

Переводит `time` формата `SYSTIME` в миллисекунды.

```
SYSTIME* systime_elapsed(SYSTIME* from, SYSTIME* res)
```

```
unsigned int systime_elapsed_ms(SYSTIME* from)

unsigned int systime_elapsed_us(SYSTIME* from)


void get_uptime(SYSTIME* uptime)
```

Возвращает время с момента запуска системы в формате `SYSTIME`. Дельта этого события может использоваться для оценки времени исполнения задач. Результат записывает в переменную `uptime`.

8.2 Программные таймеры

Программные таймеры позволяют сэкономить на аппаратных таймерах, в случае если лишние ресурсы являются важнее жесткого реалтайма аппаратных таймеров. Также они легко интегрируются в жизненный цикл процесса.
Рассмотрим небольшой пример:

```
static inline void app_timeout(APP* app)
{
    printf("app timer timeout test\n");

    // Запускаем таймер
    timer_start_ms(app->timer, 1000);
}

static inline void app_init(APP* app)
{
    .....

    // Создание программного таймера.
    app->timer = timer_create(0, HAL_APP);

    ......
}

void app()
{
    APP app;

    // Инициализируем
    app_init(&app);

    for (;;) {
        ipc_read(&ipc);

        ...........

        // Обрабатываем сообщение от тайм-аута
        case HAL_APP:
            app_timeout(&app);
            break;
```

```
        ...........

        }
        ipc_write(&ipc);
    }
}
```

API

`HANDLE timer_create(unsigned int param, HAL hal)`

Создает программный таймер, где `param` произвольный параметр пользователя, а `hal` группа `HAL`. Возвращает хэндл таймера или `INVALID_HANDLE` в случае ошибки.

`void timer_start(HANDLE timer, SYSTIME* time)`

`void timer_start_ms(HANDLE timer, unsigned int time_ms)`

`void timer_start_us(HANDLE timer, unsigned int time_us)`

Запускает таймер с таймаутом `time`, если произойдет таймаут, то процессу будет отправлено `IPC` сообщение, в котором `cmd` содержит группу `hal` (мы указали при создании таймера) с сообщением `IPC_TIMEOUT`, и в `param1` будет записан `param`, который мы указали при создании таймера.

`void timer_stop(HANDLE timer, unsigned int param, HAL hal)`

Остановить таймер с хэндлом `timer` параметром `param` и группой `hal`.

`void timer_destroy(HANDLE timer)`

Уничтожить таймер с хэндлом `timer`.

Функции, которые должны использоваться из прерываний:

`void timer_istart(HANDLE timer, SYSTIME* time)`

`void timer_istart_us(HANDLE timer, unsigned int time_us)`

`void timer_istart_ms(HANDLE timer, unsigned int time_ms)`

`void timer_istop(HANDLE timer)`

9. Системные библиотеки

Системные библиотеки предоставляют возможность общего использования библиотек как непосредственно ядром, так и пользовательскими процессами. Системные библиотеки расположены в отдельном адресном пространстве и доступны через указатель __GLOBAL->lib. Вызовы библиотек из ядра требуют дополнительной замены __GLOBAL->heap адресного пространства процесса на адресное пространство ядра, для возврата кода ошибки. Данная операция производится при помощи макросов LIB_ENTER, LIB_EXIT. Как правило, большинство вызовов ядра уже содержит эти макросы и имеют префикс 'k' в названии.

9.1 stdlib

`void* malloc(int size)`

Выделить динамическую размером `size` память в адресном пространстве процесса (или ядра). Возвращает указатель на выделенную память или `NULL`. В последнем случае код ошибки будет записан и может быть прочитан через `get_last_error()`

`void* realloc(void* ptr, int size)`

Изменить размер ранее выделенной динамической памяти, где `ptr` указатель на ранее выделенную память, `size` размер памяти в байтах. Возвращает указатель на выделенную память или `NULL`. В последнем случае код ошибки будет записан и может быть прочитан через `get_last_error()`.

`void free(void* ptr)`

Освободить ранее выделенную динамическую память, где `ptr` указатель на ранее выделенную память.

`unsigned long atou(const char *const buf, int size)`

Преобразовать текст в число. Возвращает цифровое значение, где `buf` указатель на строку или часть строки, `size` количество байт для преобразования.

`int utoa(char* buf, unsigned long value, int radix, bool uppercase)`

Преобразовать число в текст. Возвращает количество символов, где `buf` — входной буфер, `value` — значение, `radix` — система счисления, `uppercase` — необходимость преобразования букв в верхний регистр.

`unsigned int srand()`

Инициализировать генератор случайных чисел. Возвращает случайно сгенерированное

число.

```
unsigned int rand(unsigned int* seed)
```

Возвращает случайное число от 0 до `seed`.

9.2 stdio

`STDIO` реализовано на базе потоков. В заголовке процесса хранятся хэндлы экземпляров потока, и все библиотечные функции работают непосредственно с ними (`printf, putc, getc`). Первоначальные хэндлы потоков хранятся глобальными объектами ядра. Процесс может использовать их либо, если необходимо, использовать для этих целей любой другой поток. Номера объектов ядра для `STDIO` должны быть определены в конфигурационных файлах. Сама же реализация зачастую располагается непосредственно в драйверах устройств.
`SYS_OBJ_STDOUT` — номер объекта основного хэндла потока `STDOUT SYS_OBJ_STDIN` — номер объекта основного хэндла потока `STDIN`

```
bool open_stdout()
```

```
bool open_stdin()
```

Создать экземпляр потока `stdout/stdin`, используя глобальный хэндл. Возвращает `true` в случае успеха.

```
void close_stdout()
```

```
void close_stdin()
```

Удалить экземпляр потока `stdout/stdin`.

После создания экземпляра потока stdout работа не отличается с классическим stdlib:

```
#include «userspace/stdlib.h»
...
printf("Hello, world!");
```

Реализация `stdout` уровня ядра принципиально отличается от реализации `stdout` для процессов. Чтобы ядро могло отправлять отладочную информацию, процесс должен зарегистрировать каллбэк-функцию. Эта функция является вторым исключением (первое — системный таймер), при котором супервизор напрямую вызывает код приложения, поэтому здесь также должна быть соблюдена особая осторожность при написании. В целях безопасности функция регистрации также блокируется после первого вызова. Функция каллбэк должна иметь возможность работы при запрещенных прерываниях для отладки системных исключений.
Прототип функции для организации stdout ядра.

```
typedef void (*STDOUT)(const char *const buf, unsigned int size,
void* param)
```
`buf`: буфер символов для отправки;
`size`: длина буфера;
`param`: параметр, специфичный для реализации.

Регистрации функции-каллбэка. Может быть вызвана только один раз после запуска системы.

```
void setup_dbg(STDOUT stdout, void* param);
```
`stdout`: функция-каллбэк;
`param`: параметр, специфичный для реализации. Будет возвращен при вызове каллбэка.
Само же использование функции практически не отличается от использования в процессах:

```
#include «kernel/dbg.h»
...
printk("Hello from kernel!");
```

Следует также отметить, что для реализации форматирования используется одна и та же библиотечная функция и дублирование кода не производится.

API

```
void format(const char *const fmt, va_list va, STDOUT
write_handler, void* write_param)
```

Следующие функции можно использовать только в контексте пользовательских процессов:

```
void printf(const char *const fmt, ...)
```

Произвести форматирование строки и отправить в `stdout`. Список аргументов соответствует стандарту языка Си.

```
void sprintf(char* str, const char *const fmt, ...)
```

Аналогично предыдущему, но отправить форматирование в выходную строку str.

```
void puts(const char* s)
```

Отправить строку в `stdout`.

```
void putc(const char c)
```

Отправить символ в `stdout`.

```
char getc()
```

Прочитать символ из `stdin`.

```
char* gets(char* s, int max_size)
```

Прочитать строку из stdin.

Следующие функции можно использовать вне контекста пользовательских процессов:

```
void printd(const char *const fmt, ...)
```

Произвести форматирование строки и отправить в stdout. Список аргументов соответствует стандарту языка Си.

```
void iprintd(const char *const fmt, ...)
```

Аналогична предыдущей, но должна использоваться из прерываний.

```
void printk(const char *const fmt, ...)
```

Произвести форматирование строки и отправить в stdout. Используется при отладке ядра ОС.

9.3 Кольцевые буферы

Кольцевые буферы представляют собой следующую структуру:

```
typedef struct {
    unsigned int head, tail, size;
} RB;
```

`head`: голова.
`tail`: хвост.
`size`: размер буфера.

Библиотека кольцевых буферов в `REx` реализует не хранение определенного типа данных, а хранения индексов некоего массива данных. Прикладной процесс самостоятельно решает, какие конкретно данные хранятся в массиве.

API

```
void rb_init(RB* rb, unsigned int size)
```

Функция инициализирует буфер `rb` с количеством элементов `size`.

```
void rb_clear(RB* rb)
```

Очищает буфер.

```
bool rb_is_empty(RB* rb)
```

Проверяет, является ли буфер пустым.

```
bool rb_is_full(RB* rb)
```

Проверяет, заполнен ли буфер полностью.

```
unsigned int rb_put(RB* rb)
```

Добавляет элемент в буфер, возвращает индекс.

```
unsigned int rb_get(RB* rb)
```

Вычитывает элемент из буфера, возвращает индекс.

```
unsigned int rb_size(RB* rb)
```

Возвращает количество занятого пространства в буфере.

```
unsigned int rb_free(RB* rb)
```

Возвращает количество свободно пространства в буфере.

9.4 Двусвязные списки

Двусвязный список состоит из элементов данных, каждый из которых содержит ссылки как на следующий, так и на предыдущий элемент. Но в REx они напоминают кольцевой буфер, потому что в первом (головном) элементе списка ссылка на предыдущий элемент указывает на последний элемент этого списка, и, наоборот, в последнем элементе списка указатель на следующий элемент указывает на первый элемент этого списка. Двусвязные списки представляют собой следующую структуру:

```
typedef struct _DLIST {
    struct _DLIST * prev;
    struct _DLIST * next;
} DLIST;
```

prev - предыдущий элемент списка. next - следующий элемент списка.

Инициализация

С помощью функции `dlist_add_head` можно добавить элемент списка в начало списка (установить как головной).

Циклический перебор элементов

В REx доступна энумерация списков.

```
......

DLIST_ENUM de;
DLIST* cur;

.......

// Инициализация энумератора
dlist_enum_start((DLIST**)&__KERNEL->processes, &de);

// Получить следующий элемент
while (dlist_enum(&de, (DLIST**)&cur))
    if (kprocess_to_save->base_priority < cur->base_priority)
    {
        dlist_add_before((DLIST**)&__KERNEL->processes,
            DLIST*)cur, (DLIST*)kprocess_to_save);
        found = true;
        break;
    }
```

API

```
void dlist_clear(DLIST** dlist)
```

Очистить список.

```
bool is_dlist_empty(DLIST** dlist)
```

Функция проверяет список на пустоту.

```
void dlist_add_head(DLIST** dlist, DLIST* entry)
```

Установить головной элемент списка, где entry элемент, dlist список данных.

```
void dlist_add_tail(DLIST** dlist, DLIST* entry)
```

Добавить элемент entry в конец списка dlist.

```
void dlist_add_before(DLIST** dlist, DLIST* before, DLIST* entry)
```

Добавить новый элемент entry в список dlist перед элементом entry.

```
void dlist_add_after(DLIST** dlist, DLIST* after, DLIST* entry)
```

Добавить новый элемент entry в список dlist после элемента entry.

```
void dlist_remove_head(DLIST** dlist)
```

Удалить головной элемент списка dlist.

```
void dlist_remove_tail(DLIST** dlist)
```

Удалить один элемент в конце списка dlist.

```
void dlist_remove(DLIST** dlist, DLIST* entry)
```

Удалить элемент entry из списка dlist.

```
void dlist_next(DLIST** dlist)
```

Устанавливает следующий элемент списка как головной.

```
void dlist_prev(DLIST** dlist)
```

Устанавливает предыдущий элемент списка как головной.

```
void dlist_enum_start(DLIST** dlist, DLIST_ENUM* de)
```

Подготавливает список данных dlist к энумерации. Энумератор будет сохранен в переменную de.

```
bool dlist_enum(DLIST_ENUM* de, DLIST** cur)
```

Получить следующий элемент из списка.

```
void dlist_remove_current_inside_enum(DLIST** dlist, DLIST_ENUM*
```

```
de, DLIST* cur)
```

Удалить элемент списка во время энумерации. Необходимо использовать, чтобы не нарушить дальнейшую энумерацию после удаления.

```
bool is_dlist_contains(DLIST** dlist, DLIST* entry)
```

Проверка списка на принадлежность элемента entry.

9.5. Динамические массивы (array)

Динамическим называется массив, размер которого, при необходимости, может меняться во время исполнения программы. С целью уменьшения фрагментации памяти, часть данных динамического массива можно зарезервировать на стадии создания.

Создание и работа с массивами

Для создания нового массива нужно воспользоваться функцией `array_create`. Прототип функции `ARRAY* array_create(ARRAY** ar, unsigned int data_size, unsigned int reserved)`. Данная функция в качестве первого аргумента принимает указатель на неинициализированную переменную типа `ARRAY`, в качестве второго аргумента принимает размер элемента в байтах, а в качестве третьего аргумента принимает количество элементов для которых будет зарезервирована память. Результатом работы фукции будет указатель на массив. Для уничтожения массива используется функция `array_destroy`.

Добавить новый элемент можно с помщью функций `array_append, array_inser`t. Первая функция добавляет элемент в конец массива и возвращает указатель на добавленный элемент, а вторая добавляет элемент по индексу.

Удалить элемент из массива можно с помощью функции `array_remove`, а для удаления всех элементов используется функция `array_clear`. Хотелось бы обратить внимание, что при использовании данных функций не происходит реального высвобождения памяти, а только удаляются индексы элементов. Для того что бы освободить память, другими словами, сжать массив, нужно воспользоваться функцией `array_squeeze`.

API

```
typedef struct _ARRAY ARRAY
```

```
ARRAY* array_create(ARRAY** ar, unsigned int data_size, unsigned
int reserved)
```

Функция создает новый массив. В качестве аргументов принимает неинициализированный

указатель ar типа ARRAY*, размер элемента data_size массива, а так же резервирует место в памяти под определенное количество элементов reserved массива.

```
void array_destroy(ARRAY** ar)
```

Уничтожить массив.

```
void* array_at(ARRAY* ar, unsigned int index)
```

Получить указатель на элемент массива по индексу.

```
unsigned int array_size(ARRAY* ar)
```

Получить количество элементов в массиве.

```
void* array_append(ARRAY** ar)
```

Добавить элемент в конец массива. Функция вернет указатель на добавленный элемент.

```
void* array_insert(ARRAY** ar, unsigned int index)
```

Добавить элемент по индексу. Функция вернет указатель на добавленый элемент.

```
ARRAY* array_clear(ARRAY** ar)
```

Очистить массив.

```
ARRAY* array_remove(ARRAY** ar, unsigned int index)
```

Удалить элемент из массива по индексу.

```
ARRAY* array_squeeze(ARRAY** ar)
```

Сжимает массив до фактического количества элементов освобождая всю зарезервированную память под нужны ОС.

9.6. Однотипные объекты (so)

Однотипные объекты (далее по тексту SO) по своей сути очень похожи на массивы, но с одним небольшим отличием. Каждый элемент в SO имеет однозначную идентификацию в

виде хэндла. Это сделано для того, что бы пользователь мог безопасно передавать между процессами ссылку на объект/данные. Ссылкой в данном случае именуется хэндл, а объектом элемент массива однотипных объектов. А так же, если мы используем простой массив и мы передали в качестве указателя индекс элемента другому процессу-приемнику, и, если, по каким то причинам элементы в исходном массиве сдвинутся (например, будет удален первый элемент), то, процесс-приемник будет иметь не актуальный индекс, что, в принципе, может привести к непредсказуемому поведению процесса или системы в целом.

Создание и работа с массивом однотипных объектов

Создание `SO` очень похоже на создание массива. Для создания объекта `SO` нужно использовать функцию `so_create`, а для уничтожения используется функция `so_destroy`. Для добавления нового элемента используется функция `so_allocate`, которая вернет хэндл на добавленный элемент, а чтобы удалить элемент, нужно воспользоваться функцией `so_free`.

В `SO` объекты хранятся не в строго определенной последовательности, как в случае с массивами (будь то статические или динамические), и, вследствие этого, хэндлы, указывающие на элементы в `SO`, не принадлежат к строго возрастающей последовательности. Из-за этого невозможно последовательно получить очередной объект `SO` через цикл `for`, а использовать специальные функции. К ним относятся `so_first` и `so_next`. Функция `so_first` возвращает хэндл на первый элемент `SO`, а функция `so_next` возвращает хэндл на следующий элемент после элемента `prev`. Когда будет достигнут конец массива функция `so_next` вернет `INVALID_HANDLE`. Небольшой пример:

```
HANDLE el = so_first(so);
do {
    printf("%d", el);
} while((el = so_next(so, el)) != INVALID_HANDLE);
```

API

```
SO* so_create(SO* so, unsigned int data_size, unsigned int
reserved)
```

Функция создает новый массив. В качестве аргументов принимает неинициализированный указатель `ar` типа `ARRAY*`, размер элемента `data_size` массива, а так же резервирует место в памяти под определенное количество элементов `reserved` массива.

```
void so_destroy(SO* so)
```

Уничтожить SO.

HANDLE so_allocate(SO* so)

Добавить элемент в SO. В качестве результата вернет хэндл добавленного элемента.

bool so_check_handle(SO* so, HANDLE handle)

Проверка принадлежит ли handle одному из элементов SO, если да, то вернет true, в противном случае вернет false.

void so_free(SO* so, HANDLE handle)

Удалить элемент из массива. где so - массив, handle - хэндл элемента.

void* so_get(SO* so, HANDLE handle)

Получить элемент по хэндлу. где so - массив, handle - хэндл объекта. В качестве результата вернет указатель на элемент.

HANDLE so_first(SO* so)

Получить хэндл первого элемента массива, в случае ошибки вернет INVALID_HANDLE.

HANDLE so_next(SO* so, HANDLE prev)

Получить следующий элемент после элемента. где so - массив, prev - хэндл предыдущего элемента. В качестве результата вернет хэндл элемента, в случае достижения конца массива - INVALID_HANDLE.

unsigned int so_count(SO* so)

Получить количество элементов.

10. Kernel config

Конфигурирование параметров ядра происходит в файле kernel_config.h, который необходима скопировать из каталога шаблонов в рабочий каталог проекта.

#define KERNEL_DEBUG {0|1} — вывод отладочной информации уровня ядра в

консоль. Обязательный минимум при отладке ядра ОС. Если данная опция будет отключена, то информация о произошедших системных исключениях, информация об ошибках работы системы или некорректного поведения не будет выведена.

`#define KERNEL_HANDLE_CHECKING {0|1}` — проверка системных хэндлов на валидность во время вызовов супервизора. При включенном режиме `KERNEL_DEBUG_MODE` будет произведен вывод сообщения пользователю, в противном случае произойдет ошибка выполнения `ERROR_INVALID_MAGIC`.

`#define KERNEL_MARKS {0|1}` — разберемся немного с хэндлами объектов. С точки зрения пользовательских процессов хэндл представляет собой некоторое число типа unsigned int, но с точки зрения ОС хэндл — это адрес памяти, который указывает на динамическую структуру данных. И может возникнуть такая ситуация, что сам хэндл валиден, но то место в памяти, куда ссылается хэндл (с точки зрения ОС) уже занято другим объектом или какими угодно данными, а если ядро ОС обратится по данному адресу, то произойдет критическая ошибка. Но если будет включена данная опция, то у всех системных объектах будет выставлена определенная magic отметка. А при работе с этими объектами через системные вызовы они будут проверены ядром ОС на соответствие отметки. И в случае совпадения пользователю будет выведено сообщение `INVALID MAGIC` с последующей остановкой ОС, но при том условии, что будет включена опция `KERNEL_DEVELOPER_MODE`, в противном случае будет отправлена ошибка `ERROR_INVALID_MAGIC`.

`#define KERNEL_RANGE_CHECKING {0|1}` — помечает динамически выделенные блоки определенным маркером в начале и в конце блока. Данная опция позволяет упростить обнаружение утечек памяти.

`#define KERNEL_ADDRESS_CHECKING {0|1}` — во время вызовов супервизора будет производиться проверка адресов передаваемых переменных на их принадлежность процессу. При включенном режиме `KERNEL_DEBUG_MODE` будет произведен вывод сообщения пользователю, в противном случае произойдет ошибка выполнения `ERROR_ACCESS_DENIED`.

`#define KERNEL_PROFILING {0|1}` — включение этой опции позволяет контролировать целостность системных объектов. Представим ситуацию, что стек ОС вышел за допустимые границы, и данные стека начали записываться в основной пул ОС, в котором хранятся все системные объекты, — данные в пуле будут повреждены. Эта опция не предотвращает сложившуюся ситуацию, но может существенно упростить отладку.

`#define KERNEL_TIMER_DEBUG {0|1}` — данная опция используется при разработке и отладке драйверов таймера. Рассмотрим нормальную работу системного таймера. После запуска системы будет проинициализирован высокоточный системный таймер в режиме free run (интервал больше 1 секунды, по умолчанию 2 секунды), и запущен таймер `RTC`. После срабатывания таймера `RTC` будет перезапущен высокоточный таймер снова в режиме `free run`, если нет никаких системных событий или же перезапущен с интервалом меньше 1

секунды, если есть какое-либо системное событие, такое поведение и будет считаться нормальным режимом работы. Но если таймер RTC не работает или настроен неправильно, то у нас и не будет ежесекундного импульса, который перезапустит высокоточный таймер в режиме free run, и при включенной данной опции пользователю будет выдано сообщение о том, что отсутствует секундный импульс или HPET настроен неверно.

#define KERNEL_DEVELOPER_MODE {0|1} — вывод максимально возможного количества отладочных сообщений при отладке ядра, а также в случае возникновения критических ситуаций, когда может произойти автоматическая перезагрузка системы, данная опция остановит работу всей системы до ее перезагрузки вручную. В релизе данная опция должна быть отключена.

#define KERNEL_IPC_DEBUG {0|1} — вывод отладочных сообщений, связанных с IPC сообщениями. Выводит такую информацию как переполнение очереди IPC сообщений процесса и при возникновении dead lock ситуаций.

#define KERNEL_PROCESS_STAT {0|1} — разрешает сбор и вывод статистической информации о работе процессов.

Вывод статистической информации во время работы ядра. Например, вывод информации процессах после запуска системы. Их количество, имя, количество занимаемой памяти и т.д.

#define KERNEL_OBJECTS_COUNT {1..n} — количество объектов ядра, которые по сути своей представляют глобальные хэндлы, зарегистрированные в ядре и доступны всем процессам для чтения. В основном, это как минимум 2 объекта. Монолитный драйвер core и стандартный поток (если пользователю нужно получать сообщения в консоль).

#define KERNEL_IPC_COUNT {1..n} — размер буфера IPC сообщений у процесса.

11. Пользовательские библиотеки

11.1. GPIO

С точки зрения защиты памяти, реализация работы с прерываниями и регистрами должна быть выделена в процесс драйвера. Однако, существуют задачи, в которых требуется управление периферией непосредственно через GPIO в режиме bitbang. Например, это медленная периферия или непосредственная работа с шинами адреса и данных без аппаратной поддержки со стороны микроконтроллера. Решение по этому вопросу — предоставлять или нет прямой доступ к периферии из приложения — лежит полностью на пользователе.
Библиотека GPIO предоставляет базовый режим управления пинами микроконтроллера в ограниченном наборе режимов, необходимом для аппаратной абстракции. В общем случае следует устанавливать режим максимальной производительности, оставляя возможность более тонкой настройки через драйвер.

```
typedef enum {
    GPIO_MODE_OUT = 0,
```

```
    GPIO_MODE_IN_FLOAT,
    GPIO_MODE_IN_PULLUP,
    GPIO_MODE_IN_PULLDOWN
} GPIO_MODE;
```

`GPIO_MODE_OUT` — пин сконфигурирован на вывод.
`GPIO_MODE_IN_FLOAT` — пин сконфигурирован на вход, подтяжка не используется.
`GPIO_MODE_IN_PULLUP` — пин сконфигурирован на вход, подтяжка к логической единице.
`GPIO_MODE_IN_PULLDOWN` — пин сконфигурирован на вход, подтяжка к логическому нулю.

```
void gpio_enable_pin(unsigned int pin, GPIO_MODE mode)
```

Включает ножку в одном из режимов. Пример:

```
gpio_enable_pin(C9, GPIO_MODE_OUT);
```

```
void gpio_disable_pin(unsigned int pin)
```

Отключает ножку. Пример:

```
gpio_disable_pin(C9)
```

```
void gpio_enable_mask(unsigned int port, GPIO_MODE mode, unsigned
int mask)
```

Включает ножки по маске, где `port` — номер порта ножек, `mode` — режим открытия, `mask` — маска. Пример:

```
gpio_enable_mask(GPIO_PORT_A, GPIO_MODE_OUT, 0xff)
```

В данном случае будут включены ножики с `A0` по `A7`, потому что число `0xff` в бинарном виде равно `0000000011111111`.

```
void gpio_disable_mask(unsigned int port, unsigned int mask)
```

Отключает ножки по маске, где `port` — номер порта ножек, `mask` — маска.

```
void gpio_set_pin(unsigned int pin)
```

Переключает состояние ножки в логическую единицу.

```
void gpio_set_mask(unsigned int port, unsigned int mask)
```

Переключает состояние ножек в логическую единицу по маске, где `port` — номер порта ножек, `mask` — маска.

```
void gpio_reset_pin(unsigned int pin)
```

Переключает состояние ножки в логический ноль.

```
void gpio_reset_mask(unsigned int port, unsigned int mask)
```

Переключает состояние ножек в логический ноль по маске, где `port` — номер порта ножек , `mask` — маска.

```
bool gpio_get_pin(unsigned int pin)
```

Считываем значение с ножки.

```
unsigned int gpio_get_mask(unsigned int port, unsigned int mask)
```

Считываем значение с ножек по маске, где `port` — номер порта ножек , `mask` — маска.

`void gpio_set_data_out(unsigned int port, unsigned int wide)`
Переключает ножки в режим вывода по маске, где `port` — номер порта ножек , `mask` — маска.

`void gpio_set_data_in(unsigned int port, unsigned int wide)`
Переключает ножки в режим ввода по маске, где `port` — номер порта ножек , `mask` — маска.

11.2. Работа со временем (time)

Структуры и функции, представленные в данном разделе, похожи на стандарт `POSIX`.

```
typedef struct {
    long day;
    unsigned int ms;
} TIME;
```

`day`: количество дней от рождества Христова, может быть отрицательным.
`ms`: количество миллисекунд с начала дня.

Тип `TIME` похож на `POSIX` тип `time_t` (который хранит количество секунд начиная с эпохи `UNIX`), но данные хранятся не в одной 64-битной переменной, а в структуре, которая занимает те же 64-бита. Обусловлено это тем, что `ARM` процессоры (в большинстве своем) не поддерживают аппаратную работу с 64-битными типами данными и для того чтобы ее задействовать, необходимо установить специальные библиотеки которые могут занимать несколько килобайт, а так же это снижает производительность. С другой стороны, нам нет необходимости проводить излишние операции деления для выделения количества дней, как это нужно делать с `POSIX time_t`.

Все внутреннее время хранится в UTC.

API

`bool is_leap_year(long year)`

Проверяет является ли год `year` високосным.

`unsigned short year_month_max_day(long year, unsigned short mon)`

Возвращает количество дней в месяце `mon` определенного года `year`.

`TIME* mktime(struct tm* ts, TIME* time)`

Конвертировать данные из `struct tm` во внутренний тип `TIME`.

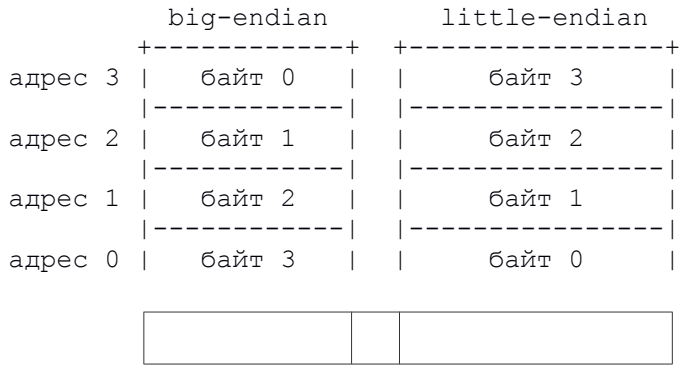
`struct tm* gmtime(TIME* time, struct tm* ts)`

Конвертирует данные из внутреннего типа `TIME` в `struct_tm`.

11.3. Конверсия представления чисел (endian)

В современной вычислительной технике и цифровых системах связи информация обычно представлена в виде последовательности байтов. В том случае, если число не может быть представлено одним байтом, имеет значение в каком порядке байты записываются в памяти компьютера или передаются по линиям связи. Часто выбор порядка записи байтов произволен и определяется только соглашениями.
В принципе достаточно знать только два вида порядка записи байтов:

1. `little-endian` - порядок от младшего к старшему. Этот порядок записи принят в памяти персональных компьютеров с `x86`-процессорами, в связи с чем иногда его называют интеловский порядок байт, а так же в системах с процессором `ARM`.
2. `big-endian` - Порядок от старшего к младшему. Этот порядок является стандартным для протоколов `TCP/IP`, он используется в заголовках пакетов данных и во многих протоколах более высокого уровня, разработанных для использования поверх `TCP/IP`. Поэтому, порядок байтов от старшего к младшему часто называют сетевым порядком байтов.



`unsigned short be2short(const uint8_t* be)`
Конвертировать массив байтов `be` из big endian в little endian `unsigned short`.

`unsigned int be2int(const uint8_t* be)`

Конвертировать массив байтов `be` big endian последовательности к `unsigned int` little endian последовательности.

`void short2be(uint8_t* be, unsigned short value)`

Конвертировать `value` из little-indian в массив байтов `be` big-endian.

`void int2be(uint8_t* be, unsigned int value)`

Конвертировать `value` из little-indian в массив байтов `be` big-endian.

11.4. Работа со строками (conv, strings)

`int hex_decode(char* text, uint8_t* data, unsigned int size_max)`
Декодировать шестнадцатеричную строку в массив данных, где `text` - декодируемая строка, `size_max` - количество декодируемых символов, `data` - массив данных.

`void hex_encode(uint8_t* data, unsigned int size, char* text)`

Закодировать массив данных в шестнадцатеричную строку, где `data` массив данных, `size` - количество шифруемых элементов, `text` - результирующая строка.

`unsigned int utf8_char_len(const char* utf8)`

Получить количество байт в символе, представленном в `UTF-8` кодировке.

`uint32_t utf8_to_utf32(const char* utf8)`

Конвертировать символ из `UTF-8` в кодировку `UTF-32`.

`unsigned int utf8_len(const char* utf8)`

Получить количество символов в UTF-8 строке.

11.5. Графические библиотеки (cavas, graphics, font)

11.5.1. Холст (canvas)

Холст это объект данных, в котором рисуются графические примитивы, такие как, например, линии и шрифты.

Создание/уничтожение холста

Новый холст можно создать с помощью функции `canvas_create` или `canvas_create_io`. Различие этих функций заключается в том, что при использовании `canvas_create` новый холст создается в адресном пространстве процесса пользователя, а `canvas_create_io` создает новый блок IO в который и помещает холст. Во втором случае при использовании `canvas_create_io` процесс пользователя имеет возможность передать этот холст другому процессу, например видеодрайверу.

Для того чтобы уничтожить холст нужно воспользоваться функцией `canvas_destroy`, если холст был создан с помощью функции `canvas_create`, и `canvas_destroy_io`,

если холст был создан с помощью функции `canvas_create`.

Размер холста и очистка

Размер холста в процессе работы может быть изменен с помощью функции `canvas_resize`, при этом размер данных нового холста не должен быть больше первоначального холста. Данная функция используется для отрисовки спрайтов на основной холст видеодрайвера, когда размеры спрайтов могут отличаться, для уменьшения фрагментации. Так же содержимое холста можно очистить с помощью функции `canvas_clear`.

API

```
CANVAS* canvas_create(unsigned short width, unsigned short height, unsigned short bits_per_pixel)
```

Создает новый холст для рисования, где - `width` - ширина холста, `height` - высота холста, `bits_per_pixel` — количество цветовых битов на пиксель.

```
IO* canvas_create_io(unsigned short width, unsigned short height, unsigned short bits_per_pixel)
```

Создает новый холст для рисования и добавляет его в блок IO, где - `width` - ширина холста, `height` - высота холта, `bits_per_pixel` — количество цветовых битов на пиксель.

```
bool canvas_resize(CANVAS* canvas, unsigned short width, unsigned short height, unsigned short bits_per_pixel)
```

Изменить характеристики холста, где - `width` - ширина холста, `height` - высота холста, `bits_per_pixel` — количество цветовых битов на пиксель. Вернет `true` в случае успеха и `false` в случае неудачи.

```
void canvas_clear(CANVAS* canvas)
```

Очистить содержимое холста.

```
void canvas_destroy(CANVAS* canvas)
```

Уничтожить холст.

```
void canvas_destroy_io(IO* io)
```

Уничтожить блок IO, если холст был создан с помощью функции `canvas_create_io`.

11.5.2. Графические примитивы (graphics)

Библиотека предназначена для отрисовки графических примитивов на холсте. Например таких как: точки, линии и прямоугольники.

Тип точка:

```
typedef struct {
    unsigned short x, y;
} POINT;
```

`x`, `y` - координаты точки на холсте.

Тип прямоугольник:

```
typedef struct {
  unsigned short left, top, width, height;
} RECT;
```

`x`, `y` - координаты начала прямоугольника.
`width`, `height` - ширина, высота прямоугольника.

Режим наложения:

При рисовании можно использовать следующие режимы наложения.

`GUI_MODE_OR` - наложение поверх содержимого холста.

`GUI_MODE_XOR` - наложение поверх содержимого холста с инверсией/выделением цвета.

`GUI_MODE_AND` - наложение по пересечению.

`GUI_MODE_FILL` - наложение с перезаписью.

API

`void put_pixel(CANVAS* canvas, const POINT* point, unsigned int color)`

Нарисовать пиксель на холсте, где `canvas` - холст, `point` - координаты пикселя, `color` - цвет пикселя.

`unsigned int get_pixel(CANVAS* canvas, const POINT* point)`

Получить пиксель с холста, где `canvas` - холст, `point` - координаты пикселя.

`void line(CANVAS* canvas, const POINT* a, const POINT* b, unsigned int color)`

Нарисовать линиию, где canvas - холст, a - начальная точка, b - конечная точка, color - цвет линии.

```
void filled_rect(CANVAS* canvas, const RECT *rect, unsigned int
color, unsigned int mode)
```

Нарисовать закрашенный прямоугольник, где canvas - холст,rect - координаты и размер прямоугольника, color - цвет заливки, mode - режим наложения GUI_MODE_OR, GUI_MODE_XOR, GUI_MODE_AND, GUI_MODE_FILL.

```
void image(CANVAS* canvas, const RECT* rect, const RECT*
data_rect, const uint8_t* pix, unsigned int mode)
```

canvas: указатель на холст

rect: прямоугольник на холсте, в который нужно вписать изображение

data_rect: прямоугольник в массиве данных, откуда копировать данные

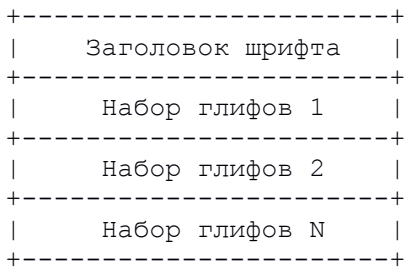
pix: массив данных

mode: режим наложения

Скопировать битовый массив данных (изображение) на холст. Более подробно о формате массива данных см. шрифты.

11.5.3. Шрифты (font)

Шрифты представляют из себя набор групп глифов. Высота каждого глифа в шрифте одинакова, ширина может отличаться.



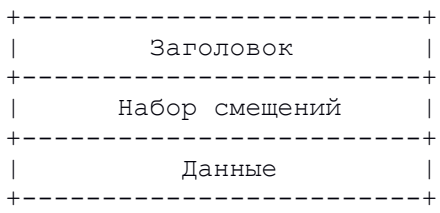
Заголовок шрифта представляет из себя следующую структуру:

```
typedef struct {
    uint16_t face_count;
    uint16_t height;
    //face 0, than other faces
} FONT;
```

face_count - количество наборов глифов.

`height` - высота шрифта.

Как правило, один набор глифов содержит в себе один алфавит определенного языка (например английский, русский). Так же в отдельном наборе могут содержаться разного рода спецсимволы. Рассмотрим формат хранения набора глифов.



Заголовок набора глифов представляет из себя следующую структуру:

```
typedef struct {
    uint32_t total_size;
    uint32_t char_offset;
    uint16_t count;
    uint16_t width;
} FACE;
```

`total_size`: размер данных в байтах.
`char_offset`: смещение первого символа в наборе относительно кодовой страницы UTF-8.
`count`: количество символов в наборе.
`width`: общая ширина набора данных в пикселях.

Для дальнейшего объяснения формата хранения глифов рассмотрим пример:

```
#define FONT_SMALL_SPECIAL_COUNT            2
#define FONT_SMALL_SPECIAL_DATA_SIZE        (13 + 1)


typedef struct {
    // Заголовок
    FACE digits_face;
    // Набор смещений
    uint16_t digits_offsets[FONT_SMALL_SPECIAL_COUNT + 1];
    // Набор данных
    uint8_t digits_data[FONT_SMALL_SPECIAL_DATA_SIZE];
} FONT_SMALL_FACE_SPECIAL;

// Высота шрифта равна 10 пикселей
{
    // общий размер набора равен
    // 14 байт минус 2 служебрых байта равно 12 байт или 96 бит.
    sizeof(FONT_SMALL_FACE_SPECIAL),
    // смещение первого глифа относительно
    // кодовой страницы
    0x000000b1,
    // количество глифов в наборе = 2 глифа.
    FONT_SMALL_SPECIAL_COUNT,
    // общая ширина набора данных
    10
}, {
    // Набор смещений
    0,    6,   10
}, {
    // Набор данных. Первый два байта отведены под служебные нужды.
```

```
    // содержит сиволы плюс-минус и знак квадрата числа.
    0x00,
    0x03, 0x00, 0x22, 0x10, 0x88, 0xfb, 0x88, 0x02, 0x03, 0xe0, 0x00, 0x00, 0x00
}
```

Набор данных содержит в себе двумерный массив (изображение) размером [10x10] пикселей, потому что, высота шрифта равна 10 и общая ширина равна 10 пикселям. Количество бит для хранения одного пикселя равно 1(монохромное изображение).
Заполнение массива (изображения) происходит последовательно слева направо, сверху вниз.
Переведем набор данных в бинарный вид:

```
0x03 = 0000:0011, 0x00 = 0000:0000, 0x22 = 0010:0010,
0x10 = 0001:0000, 0x88 = 1000:1000, 0xfb = 1111:1011,
0x88 = 1000:1000, 0x02 = 0000:0010, 0x03 = 0000:0011,
0xe0 = 1110:0000, 0x00 = 0000:0000, 0x00 = 0000:0000,
0x00 = 0000:0000.
```

Заполним матрицу [10x10] битами:

```
   1 2 3 4 5 6 7 8 9 0
  +-+-+-+-+-+-+-+-+-+-+
1 |0|0|0|0|0|0|1|1| | | <- первые два байта
2 | | | | | | | |#| |    остальные байты
3 | | |#| | | | |#| | |    0 = пробел, 1 = #
4 | | |#| | | |#| | | |    один бит = одна ячейка
5 |#|#|#|#|#| |#|#|#| |
6 | | |#| | | | | | | |
7 | | |#| | | | | | | |
8 |#|#|#|#|#| | | | | |
9 | | | | | | | | | | |
0 | | | | | | | | | | |
  +-+-+-+-+-+-+-+-+-+-+
```

Ширина первого символа равна 6 пикселей ширина второго равна 4 пикселя. Это вычисляется из набора смещений 6 - 0 = 6 пикселей, 10 - 6 = 4 пикселя.

Вывод глифов

Для вывода глифов на холст можно использовать следующие функции: `font_render_glyph`, `font_render_char`, `font_render_text`. Работа данных функций схожа между собой (более подробно смотрите API). Печать символа по коду utf-8 или по порядковому номеру в наборе глифов, в определенной точке холста. Так же имеется еще одна функция `font_render`, которая печатает строку символов вписанную в прямоугольник с последующим выравниванием текста в этом прямоугольнике.
Выравнивание может быть:

`FONT_ALIGN_LEFT` - по левому краю, `FONT_ALIGN_RIGHT` - по правому краю, `FONT_ALIGN_HCENTER` - горизонтально по центру, `FONT_ALIGN_TOP` - по верхнему краю, `FONT_ALIGN_BOTTOM` - по нижнему краю, `FONT_ALIGN_VCENTER` - вертикально по центру, `FONT_ALIGN_CENTER` - горизонтально и вертикально по центру.

API

`unsigned short font_get_glyph_width(FACE* face, unsigned short num)`

Получить ширину глифа в пикселях `face` с порядковым номером `num`, где `face` - набор глифов, `num` - порядковый номер глифа в наборе `face`.

```
void font_render_glyph(CANVAS* canvas, const POINT* point, FACE*
face, unsigned short height, unsigned short num)
```

Отрисовать глиф на холсте, где `canvas` - холст, `point` - координаты отрисовки, `face` - набор глифов, `height` - высота шрифта в пикселях, `num` - порядковый номер глифа в наборе `face`.

```
unsigned short font_get_char_width(const FONT *font, const char*
utf8)
```

Получить ширину глифа по `utf8` символу, где `font` - набор шрифтов, `utf8` - символ в кодировке UTF-8.

```
void font_render_char(CANVAS* canvas, const POINT* point, const
FONT *font, const char* utf8)
```

Отрисовать глиф на холсте, где `canvas` - холст, `point` - координаты отрисовки, `font` - набор шрифтов, `utf8` - символ в кодировке UTF-8.

```
unsigned short font_get_text_width(const FONT *font, const char*
utf8)
```

Получить ширину строки в пикселях, где `font` - набор шрифтов, `utf8` - строка символов в кодировке UTF-8.

```
void font_render_text(CANVAS* canvas, const POINT *point, const
FONT *font, const char* utf8)
```

Отрисовать текст на холсте, где `canvas` - холст, `point` - координаты отрисовки, `font` - набор шрифтов, `utf8` - строка символов к кодировке UTF-8.

```
void font_render(CANVAS* canvas, const RECT* rect, const FONT
*font, const char* utf8, unsigned int align)
```

Отрисовать текст вписанный в прямоугольник с выравниванием, где `canvas` - холст, `rect` - прямоугольник, `font` - набор шрифтов, `utf8` - строка символов в кодировке UTF-8, `align` - режим выравнивания.

11.6. Прикладные криптографические библиотеки

11.6.1. sha1, sha256

Пример использования:

```
SHA1_CTX sha1;
uint8_t hash_in[SHA1_BLOCK_SIZE];

...........

sha1_init(&sha1); // Инициализация
sha1_update(&sha1, io_data(core->sdmmc.io), core->sdmmc.total); // Добавление
данных для хеширования.
// запись хеша в переменную hash_in или core->sdmmc.hash
sha1_final(&sha1, core->sdmmc.state == SDMMC_STATE_VERIFY ? hash_in : core-
>sdmmc.hash);
```

API

`void sha1_init(SHA1_CTX *ctx)`

Инициализация, где `ctx` — контекст.

`void sha1_update(SHA1_CTX *ctx, const BYTE data[], size_t len)`

Добавление данных для хеширования, где `ctx` - контекст, `data` - массив данных, `len` - количество шифруемых элементов.

`void sha1_final(SHA1_CTX *ctx, BYTE hash[])`

Финализирование с возвращением результата в `hash`, где `ctx` - контекст, `hash` - массив байт.

`void sha256_init(SHA256_CTX *ctx)`

Инициализация, где `ctx` - контекст.

`void sha256_update(SHA256_CTX *ctx, const BYTE data[], size_t len)`

Добавление данных для хеширования, где `ctx` - контекст, `data` - массив данных, `len` - количество шифруемых элементов.

`void sha256_final(SHA256_CTX *ctx, BYTE hash[])`

Финализирование с возвращением результата в `hash`, где `ctx` - контекст, `hash` - массив байт.

11.6.2. HMAC

Функции hmac базируются на одном из алгоритмов хеширования sha1 или sha256.

API

```
void hmac_setup(HMAC_CTX* ctx, const HMAC_HASH_STRUCT*
hash_struct, void* hash_ctx, const void *key, unsigned int
key_size)
```

Установка алгоритма хеширования, `ctx` - контекст, `hash_struct` - алгоритм хеширования (`__HMAC_SHA1`, `__HMAC_SHA256`), `hash_ctx` - контекст хеш, `key` - ключ, `key_size` - размер ключа.

```
void hmac_init(HMAC_CTX* ctx)
```

Инициализация, где `ctx` — контекст.

```
void hmac_update(HMAC_CTX* ctx, const void *data, unsigned int
size)
```

Генерация хеша, где `ctx` - контекст, `data` - массив данных, `size` - количество шифруемых элементов.

```
void hmac_final(HMAC_CTX* ctx, void* hmac)
```

Запись хеша в массив `hash`, где `ctx` - контекст, `hmac` - массив байт.

11.6.3. AES

API

```
int AES_set_encrypt_key(const unsigned char *userKey, const int
bits, AES_KEY *key)
```

Подготовка round keys для шифрования, где - `userKey` - криптографический ключ, `bits` - битность ключей шифрования, `key` - round key.

```
int AES_set_decrypt_key(const unsigned char *userKey, const int
bits, AES_KEY *key)
```

Подготовка round keys для дешифрования, где `userKey` - криптографический ключ, `bits` - битность ключей шифрования, `key` - round key.

```
void AES_encrypt(const unsigned char *in, unsigned char *out,
```

```
const AES_KEY *key)
```

Функция шифрует один блок данных `in` с помощью ключа `key` и записывает его в `out`.

```
void AES_decrypt(const unsigned char *in, unsigned char *out,
const AES_KEY *key)
```

Функция дешифрует один блок данных `in` с помощью ключа `key` и записывает его в `out`.

```
void AES_cbc_encrypt(const unsigned char *in, unsigned char *out,
size_t length, const AES_KEY *key, unsigned char *ivec, const int
enc)
```

Шифрует или дешифрует набор данных `in` в режиме `CBC`, `out` - результат, `length` - размер данных в байтах, `key` - шифроключ, `ivec` - вектор инициализации (случайный набор данных), `enc` - признак шифрования `1` или дешифрования `0`.

11.6.4. Криптографическое выравнивание

API

```
unsigned int pkcs7_encode(void* m, unsigned int size, unsigned int
block_size)
```

Функция дополняет шифруемые данные `m` служебным отступом, до размера кратному `block_size`. Возвращает размер шифруемых данных с учетом отступа.

```
int eme_pkcs1_v1_15_decode(const void *em, unsigned int em_size,
void* m, unsigned int m_max)
```

```
int pkcs7_decode(void* em_m, unsigned int size)
```

Функция возвращает реальный размер расшифрованных данных, где em_m - расшифрованные данные, size - размер зашифрованных данных.

12. Драйверы

Данный раздел включает в себя базовое описание общего интерфейса для всех драйверов, вне зависимости от аппаратной платформы (HAL).

12.1. ADC

```
int adc_get(int channel)
```

Получить калиброванное значение с АЦП, где `channel` — канал.

12.2. DAC

Режимы работы ЦАП:

```
typedef enum {
    DAC_MODE_LEVEL = 0,
    DAC_MODE_WAVE,
    DAC_MODE_NOISE,
    DAC_MODE_STREAM
} DAC_MODE;
```

DAC_MODE_LEVEL - данный режим используется, когда устанавливается определенный уровень сигнала.

DAC_MODE_WAVE - режим для генерации определенной формы волны.

DAC_MODE_NOISE - режим генерации шума.

DAC_MODE_STREAM - данный режим используется при передаче потоковых данных.

Форма волны:

```
typedef enum {
    DAC_WAVE_SINE = 0, // синусоидальная
    DAC_WAVE_TRIANGLE, // треугольная
    DAC_WAVE_SQUARE // прямоугольная
} DAC_WAVE_TYPE;
```

API

`bool dac_open(int channel, DAC_MODE mode, unsigned int samplerate)`

Открыть канал `channel` ЦАП в режиме `mode` с частотой дискретизации `samplerate`

`void dac_set(int channel, SAMPLE value)`

Установить значение `value` по каналу `channel`.

`void dac_wave(int channel, DAC_WAVE_TYPE wave_type, SAMPLE amplitude)`

Отправить волну по каналу `channel` с типом сигнала `wave_type` и определенной амплитудой `amplitude`.

sys_config

`#define SAMPLE uint16_t`

Указать тип для сэмпла данных.

Так же в конфигурационном файле можно определить переменные `WAVEGEN_SINE`, `WAVEGEN_TRIANGLE`, `WAVEGEN_SQUARE` со значением 1 или 0. Если выставить 0, скажем, у переменной `WAVEGEN_SQUARE`, то драйвером не будет поддерживаться возможность генерации волны прямоугольной формы, но это действие позволит уменьшить размер готового образа системы.

12.3. EEPROM

`int eep_read(unsigned int offset, void* buf, unsigned int size)`

Считать буфер `buf` размером `size` из EEPROM начиная со смещения `offset` от начала блока EEPROM.

`int eep_write(unsigned int offset, const void* buf, unsigned int size)`
Записать буфер `buf` размером `size` из EEPROM начиная со смещения `offset` от начала блока EEPROM.

12.4. Ethernet

Режимы соединения:

```
typedef enum {
    ETH_10_HALF = 0,
    ETH_10_FULL,
    ETH_100_HALF,
    ETH_100_FULL,
    ETH_AUTO,
    ETH_LOOPBACK,
    ETH_NO_LINK,
    ETH_REMOTE_FAULT
} ETH_CONN_TYPE;
```

где `ETH_10_HALF`, `ETH_10_FULL`, `ETH_100_HALF`, `ETH_100_FULL`, `ETH_AUTO`, `ETH_LOOPBACK` - режим соединения, `ETH_NO_LINK`, `ETH_REMOTE_FAULT` - относятся к статусу соединения.

API

`void eth_set_mac(HANDLE eth, unsigned int eth_handle, const MAC* mac)`
Установить mac адрес, где `eth` - хэндл процесса, `eth_handle` — пользовательский хэндл или адрес `ethernet PHY`, `mac` - указатель на mac адрес.

`void eth_get_mac(HANDLE eth, unsigned int eth_handle, MAC* mac)`
Получить mac адрес, где `eth` - хэндл процесса, `eth_handle` — пользовательский хэндл или адрес `ethernet PHY`, `mac` - указатель на mac адрес.

`unsigned int eth_get_header_size(HANDLE eth, unsigned int eth_handle)`
Данная функция предназначена исключительно для работы с RNDIS. Протокол RNDIS в начале пакета добавляет служебные данные (заголовок). Функция возвращает размер заголовка. В физическом `ETH` не используется.

sys_config

`#define ETH_AUTO_NEGOTIATION_TIME 5000` — Количество миллисекунд для установки соединения, после которых будет сигнализировано об ошибке.
`#define ETH_DOUBLE_BUFFERING` - двойная буферизация.

12.5. Шина I2C

В REx реализована как простое чтение/запись, так и наиболее используемые расширенные варианты реализации. В частности режим адреса используется во внешних EEPROM, а режим длины встречается в смарт-картах.

Простое чтение/запись

Для чтения данных используется функция i2c_read, а для записи i2c_write.

```
         | запрос master |      ответ slave    |
 +-------+---------------+-------------+------+------+
 | старт |    sla/read   |   данные    | NACK | стоп |
 +-------+---------------+-------------+------+------+

         |         запрос master        |
 +-------+---------------+-------------+------+
 | старт |    sla/write  |   данные    | стоп |
 +-------+---------------+-------------+------+
```

Чтение/запись с указанием длины

Для чтения данных используется функция i2c_read_len, а для записи i2c_write_len.

```
         | запрос master |  ответ slave  |
 +-------+---------------+---------------+------+
 + старт |    sla/read   | len | данные  | стоп |
 +-------+---------------+-----+---------+------+


         |          запрос master        |
 +-------+---------------+---------------+------+
 + старт |    sla/write  | len | данные  | стоп |
 +-------+---------------+-----+---------+------+
```

Чтение/запись с указанием адреса

Для чтения данных используется функция i2c_read_addr, а для записи i2c_write_addr.

```
        |  запрос master  |     |  ответ slave   |
+-------+----------+------+-----+---------+------+------+
+ старт | sla/read | addr | R/S | данные  | NACK | стоп |
+-------+----------+------+-----+---------+------+------+

        |       запрос master      |
+-------+----------+------+--------+------+
+ старт | sla/write| addr | данные | стоп |
+-------+----------+------+--------+------+
```

Чтение/запись с указанием адреса и длины

Для чтения данных используется функция i2c_read_addr_len, а для записи i2c_write_addr_len.

```
        |  запрос master  |     |  ответ slave |
+-------+----------+------+-----+-----+--------+------+
+ старт | sla/read | addr | R/S | len | данные | стоп |
+-------+----------+------+-----+-----+--------+------+
```

```
        |          запрос master        |
+-------+----------+------+-----+-------+------+
+ старт | sla/write| addr | len |данные | стоп |
+-------+----------+------+-----+-------+------+
```

API

`bool i2c_open(I2C_PORT port, unsigned int mode, unsigned int speed)`

Открыть `port` в режиме `mode` (ведущий `I2C_MODE_MASTER`, ведомый `I2C_MODE_SLAVE`), где `speed` - скорость обмена.

`void i2c_close(I2C_PORT port)`

Закрыть порт.

`int i2c_read(I2C_PORT port, uint8_t sla, IO* io, unsigned int max_size)`

`int i2c_read_len(I2C_PORT port, uint8_t sla, IO* io, unsigned int max_size)`

`int i2c_read_addr(I2C_PORT port, uint8_t sla, uint8_t addr, IO* io, unsigned int max_size)`

`int i2c_read_addr_len(I2C_PORT port, uint8_t sla, uint8_t addr, IO* io, unsigned int max_size)`

Прочитать данные из устройства, где `port` - порт, `sla` - адрес устройства, `io` - блок данных в который будут записаны данные, `max_size` — максимальный размер данных. Для запросов с адресом `addr` — адрес регистра.

`int i2c_write(I2C_PORT port, uint8_t sla, IO* io)`

`int i2c_write_len(I2C_PORT port, uint8_t sla, IO* io)`

`int i2c_write_addr(I2C_PORT port, uint8_t sla, uint8_t addr, IO* io)`

`int i2c_write_addr_len(I2C_PORT port, uint8_t sla, uint8_t addr, IO* io)`

Записать данные в устройство, где `port` - порт, `sla` - адрес устройства, `io` - блок данных для записи. Для запросов с адресом `addr` — адрес регистра.

12.6. Аппаратные таймеры

При работе с аппаратными таймерами пользователю необходимо задать тип указания параметров таймера:

`TIMER_VALUE_HZ` — В герцах

`TIMER_VALUE_US` — В микросекундах

`TIMER_VALUE_CLK` — В тактах системной частоты

Для каждого канала аппаратного таймера необходимо установить тип тактования:

`TIMER_CHANNEL_GENERAL` — без привязки к внешним сигналам.

`TIMER_CHANNEL_INPUT_RISING` — по восходящему фронту.

`TIMER_CHANNEL_INPUT_FALLING` — по нисходящему фронту.

`TIMER_CHANNEL_INPUT_RISING_FALLING` — по обоим фронтам.

`TIMER_CHANNEL_PWM` — режим работы ШИМ.

`TIMER_CHANNEL_DISABLE` — закрыть канал таймера

API

```
bool htimer_open(int num, unsigned int flags)
```

Открыть аппаратный таймер, где `num` - номер таймера, `flags` - флаги которые относятся к аппаратно-зависимой части драйвера.

```
void htimer_close(int num)
```

Закрыть таймер с номером `num`.

```
void htimer_start(int num, TIMER_VALUE_TYPE value_type, unsigned int value)
```

Запустить таймер, где `num` - номер таймера, `value_type` - тип срабатывания таймера, `value` - значение по истечению которого сработает таймер.

```
void htimer_stop(int num)
```

Остановить таймер с номером `num`.

```
void htimer_setup_channel(int num, int channel, TIMER_CHANNEL_TYPE type, unsigned int value)
```

Настроить канал таймера, где `num` - номер таймера, `channel` - канал, `type` - тип, `value` — значение.

12.7. Часы реального времени (RTC)

```
TIME* rtc_get(TIME* time)
```

Получить системное время.

```
void rtc_set(TIME* time)
```

Установить системное время.

12.8. Сторожевой таймер (WDT)

```
void wdt_kick()
```

Перезапустить сторожевой таймер.

12.9. Порт UART

Конфигурация режима работы UART:

```
typedef struct {
    uint32_t baud;
    uint8_t data_bits;
    char parity;
    uint8_t stop_bits;
} BAUD;
```

`baud` - скорость передачи данных

`data_bits` - количиество бит данных: 7, 8

`parity` - проверка посылки: `N` - без бита четности, `O` - с битом проверки на четность, `E` - с битом проверки на не четность.

API

```
bool uart_open(int num, unsigned int mode)
```

Открыть UART порт с номером num. В случае успеха вернет true, в противном случае false.

```
void uart_close(int num)
```

Закрыть UART порт с номером num.

```
void uart_setup_printk(int num)
```

Настроить отладочный интерфейс ядра в порт UART с номером num.

```
void uart_setup_stdout(int num)
```

Настроить интерфейс stdout в порт UART с номером num.

```
void uart_setup_stdin(int num)
```

Настроить интерфейс stdin в порт UART с номером num.

```
void uart_set_baudrate(int num, BAUD* baudrate)
```

Задать параметры передачи данных порта UART num.

```
void uart_flush(int num)
```

Очистить буфер данных UART с номер num.

12.10. Интерфейс storage

Интерфейс storage предназначен для работы с блочными устройствами хранения информации, такими как SD/MMC, NAND и т. д. Любое устройство с блочной организацией памяти базируется на данном абстрактном интерфейсе.

Начало работы

Для начала работы необходимо открыть устройство с помощью функции storage_open. Первым параметром необходимо передать номер группы аппаратной абстракции устройства например HAL_SDMMC, второй параметр принимает хэндл драйвера который отвечает за работу с данным устройством, а третий параметр принимает пользовательский хэндл, например если мы работаем с MSC (Mass Storage Class), то хэндл будет содержать

номер USB. Другими словами пользовательский хэндл зависит от специфики конкретного устройства и его описание содержится в аппаратно зависимом интерфейсе драйвера.

Функция storage_get_media_descriptor возвращает информацию о носителе.

```
typedef struct {
    uint32_t num_sectors;
    uint32_t num_sectors_hi;
    uint32_t sector_size;
} STORAGE_MEDIA_DESCRIPTOR;
```

num_sectors - количество секторов.
num_sectors_hi - количество старших секторов.
sector_size - размер сектора в байтах.

Работа с устройством

Для чтения используется функции storage_read, storage_verify и аналогичные функции, которые работают в синхронном режиме. Для записи используются функции storage_write, storage_write_verify. Для удаления используется функция storage_erase. Стоит обратить внимание на функции с пост фиксом _verify, они предназначены для проверки данных, записи данных с проверкой и их реализация зависит от конкретного драйвера.

Так же для проверки состояния имеются функции storage_request_notify_state_change, storage_cancel_notify_state_change, storage_request_activity_notify. Первые две функции используют для опроса состояния съемных носителей информации например такие как SD/MMC. Функция storage_request_activity_notify отправляет запрос на получение уведомления активности устройства, такой как запись или чтение данных.

API

```
bool storage_open(HAL hal, HANDLE process, HANDLE user)
```

Открыть устройство, где hal - группа, process - хэндл драйвера, user - пользовательский хэндл.

```
void storage_get_media_descriptor(HAL hal, HANDLE process, HANDLE
user, IO* io)
```

Получить дескриптор устройства, где hal - группа, process - хэндл драйвера, user - пользовательский хэндл,

```
STORAGE_MEDIA_DESCRIPTOR* storage_get_media_descriptor_sync(HAL
hal, HANDLE process, HANDLE user, IO* io)
```

Аналогична предыдущей функции, но работает в синхронном режиме.

```
void storage_read(HAL hal, HANDLE process, HANDLE user, IO* io,
unsigned int sector, unsigned int size)
```

Прочитать данные с устройства, где `hal` - группа, `process` - хэндл драйвера, `user` - пользовательский хэндл, `io` - блок в который будут записаны данные, `sector` - сектор начиная с которого будут прочитаны данные, `size` - размер читаемых данных в секторах.

```
bool storage_read_sync(HAL hal, HANDLE process, HANDLE user, IO*
io, unsigned int sector, unsigned int size)
```

Аналогична предыдущей, но работает в синхронном режиме, в случае успеха вернет `true`, в противном случае вернет `false`.

```
void storage_write(HAL hal, HANDLE process, HANDLE user, IO* io,
unsigned int sector)
```

Записать данные, где `hal` - группа, `process` - хэндл драйвера, `user` - пользовательский хэндл, `io` - данных для записи, `sector` - сектор начиная с которого будут записаны данные.

```
bool storage_write_sync(HAL hal, HANDLE process, HANDLE user, IO*
io, unsigned int sector)
```

Аналогична предыдущей, но работает в синхронном режиме, в случае успеха вернет `true`, в противном случае вернет `false`.

```
void storage_erase(HAL hal, HANDLE process, HANDLE user, IO* io,
unsigned int sector, unsigned int size)
```

Стереть данные, где `hal` - группа, `process` - хэндл драйвера, `user` - пользовательский хэндл, `io` - данных для записи, `sector` - сектор начиная с которого будут стерты данные, `size` - размер стираемых данных в секторах.

```
bool storage_erase_sync(HAL hal, HANDLE process, HANDLE user, IO*
io, unsigned int sector, unsigned int size)
```

Аналогична предыдущей, но работает в синхронном режиме, в случае успеха вернет `true`, в противном случае вернет `false`.

```
void storage_verify(HAL hal, HANDLE process, HANDLE user, IO* io,
unsigned int sector)
```

Прочитать данные с проверкой, где `hal` - группа, `process` - хэндл драйвера, `user` - пользовательский хэндл, `io` - блок данных в который будут записаны проверяемые данные, `sector` - начальный сектор.

```
bool storage_verify_sync(HAL hal, HANDLE process, HANDLE user, IO*
io, unsigned int sector)
```

Аналогична предыдущей, но работает в синхронном режиме, в случае успеха вернет `true`, в противном случае вернет `false`.

```
void storage_write_verify(HAL hal, HANDLE process, HANDLE user,
IO* io, unsigned int sector)
```

Записать данные с проверкой, где `hal` - группа, `process` - хэндл драйвера, `user` - пользовательский хэндл, `io` - блок данных, `sector` - начальный сектор.

```
bool storage_write_verify_sync(HAL hal, HANDLE process, HANDLE
user, IO* io, unsigned int sector)
```

Аналогична предыдущей, но работает в синхронном режиме, в случае успеха вернет `true`, в противном случае вернет `false`.

```
void storage_request_notify_state_change(HAL hal, HANDLE process,
HANDLE user)
```

Запросить уведомление об изменении состоянии подключения устройства.

```
void storage_cancel_notify_state_change(HAL hal, HANDLE process,
HANDLE user)
```

Отменить предыдущее уведомление.

```
void storage_request_activity_notify(HAL hal, HANDLE process,
HANDLE user)
```

Запросить уведомление об активности устройства.

12.11. Pinboard

`Pinboard` — модуль программной клавиатуры на базе единичных пинов `GPIO`. Производит периодический опрос пинов, реализует программный антидребезг и генерирует события нажатия клавиатуры, такие как:

`PINBOARD_KEY_DOWN` — нажатие кнопки.

`PINBOARD_KEY_UP` — отжатие кнопки.

`PINBOARD_KEY_PRESS` — короткое нажатие.

`PINBOARD_KEY_LONG_PRESS` — длинное нажатие.

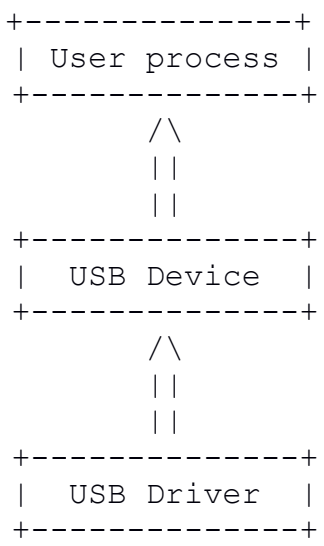
Конфигурирование кнопки происходит при помощи сообщения `IPC_OPEN`:

```
ack(pinboard, HAL_CMD(HAL_PINBOARD, IPC_OPEN), my_pin,
PINBOARD_FLAG_INVERTED | PINBOARD_FLAG_DOWN_EVENT |
PINBOARD_FLAG_UP_EVENT | PINBOARD_FLAG_PULL, 3000)
```

Производит конфигурирование пина `my_pin` в инвертированном режиме с подтяжкой и регистрацией нажатий и отжатий. Нажатие более 3 секунд считается длинным нажатием. Более подробно см. `pinboard.h`.

13. Подсистема USB

Подсистему USB схематично можно изобразить следующим образом:



1. USB Driver. Уровень аппаратного драйвера USB необходим для управления ресурсами USB. Он отвечает за выполнение таких действий как: открытие/закрытие конечных точек и запись в конечные точки; назначение логического адреса устройства USB.
2. USB Device. Программный стэк USB устройства. Включает в себя реализацию интерфейсов для работы с разными классами USB.
3. User process. Пользовательский интерфейс для работы с USB устройством.

13.1 Работа с USB устройством.

Работа с USB устройством сводится к следующей последовательности действий:

1. Создание процесса и конфигурирование USB устройства. Открытие устройства.
2. Основной цикл работы с устройством.
3. Закрытие устройства.

Описание конфигурации

Первым делом пользователю необходимо определить все дескрипторы, такие как: дескриптор устройства, дескриптор объединения (если устройство является композитным), интерфейсные дескрипторы, классовые дескрипторы (дескрипторы специфичные для класса устройства), дескрипторы конечных точек, дескрипторы строк. Дескриптор устройства USB_DEVICE_DESCRIPTOR является стандартным и описан в спецификации USB.

```
const USB_DEVICE_DESCRIPTOR __DEVICE_DESCRIPTOR = {
    sizeof(USB_DEVICE_DESCRIPTOR),
    USB_DEVICE_DESCRIPTOR_TYPE,
    0x0200,
    USB_MISCELLANEOUS_DEVICE_CLASS,
    USB_MISCELLANEOUS_COMMON_SUBCLASS,
    USB_IAD_PROTOCOL,
    ........................
};
```

После чего все остальные дескрипторы (кроме дескрипторов строк) должны быть сгруппированы в конфигурацию устройства. REx не накладывает ограничение на количество конфигураций для USB устройств, поэтому их может быть несколько. Пример конфигурации

USB устройства:

```
typedef struct {
    USB_CONFIGURATION_DESCRIPTOR configuration; // дескриптор
конфигурации
    USB_INTERFACE_ASSOCIATION_DESCRIPTOR comm_iad; // дескриптор
объединения

    // Специфичный набор интерфесов,
    // дескрипторов устройства,
    // и конечных точек
    // для данного USB класса.
    USB_INTERFACE_DESCRIPTOR comm_interface;
    CDC_HEADER_DESCRIPTOR_TYPE cdc_header;
    CDC_ACM_DESCRIPTOR_TYPE cdc_acm;
    CDC_UNION_DESCRIPTOR_TYPE cdc_union;
    uint8_t bSubordinateInterface0;
    CDC_CALL_MANAGEMENT_DESCRIPTOR_TYPE call_management;
    USB_ENDPOINT_DESCRIPTOR comm_endpoint;
    USB_INTERFACE_DESCRIPTOR data_interface;
    USB_ENDPOINT_DESCRIPTOR data_endpoints[2];
} CONFIGURATION;
```

Первым полем в конфигурации обязательно должен идти дескриптор конфигурации USB_CONFIGURATION_DESCRIPTOR, все его поля являются стандартными по спецификации USB. Следующим полем может идти дескриптор объединения, если устройство является композитным. После чего идет специфичный набор дескрипторов для данного класса USB такие как: дескрипторы интерфейса, классовые дескрипторы, дескрипторы конечных точек, специфичные для класса дескрипторы.

После чего пользователю необходимо описать дескрипторы строк, но данный шаг является необязательным.

```
const char __STRING_MANUFACTURER[] = {
    6 * 2 + 2,
    USB_STRING_DESCRIPTOR_TYPE,
    'R', 0,
    'E', 0,
    'x', 0,
    ' ', 0,
    'O', 0,
    'S', 0
};
```

Создание и назначение конфигурации

Пользователю необходимо создать процесс USB устройства с помощью функции usbd_create, которая вернет хэндл процесса. Первым параметром передается номер порта, вторым размер процесса, а третьим параметром приоритет. Для некоторых классов

USB устройства необходимо зарегистрировать первоначальную конфигурацию с помощью функции usbd_register_configuration, например для класса mass storage class.

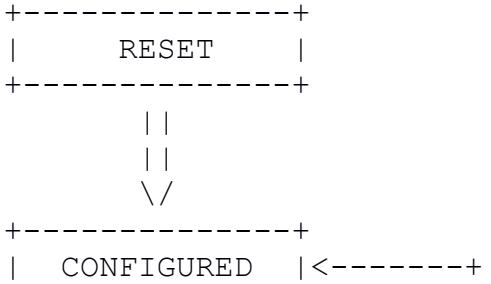
```
HANDLE usbd = usbd_create(USB_0, 900, 150);

// Оповещение
ack(usbd, HAL_REQ(HAL_USBD, USBD_REGISTER_HANDLER), 0, 0, 0);

// Регистрация конфигурации
usbd_register_const_descriptor(usbd, &__DEVICE_DESCRIPTOR, 0, 0);
usbd_register_const_descriptor(usbd, &__CONFIGURATION_DESCRIPTOR,
0, 0);
usbd_register_const_descriptor(usbd, &__STRING_MANUFACTURER, 1,
0x0409);

// Открытие устройства
ack(usbd, HAL_REQ(HAL_USBD, IPC_OPEN), USB_PORT_NUM, 0, 0);

.....
.....
.....

// Закрытие устройства
ack(usbd, HAL_REQ(HAL_USBD, IPC_CLOSE), USB_PORT_NUM, 0, 0);
```

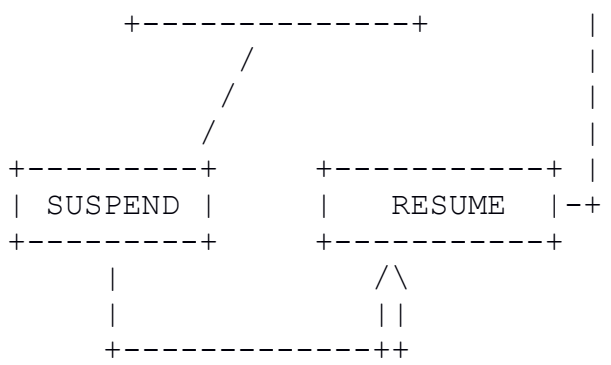
Основной цикл работы

Основной цикл работы с USB устройством в процессе пользователя сводится к приему и отправки IPC сообщений процессу USBD. USB устройство может находится в одном из следующих состояний.

1. RESET - первоначальное состояние USB устройства
2. CONFIGURED - хост завершил конфигурирование устройства.
3. SUSPEND - устройство переведено в режим низкого энергопотребления.
4. RESUME - устройство возвращено в нормальный режим работы.

Автомат состояний можно представить следующим образом:





В состоянии `RESET` устройство находится в тот момент, когда было подключено к хосту и еще не сконфигурировано, после чего хост запрашивает и устанавливает конфигурацию `USB` устройства и данное состояние является основным режимом работы. Хост может перевести устройство в режим `SUSPEND`. Из состояния `SUSPEND` хост может перевести устройство сигналом `RESUME` в состояние `CONFIGURED`, другими словами основной режим работы. Процесс `USB` устройства будет отправлять `IPC` сообщения пользовательскому процессу, которые содержат информацию о состоянии устройства. Сообщения относятся к группе `HAL_USBD`, код сообщения `USBD_ALERT`, а первым параметром содержится код состояния устройства: `USBD_ALERT_RESET`, `USBD_ALERT_SUSPEND`, `USBD_ALERT_RESUME`, `USBD_ALERT_CONFIGURED`.

Пользователю необходимо дождаться, когда устройство будет переведено в нормальный режим работы `CONFIGURED` и продолжить работу через класс устройства или через запросы `VSR`.

Основной цикл работы с `USB` устройством сводится к работе с классами устройств, которые специфичны для каждого класса и более подробно будут описаны далее в следующих главах.

API

```
HANDLE usbd_create(USB_PORT_TYPE port, unsigned int process_size,
unsigned int priority)
```

Создать процесс для работы с USB устройством, где `port` - номер USB порта, `process_size` - размер процесса, `priority` - приоритет процесса. Возвращает хэндл процесса.

```
bool usbd_register_const_descriptor(HANDLE usbd, const void* d,
unsigned int index, unsigned int lang)
```

```
bool usbd_register_descriptor(HANDLE usbd, const void* d, unsigned
int index, unsigned int lang)
```

Зарегистрировать дескриптор, где `usbd` - хэндл `USBD` процесса, `d` - указатель на дескриптор, `index` - индекс дескриптора, `lang` - код языка. В случае успеха вернет `true`. Параметры

index и lang нужно только для дескрипторов строки, во всех остальных случаях должны быть равны 0.

```
bool usbd_register_ascii_string(HANDLE usbd, unsigned int index,
unsigned int lang, const char* str)
```

Зарегистрировать дескриптор строки, где usbd - хэндл USB процесса, index - индекс дескриптора, lang - код языка, str - ANSI строка.

```
bool usbd_register_configuration(HANDLE usbd, uint16_t cfg,
uint16_t iface, const void* d, unsigned int size)
```

Задать первоначальную конфигурацию, где usbd - хэндл USB процесса, cfg - номер конфигурации, iface - номер интерфейс, d - данные инициализации, size - размер данных.

```
bool usbd_unregister_configuration(HANDLE usbd, uint16_t cfg,
uint16_t iface)
```

Отменить первоначальную конфигурацию, где usbd - хэндл USBD процесса, cfg - номер конфигурации, iface - номер интерфейса.

sys_config

#define USBD_DEBUG - вывод отладочных сообщений в стандартный поток вывода.

#define USBD_DEBUG_ERRORS - вывод ошибок в связанных с работой устройства например: ошибка конфигурирования устройства.

#define USBD_DEBUG_REQUESTS - вывод отладочных сообщений, в основном связанных с управляющими запросами к устройству.

#define USBD_DEBUG_FLOW - вывод низкоуровневых отладочных сообщений.

#define USBD_VSR - поддержка VSR запросов

#define USBD_IO_SIZE - размер блока IO в байтах, который используется в VSR запросах, а так же при чтении/записи в конечные точки.

#define USBD_CDC_ACM_CLASS - включить поддержку класса CDC.

#define USBD_RNDIS_CLASS - включить поддержку класса RNDIS.

#define USBD_HID_KBD_CLASS - включить поддержку класса HID.

#define USBD_CCID_CLASS - включить поддержку класса CCID.

`#define USBD_MSC_CLASS` - включить поддержку класса `MSC`.

`#define USB_EP_COUNT_MAX` — максимальное количество двунаправленных контрольных точек в USB устройстве

`#define USB_DEBUG_ERRORS` — включить низкоуровневую отладку драйвера USB. Рекомендуется использовать только для разработчиков драйверов USB.

`#define USB_TEST_MODE_SUPPORT` — разрешить хостом перевод USB устройства в режим низкоуровневого тестирования. Используется только при наличии серьезных аппаратных проблем и/или разработке драйвера.

13.2. Виды запросов

Запросы к USB устройству можно разделить на три вида:

1. Управляющие запросы (Открытие, закрытие, проверка статуса устройства)
2. Работа с классами устройства
3. Запросы VSR

Управляющие запросы

Примеры управляющих запросов:

```
ack(usbd, HAL_REQ(HAL_USBD, IPC_OPEN), USB_PORT_NUM, 0, 0);
ack(usbd, HAL_REQ(HAL_USBD, IPC_CLOSE), USB_PORT_NUM, 0, 0);
ack(usbd, HAL_REQ(HAL_USBD, USB_RESET), USB_PORT_NUM, 0, 0);
ack(usbd, HAL_REQ(HAL_USBD, USB_WAKEUP), USB_PORT_NUM, 0, 0);
```

Запросы к классам USB

Работы с классами `USB` устройства идет сводится к работе с хэндлам через IPC запросы/ответы к процессу `USBD`. Для маршрутизации внутри `USBD`, запросы к классам обязаны содержать пользовательский хэндл `(ipc->param1)`, сформированный через макрос `USBD_IFACE`, и/или принадлежать `HAL` группе команды `HAL_USBD_IFACE`. `USBD_IFACE` первым параметром принимает номер интерфейса, а вторым его подтип.

```
// Запрос специфичный для CDC
// Получить хэндл потока на передачу HAL_REQ(HAL_USBD_IFACE,
IPC_GET_TX_STREAM)
// С номером порта 0 USBD_IFACE(0, 0)
tx_stream = get_handle(app->usbd, HAL_REQ(HAL_USBD_IFACE,
IPC_GET_TX_STREAM), USBD_IFACE(0, 0), 0, 0);
```

VSR

Работа с `VSR` сводится к отправке и приему сообщений в основном цикле работы пользовательского процесса. Данные передаются в блоке `IO`, на стеке которого помещен

SETUP пакет.

```
static inline void app_usb_vsr(APP*app, IO* io) {
  FW_USB* fw_usb;
  SETUP* setup = io_stack(io);
  switch (setup->bRequest) {
  case USB_VSR_GET_LAST_ERROR:
    printf("USB VSR: Get last error\n");
    io_data_write(io, &app->rc.error, sizeof(uint32_t));
    break;
  case USB_VSR_GET_CONFIGURATION:
    printf("USB VSR: Get configuration\n");
    fw_usb = io_data(io);
    ....
    break;
  default:
    error(ERROR_NOT_SUPPORTED);
    break;
  }
}

void app_usb_request(APP* app, IPC* ipc) {
  switch (HAL_ITEM(ipc->cmd)) {
  case USBD_ALERT:
    switch (ipc->param1) {
    case USBD_ALERT_CONFIGURED:
        app_usb_enable(app);
        break;
    case USBD_ALERT_RESET:
        app_usb_disable(app);
        break;
    }
    break;
  case USBD_VENDOR_REQUEST:
    app_usb_vsr(app, (IO*)ipc->param2);
    break;
    ......
    ......
    ......
  default:
    error(ERROR_NOT_SUPPORTED);
  }
}
```

13.3 Разбор конфигурационных дескрипторов

Данная библиотека используется как классами USB устройств, так и может потребоваться самому пользователю.

API

```
USB_INTERFACE_DESCRIPTOR* usb_get_first_interface(const
```

```
USB_CONFIGURATION_DESCRIPTOR* cfg)
```

Получить указатель на первый дескриптор интерфейса, где `cfg` - дескриптор конфигурации.

```
USB_INTERFACE_DESCRIPTOR* usb_get_next_interface(const
USB_CONFIGURATION_DESCRIPTOR* cfg, const USB_INTERFACE_DESCRIPTOR*
start)
```

Получить указатель на следующий дескриптор интерфейса, где `cfg` - декскриптор конфигурации, `start` - дескриптор интерфейса относительно которого будет найден следующий дескриптор интерфейса.

```
USB_INTERFACE_DESCRIPTOR* usb_find_interface(const
USB_CONFIGURATION_DESCRIPTOR* cfg, uint8_t num)
```

Найти дескриптор интерфейса по номеру, где `cfg` - дескриптор конфигурации, `num` - номер интерфейса.

```
void* usb_interface_get_first_descriptor(const
USB_CONFIGURATION_DESCRIPTOR* cfg, const USB_INTERFACE_DESCRIPTOR*
start, unsigned int type)
```

Получить указатель на первый дескриптор у которого тип (`bDescriptorType`) равен `type`, где `cfg` - дескриптор конфигурации, `start` - дескриптор интерфейса относительно которого будет найден дескриптор.

```
void* usb_interface_get_next_descriptor(const
USB_CONFIGURATION_DESCRIPTOR* cfg, const void *start, unsigned int
type)
```

Получить указатель на следующий дескриптор у которого (`bDescriptorType`) равен `type`, где `cfg` - дескриптор конфигурации, `start` - дескриптор относительно которого будет найден следующий дескриптор интерфейса.

13.4. Класс CDC ACM

Класс `CDC ACM` используется для передачи/приема неструктурированных последовательных данных с контролем режима передачи или без него. Основное использование — виртуальный коммуникационный порт.

Пользователю для работы с CDC ACM достаточно получить хэндл на поток приема/передачи и отправлять/получать данные.

IPC_GET_TX_STREAM - получить хэндл TX потока.

IPC_GET_RX_STREAM - получить хэндл RX потока.

Пример использования:

```
void comm_usbd_stream_rx(APP* app, unsigned int size) {
  HANDLE tx_stream = get_handle(app->usbd, HAL_REQ(HAL_USBD_IFACE,
IPC_GET_TX_STREAM), USBD_IFACE(0, 0), 0, 0);
  HANDLE rx_stream = get_handle(app->usbd, HAL_REQ(HAL_USBD_IFACE,
IPC_GET_RX_STREAM), USBD_IFACE(0, 0), 0, 0);

  HANDLE tx = stream_open(tx_stream);
  HANDLE rx = stream_open(rx_stream);

  printf("rx: %d ", size);
  char c;
  unsigned int i;
  for (i = 0; i < size; ++i) {
    stream_read(rx, &c, 1);
    //echo
    stream_write(tx, &c, 1);
    if ((uint8_t)c >= ' ' && (uint8_t)c <= '~')
      printf("%c", c);
    else
      printf("\\x%u", (uint8_t)c);
  }
  printf("\n");
  stream_listen(rx_stream, 0, HAL_USBD);
}

void app_usb_request(APP* app, IPC* ipc) {
  switch (HAL_ITEM(ipc->cmd)) {
  case USBD_ALERT:
    switch (ipc->param1) {
      case IPC_STREAM_WRITE:
        comm_usbd_stream_rx(app, ipc->param3);
        break;
  default:
    error(ERROR_NOT_SUPPORTED);
  }
}
```

За установку/получение режима работы отвечают следующие запросы.
USB_CDC_ACM_SET_BAUDRATE - установить режим работы CDC.

USB_CDC_ACM_GET_BAUDRATE - получить режим работы CDC.

USB_CDC_ACM_SEND_BREAK - отправка сигнала break.

Пример использования:

```
IPC ipc;

uint32_t baud = 9600;
uint8_t data_bits = 8;
char parity = 'N'
uint8_t stop_bits = 1;

ack(app->usbd, HAL_REQ(HAL_USBD_IFACE, USB_CDC_ACM_SET_BAUDRATE),
USBD_IFACE(0, 0), baud, (data_bits << 16) | (parity << 8) |
baudrate->stop_bits);

ipc.process = app->usbd;
ipc.cmd = HAL_REQ(HAL_USBD_IFACE, USB_CDC_ACM_GET_BAUDRATE);

call(&ipc);
```

sys_config

#define USBD_CDC_ACM_TX_STREAM_SIZE 32 - размер буфера потока передачи в байтах.

#define USBD_CDC_ACM_RX_STREAM_SIZE 32 - размер буфера потока приема в байтах.

USBD_CDC_ACM_FLOW_CONTROL 0 - данная опция разрешает/запрещает поддержку таких запросов к классу устройства как: USB_CDC_ACM_SET_BAUDRATE, USB_CDC_ACM_GET_BAUDRATE, USB_CDC_ACM_SEND_BREAK.

#define USBD_CDC_ACM_DEBUG 1 - вывод отладочной информации такой как: успех/неудача при конфигурировании устройства, разного рода предупреждения.

#define USBD_CDC_ACM_DEBUG_FLOW 0 - вывод низкоуровневой отладочной информации.

13.5. Класс HID Boot Keyboard

HID - класс устройств USB для взаимодействия с человеком. Интерфейс USB HID устройств снабжен особым дескриптором, который определяет, является ли устройство загрузочным. Загрузочное устройство, строго соответствующее минимальным требованиям протокола, будет распознано и загружено базовой системой ввода/вывода компьютера. В REx реализована поддержка загрузочного протокола для клавиатуры.

USB HID клавиатура является достаточно простым классом USB устройства и поэтому он не требует какого либо дополнительного конфигурирования, для работы с классом достаточно описать USB дескриптор.

При работе с классом у пользователя имеется возможность изменять состояние нажатых/отжатых клавиш и получение информации о состоянии индикаторов клавиатуры.

USB_HID_KBD_KEY_PRESS - отправить сообщение о нажатии клавиши.

USB_HID_KBD_KEY_RELEASE - отправить сообщение об отпускании клавиши.

USB_HID_KBD_MODIFIER_CHANGE - изменить состояние модификаторов ввода, согласно спецификации usb.org (L-SHIFT, L-CTRL и т. д.).

USB_HID_KBD_GET_LEDS_STATE - получить состояние индикаторов клавиатуры, согласно спецификации usb.org (CAPSLOCK, NUMLOCK и т. д.).

Пример использования:

```
ack(app->usbd, HAL_REQ(HAL_USBD_IFACE, USB_HID_KBD_KEY_PRESS),
USBD_IFACE(0, 0), HID_KEY_SPACE, 0);

ack(app->usbd, HAL_REQ(HAL_USBD_IFACE, USB_HID_KBD_KEY_RELEASE),
USBD_IFACE(0, 0), HID_KEY_SPACE, 0);

// Активен модификатор левый ALT
ack(app->usbd, HAL_REQ(HAL_USBD_IFACE, USB_HID_KBD_KEY_RELEASE),
USBD_IFACE(0, 0), (uint8_t)0b00000100, 0);

unit8_t leds = get(app->usbd, HAL_REQ(HAL_USBD_IFACE,
USB_HID_KBD_GET_LEDS_STATE), USBD_IFACE(0, 0), 0, 0);
```

sys_config

#define USBD_HID_DEBUG_REQUESTS 0 - вывод отладочной информации при запросах к устройству.

13.6. Класс RNDIS

RNDIS - проприетарный протокол компании Microsoft. Он реализует виртуальное сетевое соединение для большинства ОС семейства Microsoft Windows и LINUX.

Класс RNDIS в REx представляет из себя прозрачную абстракцию драйвера eth и для стека TCP/IP, таким образом не требуются какие либо дополнительные настройки.

Для работы с классом RNDIS в userspace/rndis.h определены следующие функции:

```
void rndis_set_link(HANDLE usbd, unsigned int iface, ETH_CONN_TYPE
conn)
```

Установить соединение, где usbd - хэндл процесса USBD, iface - номер и подтип интерфейса заданный с помощью макроса USBD_IFACE, conn - тип соединения.

`void rndis_set_vendor_id(HANDLE usbd, unsigned int iface, uint8_t a, uint8_t b, uint8_t c)`

Установить идентификатор производителя устройства, где `usbd` - хэндл процесса `USBD`, `iface` - номер и подтип интерфейса, `a`, `b`, `c` - идентификатор согласно спецификации usb.orb.

`void rndis_set_vendor_description(HANDLE usbd, unsigned int iface, const char* vendor)`

Установить дескриптор производителя, где `usbd` - хэндл процесса `USBD`, `iface` - номер и подтип интерфейса, `vendor` - ANSI строка.

`void rndis_set_host_mac(HANDLE usbd, unsigned int iface, const MAC* mac)`

Установить mac адрес, где `usbd` - хэндл процесса `USBD`, `iface` - номер и подтип интерфейса, `mac` - указатель на `mac` адрес.

`void rndis_get_host_mac(HANDLE usbd, unsigned int iface, MAC* mac)`

Получить mac адрес, где `usbd` - хэндл процесса `USBD`, `iface` - номер и подтип интерфейса, `mac` - указатель на `mac` адрес.

sys_config

`#define USBD_RNDIS_DEBUG 1` - вывод отладочной информации такой как: успех/неудача при конфигурировании устройства, разного рода предупреждения.

`#define USBD_RNDIS_DEBUG_REQUESTS 0` - вывод отладочной информации при запросах к устройству.

`#define USBD_RNDIS_DEBUG_FLOW 0` - вывод низкоуровневой отладочной информации связанной с потоком приема/передачи.

`#define USBD_RNDIS_MAX_PACKET_SIZE 2048` — максимальный размер пакета, включая обертки класса и непосредственно `MAC` пакет.

13.7. Класс CCID

Класс `CCID` обеспечивает протокол поддержки смарт-карт.

Общий список запросов класса CCID, которые должен реализовать пользователь для функционирования класса:

USB_CCID_POWER_ON - Запрос на включение смарт-карты. При обработке данного запроса приложению пользователя необходимо вернуть ATR через блок IO, если данного ответа не последует, то это будет означать для хост-системы, что возникли аппаратные проблемы со смарт-картой.

USB_CCID_POWER_OFF - Запрос на отключение смарт-карты.

USB_CCID_GET_PARAMS - Запрос на получение параметров установленного протокола. При обработке данного запроса приложению пользователя необходимо возвратить параметры установленного протокола (T0/T1) посредством заявленного блока IO.

USB_CCID_SET_PARAMS - Запрос на установку параметров.

USB_CCID_RESET_PARAMS - Запрос на сброс параметров.

USB_CCID_APDU — В запросе класса содержится C-APDU, на которые пользователь должен отправить R-APDU. Длина R-APDU менее 2 означает аппаратные проблемы, о чем будет сообщено хосту.

USB_CCID_CARD_INSERTED — Уведомление стека устройства о появлении карты в слоте.

USB_CCID_CARD_REMOVED — Уведомление стека устройства об извлечении карты из слота.

Последние две команды используются только для съемных ридеров. По умолчанию считается, что карты нет.

Пример:

```
static inline void app_ccid_power_on(APP* app, IO* io) {
  // Записываем ATR в блок
  io_data_write(io, &__CCID_ATR, CCID_ATR_SIZE);
  io_complete(app->usbd, HAL_CMD(HAL_USBD_IFACE,
USB_CCID_POWER_ON), USBD_IFACE(0, 0), io);
}

static inline void app_ccid_get_params(APP* app, unsigned int cmd,
IO* io) {
  // Записываем параметры протокола T1
  io_data_write(io, &__T1_PARAMS, sizeof(CCID_T1_PARAMS));
  io_complete_ex(object_get(SYS_OBJ_USBD), cmd, USBD_IFACE(0, 0),
io, CCID_T_1);
}

static void app_ccid_apdu(APP* app, IO* io) {
  // Ответ хосту на APDU запрос, если произошла ошибка.
  if (app->fail) {
    io_complete_ex(app->usbd, HAL_CMD(HAL_USBD_IFACE,
```

```
USB_CCID_APDU), USBD_IFACE(0, 0), io, ERROR_HARDWARE);
    return;
  }

  if (!app->sc_fail)
    led_set_mode(app, LED_MODE_BLINK_FAST);

  .....
  .....

  // Ответ хосту на APDU запрос, если работа прошла нормально.
  io_complete(object_get(SYS_OBJ_USBD), HAL_CMD(HAL_USBD_IFACE,
USB_CCID_APDU), USBD_IFACE(0, 0), io);

  if (!app->sc_fail && !app->fail)
      led_set_mode(app, LED_MODE_ON);
}

void app_ccid_usb_request(APP* app, IPC* ipc) {
  switch (HAL_ITEM(ipc->cmd)) {
  case USB_CCID_APDU:
    app_ccid_apdu(app, (IO*)ipc->param2);
    break;
  case USB_CCID_POWER_ON:
    app_ccid_power_on(app, (IO*)ipc->param2);
    break;
  case USB_CCID_POWER_OFF:
    printf("CCID power off\n\r");
    break;
  case USB_CCID_GET_PARAMS:
  case USB_CCID_SET_PARAMS:
  case USB_CCID_RESET_PARAMS:
    app_ccid_get_params(app, ipc->cmd, (IO*)ipc->param2); break;
  default:
    error(ERROR_NOT_SUPPORTED);
  }
}
```

sys_config

`#define USBD_CCID_REMOVABLE_CARD 0` — поддержка съемных считывателей карт.

`#define USBD_CCID_DEBUG_ERRORS 0` — отладка сообщений об ошибках.

`#define USBD_CCID_DEBUG_REQUESTS 0` — вывод отладочной информации о запросах.

`#define USBD_CCID_DEBUG_IO 0` — вывод низкоуровневой отладочной информации, включая данные в `C-APDU/R-APDU`. Обязательно отключить в релизе.

13.8. Класс MSC

Класс MSC обеспечивает поддержку реализации блочных устройств, таких как: USB Flash, DVD-ROM и т. д.

Для первоначальной настройки необходимо указание конфигурации интерфейса во время инициализации процесса USBD. Конфигурирование производится посредством стандартной команды usbd_register_configuration. Конфигурация состоит из количества LUN типа uint32_t, за которой следуют конфигурации каждого конкретного тома устройства:

```
typedef struct {
    uint32_t lun_count;
    SCSI_STORAGE_DESCRIPTOR flash1;
    SCSI_STORAGE_DESCRIPTOR flash2;
    SCSI_STORAGE_DESCRIPTOR dvd;
} MY_MSC_CONFIG;
```

SCSI_STORAGE_DESCRIPTOR имеет следующий формат (см. /userspace/scsi.h):

```
typedef struct {
    HANDLE storage, user;
    char* vendor;
    char* product;
    char* revision;
    unsigned int hidden_sectors;
    uint16_t flags;
    HAL hal;
    uint8_t scsi_device_type;
} SCSI_STORAGE_DESCRIPTOR;
```

storage — хэндл процесса хранилища.

user — хэндл хранилища (будет передаваться в ipc->param1)

vendor, product, revision — описание хранилища согласно спецификации SCSI.

hidden_sectors — количество секторов в хранилище, занятые для служебной информации. Данная цифра будет вычитаться из количества секторов в STORAGE_MEDIA_DESCRIPTOR хранилища перед отправкой на хост.

flags — флаги. В данный момент используется только флаг STORAGE_FLAG_REMOVABLE, что указывает на возможность изъятия носителя из считывателя.

hal — группа HAL, которая будет использоваться при обращении к хранилищу (будет передаваться в старших битах ipc->cmd)

scsi_device_type — тип устройства по спецификации SCSI. На данный момент используются:

SCSI_PERIPHERAL_DEVICE_TYPE_DIRECT_ACCESS — блочное устройство с прямой адресацией — USB Flash, внешний HDD, и т.д.

SCSI_PERIPHERAL_DEVICE_TYPE_CD_DVD — CD/DVD привод. Для использования должна быть включена опция SCSI_MMC в sys_config.

Для реализации запросов к устройству, хранилище должно соответствовать спецификации интерфейса storage, описанного ранее в документации. При этом не важно, реализовано ли хранилище в прикладном процессе (например, виртуальная файловая система в памяти) или же в аппаратном драйвере. В последнем случае класс будет обращаться к драйверу напрямую, минуя прикладной процесс.

sys_config

#define USBD_MSC_DEBUG_ERRORS — включение режима базовой отладки.

#define USBD_MSC_DEBUG_REQUESTS — включение режима отладки запросов

#define USBD_MSC_DEBUG_IO — низкоуровневая отладка. Значительно снижает производительность. Рекомендуется использовать только для разработки драйверов.

#define USBD_MSC_IO_SIZE — размер рабочего буфера в байтах. Рекомендуется делать кратным размеру кластера целевой файловой системы. Если весь запрос не влезет в рабочий буфер, он будет фрагментирован, поэтому этот параметр рекомендуется устанавливать максимально большим.

В данный момент подсистема SCSI используется исключительно как библиотека класса MSC, поэтому конфигурирование подсистемы описано в этом разделе:

#define SCSI_SENSE_DEPTH 10 — глубина лога сообщений SCSI. Нет никаких причин изменять стандартное шаблонное значение.

#define SCSI_LONG_LBA — поддержка носителей объемом более 32ГБ. Если таковая не планируется, можно отключить для экономии памяти.

#define SCSI_VERIFY_SUPPORTED — поддержка команд проверки данных. Можно отключить для экономии памяти.

#define SCSI_WRITE_CACHE — отправка подтверждения до фактической записи данных. Значительно ускоряет скорость записи, однако, усложняется диагностика ошибок носителей.

#define SCSI_SAT — используется для более детальной отладки на некоторых новых ядрах ОС Linux. Рекомендуется отключить для экономии памяти.

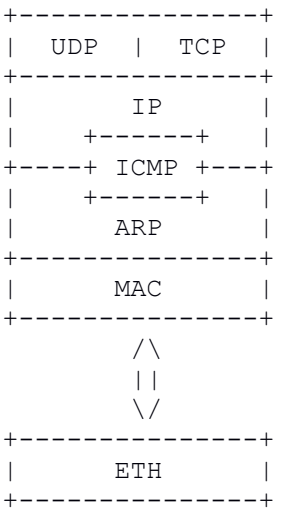
`#define SCSI_MMC` — поддержка стандарта `SCSI MMC6`. Используется для реализации `DVD-ROM`. Если функционал не используется, можно отключить для экономии памяти.

`#define SCSI_DEBUG_REQUESTS` — низкоуровневая отладка запросов к подсистеме `SCSI`. Значительно снижает производительность, рекомендуется включать только для диагностики ошибок.

`#define SCSI_DEBUG_ERRORS` — отладка ошибок запросов к подсистеме `SCSI`. Ошибками, например, считаются также события запроса носителя при его отсутствии в ридере. Поэтому также рекомендуется использовать только для диагностики ошибок.

14. TCP/IP

Стек TCP/IP состоит из следующих компонентов:



Где `eth` — отдельный процесс, который представляет собой интерфейс драйвера, описанного выше.

Работа с `TCP/IP` начинается с создания процесса:

```
HANDLE tcpip_create(unsigned int process_size, unsigned int priority, unsigned int eth_handle)
```

`process_size`: размер памяти процесса в байтах

`priority`: приоритет процесса

`eth_handle`: пользовательский хэндл драйвера. Зависит от конкретной реализации драйвера. Используется только для наименования стека, т. к. их в системе может быть несколько.

После создания необходимо провести конфигурирование подсистем TCP/IP: указать MAC, IP и т. д. Затем стек активируется функцией:

```
bool tcpip_open(HANDLE tcpip, HANDLE eth, unsigned int eth_handle, ETH_CONN_TYPE conn_type)
```

tcpip: хэндл процесса TCP/IP

eth: хэндл процесса драйвера

eth_handle: пользовательский хэндл драйвера

conn_type: требуемый тип подключения. Допустимо указывать ETH_AUTO для автоматического определения скорости среды.

Функция вернет true в случае успеха.

По завершении работы стек деактивируется функцией:

```
void tcpip_close(HANDLE tcpip)
```

tcpip: хэндл процесса TCP/IP

Узнать состояние подключения можно как у процесса драйвера напрямую, так и у самого стека:

```
ETH_CONN_TYPE tcpip_get_conn_state(HANDLE tcpip)
```

tcpip: хэндл процесса TCP/IP

Функция возвращает текущий тип подключения

sys_config

#define TCPIP_DEBUG — включить общую отладку сообщений TCP/IP. Все остальные функции отладки зависят от этой опции.

#define TCPIP_DEBUG_ERRORS — включить отладку ошибок.

#define TCPIP_MTU — размер MTU.

#define TCPIP_MAX_FRAMES_COUNT — максимальное количество фрагментированных пакетов для будущей сборке на уровне IP.

14.1. MAC

MAC уровень относится к низшему уровню стека TCP/IP. На этом уровне пакеты, пришедшие от драйвера, фильтруются и направляются на более высокий уровень — ARP, IP. Не требует конфигурирования, функции управления носят вспомогательный характер.

API

```
void mac_print(const MAC* mac)
```

mac: MAC адрес

Распечатать MAC адрес в консоль. stdout должен быть открыт.

```
bool mac_compare(const MAC* src, const MAC* dst)
```

src, dst: целевые MAC адреса.

Сравнить два MAC адреса. Возвращает true в случае успеха.

```
void mac_enable_firewall(HANDLE tcpip, const MAC* src)
```

tcpip: хэндл TCP/IP стека

src: входящий адрес

Включить firewall на прием пакетов только с источника src.

```
void mac_disable_firewall(HANDLE tcpip)
```

tcpip: хэндл TCP/IP стека

Отключить MAC-firewall.

sys_config

#define MAC_FILTER — включить программный фильтр пакетов по целевому адресу. Только если не поддерживается аппаратно.

#define MAC_FIREWALL — включить функционал MAC-firewall.

#define TCPIP_MAC_DEBUG — включить низкоуровневую отладку MAC пакетов. Ввиду большого объема данных рекомендуется использовать только для отладки системных ошибок.

14.2. ARP

ARP ответственен за связывание физических (MAC) адресов и логических (IP) адресов конечных устройств. ARP реализован в виде таблице ARP кэшей. Связывание происходит автоматически при запросе IP адреса — как входящем, так и исходящем.

API

```
bool arp_add_static(HANDLE tcpip, const IP* ip, const MAC* mac);
```

tcpip: хэндл TCP/IP стека

ip: логический адрес

mac: физический адрес

Добавляет запись в таблицу без тайм-аута. True в случае успеха.

```
bool arp_remove(HANDLE tcpip, const IP* ip)
```

tcpip: хэндл TCP/IP стека

ip: логический адрес

Удаляет запись с логическим адресом из таблицы кэшей. True в случае успеха.

```
void arp_flush(HANDLE tcpip)
```

tcpip: хэндл TCP/IP стека

Очищает все динамические записи в кэше

```
void arp_show_table(HANDLE tcpip)
```

tcpip: хэндл TCP/IP стека

Выводит в консоль таблицу кэшей. Отладка ARP должна быть включена.

sys_config

#define ARP_DEBUG — включить отладку ARP.

#define ARP_DEBUG_FLOW — низкоуровневая отладка всех сообщений о добавлении/удалении записей в таблицу. Рекомендуется использовать только для отладки.

#define ARP_CACHE_SIZE_MAX — максимальное количество записей в таблице кэшей.

`#define ARP_CACHE_INCOMPLETE_TIMEOUT` — тайм-аут в секундах определения логического адреса устройства, после которого оно считается недоступным в сети.

`#define ARP_CACHE_TIMEOUT` — тайм-аут в секундах времени жизни динамической записи в таблице кэшей

14.3. ICMP

`ICMP` ответственен за отправку и получения служебных сообщений, таких как эхо, уведомление о недоступности хоста и т. д. Является необязательной подсистемой, поэтому необходимо явно включить в `sys_config.h`. Это служебный уровень внутри стэка TCP/IP, для пользователя доступна лишь команда эхо-посылки:

```
bool icmp_ping(HANDLE tcpip, const IP* dst)
```

`tcpip`: хэндл `TCP/IP` стека

`dst`: логический адрес удаленного хоста

Проверить доступность адреса в сети. `True` в случае успеха.

sys_config

`#define ICMP` — разрешить функционал.

`#define ICMP_DEBUG` — включить отладку `ICMP`.

`#define ICMP_ECHO_TIMEOUT` — тайм-аут эхо-посылки в секундах.

`#define ICMP_ECHO` — разрешить отвечать на эхо-запросы и посылать эхо.

14.4. IP

`IP` — уровень логических пакетов. На этом уровне происходит сборка/разборка пакетов, маршрутизация и разнесение пакетов по протоколам. В данный момент (бета) поддерживается только статическая адресация адресов внутри одной подсети.

API

```
void ip_print(const IP* ip)
```

`ip`: логический адрес

Отправить логический адрес в консоль. `Stdout` должен быть открыт.

```
uint16_t ip_checksum(void *buf, unsigned int size)
```

`buf`: буфер данных пакета

`size`: размер буфера данных

Посчитать контрольную сумму пакета. В основном, используется непосредственно стеком.

```
bool ip_compare(const IP* ip1, const IP* ip2, const IP* mask)
```

`ip1, ip2`: логические адреса для сравнения

`mask`: маска для сравнения

Сравнивает два логических адреса, возвращает `true`, если идентичны с учетом маски.

```
void ip_set(HANDLE tcpip, const IP* ip)
```

`tcpip`: хэндл `TCP/IP` стека

`ip`: логический адрес

Установить логический адрес устройства. Обязательная функция перед открытием стека `TCP/IP`.

```
void ip_get(HANDLE tcpip, IP* ip)
```

`tcpip`: хэндл `TCP/IP` стека

`ip`: логический адрес

Получить текущий логический адрес устройства.

```
void ip_enable_firewall(HANDLE tcpip, const IP* src, const IP*
mask)
```

`tcpip`: хэндл `TCP/IP` стека

`src`: логический адрес

`mask`: маска для сравнения

Включить фильтрацию входящих пакетов только с адресов `src` по маске `mask`. Опция должна быть включена в `sys_config`.

```
void ip_disable_firewall(HANDLE tcpip)
```

`tcpip`: хэндл `TCP/IP` стека

Bsключить фильтрацию входящих пакетов.

sys_config

`#define IP_DEBUG` — включить отладку сообщений.

`#define IP_DEBUG_FLOW` — включить низкоуровневую отладку. Рекомендуется включать только для разработчиков стека.

`#define IP_CHECKSUM` — проверять контрольную сумму пакетов. Необходимо, если опция не поддерживается драйвером.

`#define IP_FRAGMENTATION` — функционал работы с фрагментированными пакетами.

`#define IP_FRAGMENTATION_ASSEMBLY_TIMEOUT` — тайм-аут сборки фрагментов пакета в секундах.

`#define IP_MAX_LONG_SIZE` — максимальная длина собранного пакета в байтах.

`#define IP_MAX_LONG_PACKETS` — максимальное количество одновременно собираемых фрагментированных пакетов.

`#define IP_FIREWALL` — включить функционал `IP firewall`.

14.5. UDP

Является необязательной подсистемой стека. Работа с пакетами без гарантированной доставки. Механизм подтверждения доставки определяется пользователем.

Т.к. в `UDP` отсутствует механизм подтверждения подключения к пассивному сокету, параметры исходного порта и адреса передаются через стек `IO`:

```
typedef struct {
    IP remote_addr;
    uint16_t remote_port;
} UDP_STACK;
```

Работа с сокетами состоит из:

1. Пассивного (`udp_listen`) или активного (`udp_connect`) создания соединения с удаленным хостом.
2. Чтения/записи данных в сокет. В случае, если сокет не будет открыт на данном этапе, пользователю (локальному или удаленному) будет отправлено уведомление `ICMP`

`Connection refused`. Если уровень `ICMP` отключен, пакет будет сброшен.

3. Закрытия сокета с помощью стандартного сообщения `IPC_CLOSE`.

API

```
uint16_t udp_checksum(void* buf, unsigned int size, const IP* src, const IP* dst)
```

`buf`: буфер данных пакета

`size`: размер буфера данных

`src, dst`: логические адреса, используемые при генерации контрольной суммы

Посчитать контрольную сумму `UDP` датаграммы. В основном, используется непосредственно стеком.

```
HANDLE udp_listen(HANDLE tcpip, unsigned short port)
```

`tcpip`: хэндл `TCP/IP` стека

`port`: 16 битный номер локального порта

Открыть локальный порт в пассивном режиме. Возвращает хэндл открытого сокета или `INVALID_HANDLE` в случае ошибки.

```
HANDLE udp_connect(HANDLE tcpip, unsigned short port, const IP* remote_addr)
```

`tcpip`: хэндл `TCP/IP` стека

`port`: 16 битный номер удаленного порта

`remote_addr`: логический адрес удаленного хоста

Открыть локальный порт в активном режиме. Возвращает хэндл открытого сокета или `INVALID_HANDLE` в случае ошибки.

```
void udp_read(tcpip, handle, io, size)
int udp_read_sync(tcpip, handle, io, size)
```

`tcpip`: хэндл `TCP/IP` стека

`handle`: хэндл открытого сокета

io: блок, в который будут записаны данные

size: максимальная длина датаграммы

Прочитать данные из сокета. Синхронный и асинхронный вариант. Синхронная функция возвращает количество прочитанных байт или отрицательный код ошибки.

```
void udp_write(tcpip, handle, io)
```

```
int udp_write_sync(tcpip, handle, io)
```

tcpip: хэндл TCP/IP стека

handle: хэндл открытого сокета

io: блок, в который будут записаны данные

Записать данные в активный сокет. Синхронный и асинхронный вариант. Синхронная функция возвращает количество записанных байт или отрицательный код ошибки.

```
void udp_write_listen(HANDLE tcpip, HANDLE handle, IO* io, const
IP* remote_addr, unsigned short remote_port)
```

```
int udp_write_listen_sync(HANDLE tcpip, HANDLE handle, IO* io,
const IP* remote_addr, unsigned short remote_port)
```

tcpip: хэндл TCP/IP стека

handle: хэндл открытого сокета

io: блок, в который будут записаны данные

remote_port: 16 битный номер удаленного порта

remote_addr: логический адрес удаленного хоста

Записать данные в пассивный сокет — дополнительно указываются удаленный адрес и порт. Синхронный и асинхронный вариант. Синхронная функция возвращает количество записанных байт или отрицательный код ошибки.

```
void udp_flush(HANDLE tcpip, HANDLE handle)
```

tcpip: хэндл TCP/IP стека

handle: хэндл открытого сокета

Очистить буферы подсистемы UDP.

sys_config

`#define UDP` — включить подсистему.

`#define UDP_DEBUG` — включить отладку.

`#define UDP_DEBUG_FLOW` — включить низкоуровневую отладку. Функционал разработчика, значительно снижает производительность.

14.6. TCP

`TCP`, в отличии от `UDP` гарантирует доставку конечного пакета а также имеет механизм подтверждения пассивных соединений. После открытия пассивного сокета (`tcp_listen`), пользователь должен обрабатывать сообщения `IPC_OPEN` от подсистемы `TCP` и принимать решение о принятии или закрытии соединения:

```
if (HAL_GROUP(ipc->cmd) == HAL_TCP && HAL_ITEM(ipc->cmd) ==
IPC_OPEN)
{
    if (accept)
    {
        //будет подтверждено автоматически при ipc_write
        tcp_read(tcpip, handle, io, max_size);
    }
    else
    {
         tcp_close(tcpip, handle);
    }
}
```

Для передачи специфичных для TCP параметров контроля потока используется стек поверх IO:

```
typedef struct {
    uint16_t flags;
    uint16_t urg_len;
} TCP_STACK;
```

где флаги:

`TCP_PSH` — не дожидать сборки полного пакета, отправить данные немедленно.

`TCP_URG` — данные содержат `urgent data`. Указатель на эти данные — `urg_len`.

Более подробно о флагах см. соответствующие `RFC`.

API

```
uint16_t tcp_checksum(void* buf, unsigned int size, const IP* src,
const IP* dst)
```

`buf`: буфер данных пакета

`size`: размер буфера данных

`src, dst`: логические адреса, используемые при генерации контрольной суммы

Посчитать контрольную сумму `TCP` пакета. В основном, используется непосредственно стеком.

```
void tcp_get_remote_addr(HANDLE tcpip, HANDLE handle, IP* ip)
```

```
uint16_t tcp_get_remote_port(HANDLE tcpip, HANDLE handle)
```

```
uint16_t tcp_get_local_port(HANDLE tcpip, HANDLE handle)
```

`tcpip`: хэндл `TCP/IP` стека

`handle`: хэндл открытого сокета

Получить параметры открытого сокета: удаленный и локальный порт, удаленный логический адрес.

```
HANDLE tcp_listen(HANDLE tcpip, unsigned short port)
```

`tcpip`: хэндл `TCP/IP` стека

`port`: локальный порт

Открыть сокет в пассивном режиме. Возвращает хэндл сокета или `INVALID_HANDLE` в случае ошибки.

```
void tcp_close_listen(HANDLE tcpip, HANDLE handle)
```

`tcpip`: хэндл `TCP/IP` стека

`handle`: хэндл пассивного сокета

Закрыть ранее открытый порт в пассивном режиме. Не закрывает действующие соединения.

```
HANDLE tcp_create_tcb(HANDLE tcpip, const IP* remote_addr,
uint16_t remote_port)
```

`tcpip`: хэндл `TCP/IP` стека

`remote_addr`: удаленный логический адрес

`remote_port`: удаленный порт

Создать структуру хэндла активного сокета. В качестве локального порта генерируется динамический порт. Возвращает хэндл структуры сокета или `INVALID_HANDLE` в случае ошибки.

```
bool tcp_open(HANDLE tcpip, HANDLE handle)
```

`tcpip`: хэндл `TCP/IP` стека

`handle`: структура ранее созданного хэндла

Открыть сокет в активном режиме. Возвращает `true` в случае успеха.

```
void tcp_close(HANDLE tcpip, HANDLE handle)
```

`tcpip`: хэндл `TCP/IP` стека

`handle`: хэндл пассивного сокета

Закрыть ранее созданное/открытое активное соединение.

```
void tcp_read(tcpip, handle, io, size)
int tcp_read_sync(tcpip, handle, io, size)
```

`tcpip`: хэндл `TCP/IP` стека

`handle`: хэндл открытого сокета

`io`: блок, в который будут записаны данные

`size`: максимальная длина для чтения

Прочитать данные из сокета. Синхронный и асинхронный вариант. Синхронная функция возвращает количество прочитанных байт или отрицательный код ошибки. Если пакет будет содержать флаг `TCP_PSH`, буфер будет возвращен немедленно.

```
void tcp_write(tcpip, handle, io)
int tcp_write_sync(tcpip, handle, io)
```

`tcpip`: хэндл `TCP/IP` стека

handle: хэндл открытого сокета

io: блок, в который будут записаны данные

Записать данные в сокет. Синхронный и асинхронный вариант. Синхронная функция возвращает количество прочитанных байт или отрицательный код ошибки.

```
void tcp_flush(HANDLE tcpip, HANDLE handle)
```

Очистить буферы подсистемы TCP.

sys_config

#define TCP_DEBUG — включить отладку подсистемы.

#define TCP_RETRY_COUNT — количество попыток повторной отправки пакета, после которого будет сообщено об ошибке пользователю.

#define TCP_KEEP_ALIVE — отправка keep-alive пакетов (см. RFC). Рекомендуется реализовывать keep-alive на более высоком уровне, т. к. TCP keep-alive поддерживается не всеми системами.

#define TCP_TIMEOUT — тайм-аут операции, после которого будет сообщено об ошибке.

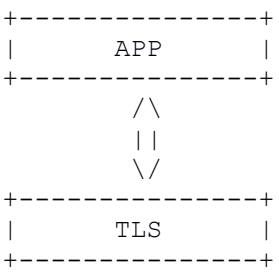
#define TCP_HANDLES_LIMIT — ограничение максимального количества одновременно открытого количества TCP сокетов.

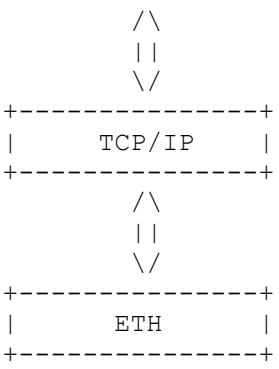
#define TCP_DEBUG_FLOW — низкоуровневая отладка данных. Режим разработчика.

#define TCP_DEBUG_PACKETS — низкоуровневая отладка данных на уровне данных заголовка пакета. Режим разработчика.

15. TLS сервер

TLS сервер реализует зашифрованное соединение. Интерфейс TLS базируется поверх TCP/IP стека и является полностью прозрачным для пользователя:





TLS на момент написания документации находится в состоянии бета-версии и не рекомендуется для промышленного использования. Текущая версия реализует протокол TLS 1.2 с профилем TLS_RSA_WITH_AES_256_CBC_SHA. Операции генерации случайных чисел и реализация RSA определяются прикладным уровнем, хеширование и шифрование — встроенными библиотеками.

Прикладной процесс ответственен за обработку сообщений TLS_GENERATE_RANDOM, TLS_PREMASTER_DECRYPT группы HAL_TLS. Все данные передаются в блоке IO:

```
void tls_request(APP* app, IPC* ipc)
{
    switch (HAL_ITEM(ipc->cmd))
    {
    case TLS_GENERATE_RANDOM:
        app_net_generate_random(app, (HANDLE)ipc->param1,
(IO*)ipc->param2);
        break;
    case TLS_PREMASTER_DECRYPT:
        app_net_premaster_decrypt(app, (HANDLE)ipc->param1,
(IO*)ipc->param2);
        break;
    default:
        error(ERROR_NOT_SUPPORTED);
        break;
    }
}
```

API

HANDLE tls_create()

Создать процесс TLS. Параметры процесса определяются в sys_config.h. Возвращает хэндл процесса или INVALID_HANDLE в в случае ошибки.

`bool tls_open(HANDLE tls, HANDLE tcpip)`

`tls`: хэндл процесса `TLS`

`tcpip`: хэндл стека `TCP/IP`

Запустить `TLS`. Возвращает `true` в случае успеха.

`void tls_close(HANDLE tls)`

`tls`: хэндл процесса `TLS`

Остановить TLS.

void tls_register_cerificate(HANDLE tls, const uint8_t* const cert, unsigned int len)

`tls`: хэндл процесса `TLS`

`cert`: сертификат `TLS`

`len`: длина сертификата в байтах

Установить сертификат для клиентов. Обязательная процедура перед открытием `TLS`.

sys_config

`#define TLS_PROCESS_SIZE` — размер памяти процесса в байтах.

`#define TLS_PROCESS_PRIORITY` — приоритет процесса.

`#define TLS_DEBUG` — включить отладку.

`#define TLS_DEBUG_REQUESTS` — отладка запросов.

`#define TLS_DEBUG_FLOW` — низкоуровневая отладка. Режим разработчика.

`#define TLS_DEBUG_SECRETS` — отладка данных шифропосылок. Только режим разработчика и только для отладки криптографии.

`#define TLS_IO_SIZE` — размер пакета блока `TLS`.

16. HTTP сервер

`HTTP` сервер в текущей реализации представлен в бета-версии. Работа с сервером заключается в конфигурировании дерева `URL`, регистрации `HTML` кодов ошибок и обработке `HTTP`-запросов.

Запрос от сервера сопровождается стеком параметров поверх IO:

```
typedef struct {
    unsigned int processed, content_size;
    HTTP_CONTENT_TYPE content_type;
    HTTP_ENCODING_TYPE encoding_type;
    HANDLE obj;
} HS_STACK;
```

processed: размер обработанных данных. Будет меньше content_size в случае, если размер IO меньше общего размера данных.

content_size: полный размер контента данных.

content_type: тип контента: текст, HTML и т. д. См. соответствующую структуру.

encoding_type: тип кодировки — см. соответствующую структуру.

obj: хэндл объекта в дереве URL.

Ответ на запрос должен сопровождаться соответствующим стеком:

```
typedef struct {
    unsigned int content_size;
    HTTP_CONTENT_TYPE content_type;
    HTTP_ENCODING_TYPE encoding_type;
    uint16_t response;
} HS_RESPONSE;
```

content_size: размер контента данных.

content_type: тип контента: текст, HTML и т. д. См. соответствующую структуру.

response: код завершения. Например: OK, FORBIDDEN и т.д.

Регистрация дерева URL производится при помощи функции hs_create_obj. При этом требуется указание на родительский объект дерева. Для первого объекта в качестве родительского необходимо использовать специальное значение хэндла HS_ROOT_OBJ. Объектом может быть как явное значение, так и специальная маска «любое значение» HS_OBJ_WILDCARD. При этом для одного родителя могут одновременно указываться и

явные значения и «любое значение» - в таком случае сервер будет пытаться сначала найти явное значение, а в случае неудачи — выбрать «любое значение». Если сервер не сможет найти запрошенный URL в дереве объектов, будет возвращена ошибка 404 Not Found.

Большая часть ошибочных запросов может быть обработана сервером без участия прикладного уровня. Для этого необходимо произвести регистрацию кода ошибки и соответствующего ей HTML при помощи функции hs_register_error. Если код ошибки не зарегистрирован, будет возвращен код HTML стандартной ошибки. Данный код должен быть обязательно зарегистрирован с указанием кода HS_GENERIC_ERROR.

API

```
HANDLE hs_create()
```

Создать процесс сервера HTTP. Параметры процесса определяются в sys_config.h. Возвращает хэндл процесса или INVALID_HANDLE в в случае ошибки.

```
bool hs_open(HANDLE hs, uint16_t port, HANDLE tcpip)
```

hs: хэндл процесса сервера HTTP.

port: порт сервера. Стандартно — 80.

tcpip: хэндл стека TCP/IP (или TLS).

Запустить сервер HTTP. Возвращает true в случае успеха.

```
void hs_close(HANDLE hs)
```

hs: хэндл процесса сервера HTTP.

Остановить сервер HTTP.

```
HANDLE hs_create_obj(HANDLE hs, HANDLE parent, const char* name,
unsigned int flags)
```

hs: хэндл процесса сервера HTTP.

parent: родительский хэндл или HS_ROOT_OBJ.

name: наименование объекта.

flags: запросы, относящиеся к данному объекту: HTTP_FLAG_GET, HTTP_FLAG_POST и

т. д. См. hs.h

Создать объект URL. Возвращает хэндл объекта или INVALID_HANDLE в случае ошибки.

```
void hs_destroy_obj(HANDLE hs, HANDLE obj)
```

hs: хэндл процесса сервера HTTP.

obj: хэндл объекта

Удалить ранее созданный объект URL.

```
HS_RESPONSE* hs_prepare_response(IO* io)
```

io: IO запроса

Инициализировать стек ответа на запрос.

```
void hs_respond(HANDLE hs, HANDLE session, unsigned int method,
IO* io, IO* user_io)
```

hs: хэндл процесса сервера HTTP.

session: хэндл сессии

method: метод ответа на запрос HTTP_GET, HTTP_POST и т. д. См. hs.h

io: IO запроса

user_io: IO, содержащее HTML ответа.

Ответить на запрос сервера.

```
void hs_respond_error(HANDLE hs, HANDLE session, unsigned int
method, IO* io, HTTP_RESPONSE code)
```

hs: хэндл процесса сервера HTTP.

session: хэндл сессии.

method: метод ответа на запрос HTTP_GET, HTTP_POST и т. д. См. hs.h

io: IO запроса.

code: код ошибки.

Ответить на запрос сервера с кодом ошибки.

```
void hs_register_error(HANDLE hs, HTTP_RESPONSE code, const char
*html)
```

hs: хэндл процесса сервера HTTP.

code: код ошибки.

html: HTML текст ошибки.

Зарегистрировать код ошибки.

```
void hs_unregister_error(HANDLE hs, HTTP_RESPONSE code)
```

hs: хэндл процесса сервера HTTP.

code: код ошибки.

Отменить регистрацию кода ошибки.

sys_config

#define HS_PROCESS_SIZE — размер памяти процесса в байтах.

#define HS_PROCESS_PRIORITY — приоритет процесса.

#define HS_DEBUG — включить отладку.

#define HS_DEBUG_HEAD — отладка заголовков HTTP.

#define HS_IO_SIZE — размер рабочего буфера сервера.

17. Глоссарий

ОСРВ — операционная система реального времени.
Контекст исполнения — логический уровень исполнения, в котором находится процессор. Различают: прикладной, системный, контекст прерывания.
Мьютекс — механизм синхронизации для обеспечения блокирующего доступа к объекту.
Событие — механизм синхронизации, при котором выполнение прикладного процесса приостанавливается до возникновения определенного события от другого процесса (или прерывания).
Очередь сообщений — механизм межпроцессного взаимодействия, при котором один процесс может отправить другому процессу сообщение, и, если тот не может его обработать мгновенно, поместить его в очередь.
Семафор — механизм синхронизации, при котором процесс, блокирующий ресурс, устанавливает соответствующий флаг — по аналогии с дорожным семафором.

Системный тик — периодическое событие, которое вызывает супервизор для обработки отсчетов времени и связанных с этим действиями.
HPET/Высокоточный таймер — системный таймер, выполняющий тот же функционал, что и системный тик, но выполняющийся не с равным промежутком времени, а в точности по достижении времени ближайшего события.
swi/svc — supervisor call. Механизм запроса к супервизору из прикладного процесса в архитектуре ARM.
PendSV — pend switch. Прерывание с минимально возможным приоритетом для обеспечения переключения контекста после вызова супервизора, но перед возвращением в пользовательский контекст.
MPU — memory protection unit. Механизм защиты адресов памяти прикладного процесса.
Хэндл — числовой идентификатор, однозначно идентифицирующий системный объект.
Системное исключение — ошибка, приводящая к вызову системных прерываний. Например, деление на 0.
Битбанг — эмуляция низкоскоростной аппаратуры посредством управления выводами микроконтроллера.
Аптайм — время, прошедшее с момента запуска ОС.
Процесс — независимо исполняемый прикладной контекст. Содержит исполняемый код, оперативную память, системные структуры. Уникально идентифицируется в составе ОС.
VIC — vectorred interrupt controller. Подсистема микроконтроллера, ответственная за реализацию прерываний.
NVIC — nested vectorred interrupt controller. Аналогично предыдущему, но с аппаратной реализацией вложенных прерываний — обычно посредством приоритетов.
Каллбэк-функция — функция, передаваемая в качестве параметра для обратного вызова из подсистемы.
dead lock — ситуация, при которой происходит взаимная блокировка.
глиф — бинарное представление графического символа.
сэмпл — единица отсчета дискретного аудиосигнала.
Ethernet PHY — аппаратно зависимая часть ethernet, зачастую реализованная внешним устройством.
ШИМ/PWM — широтно-импульсная модуляция. Процесс управления мощностью, подводимой к нагрузке, путем изменения скважности импульсов.
VSR — USB запросы, не описанные в стандарте, реализованные определенным производителем оборудования.

ATR — Answer To Reset. Набор параметров, возвращаемый смарт-картой после сброса.

LUN — Logical unit number. Количество логических устройств в интерфейсе USB MSC.

MTU — Maximum transfer unit. Максимальное число байт пакета ethernet.

MAC а) Уникальный адрес оборудования. б) подуровень стека TCP/IP.

firewall — механизм блокировки пакетов по принципу адресации и/или содержания.

ARP, ICMP, IP, UDP, TCP — подуровни стека TCP/IP. См. соответствующие RFC.

сокет — программный интерфейс для отправки/передачи с уникальной идентификацией

порта и адреса.

UDP датаграмма — единичный пакет данных протокола UDP.

keep-alive — механизм периодической отправки служебных сообщений для предотвращения закрытия либо проверки доступности удаленной службы.

TLS — протокол транспортного уровня для безопасной и прозрачной для пользователя передачи данных. См. соответствующие RFC.

HTTP — протокол прикладного уровня. См. соответствующие RFC.

URL — универсальный указатель ресурса. Механизм адресации протоколов прикладного уровня. См. соответствующие RFC.

HTML — язык гипертекстовой разметки для протоколов прикладного уровня. См. соответствующие RFC.