REx OS

Realtime Exokernel OS

v. 0.4.3

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: mutex, event, message queue, реже semaphor.
- tick-based системный таймер, 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.
- Tickless система. Нет необходимости будить процессор постоянно. Только по действительному событию с высокой прецизионностью 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.
- Стандартные объекты синхронизации. Mutex, event, semaphore оставлены, скорее, для совместимости, но при этом они отличаются полнотой реализации, включая вложенное наследование приоритетов мьютексов. Наиболее востребованным объектом синхронизации остается IPC, максимально адаптированный под 32-битную архитектуру Нестандартные объекты синхронизации:

- Stream - байтовый поток данных;
- IO - блок данных.

- Программные таймеры. Используют в качестве аппаратного таймеры ядра и позволяют сэкономить на аппаратных таймерах микроконтроллера. Тем не менее, они могут быть отключены для экономии порядка 400 байт кода.
- 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 стандартизированных методов передачи данных на базе файлов. И, в конечном счете, через создание 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: time, malloc, free, realloc.
- Time routines: Функции для работы с временем.
- Динамические массивы. Позволяют реализовать сложные динамические структуры, где это необходимо. Особенно — в стеках протоколов.
- 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.

- Прикладные библиотеки. Могут использоваться приложениями для встраивания в свое адресное пространство.

- Односвязные списки, двусвязные списки, кольцевые буферы.
- soft rand
- Libusb - работа с USB дескрипторами. Позволяет генерировать дескрипторы.
- GUI - базовый набор библиотек для работы с графикой: canvas, работа со шрифтами (включая поддержку utf-8), graphics.

- 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
- RNDIS

- SCSI Поддержка основных команд (SPC5, SPC3)

- - TCP/IP. Поддержка IP фрагментации, а также:
- MAC level 802.3
- RFC768, RFC791, RFC792, RFC793, RFC826
- ICMP эхо сообщения, flow control (*)
- ARP

- Криптография: AES, AES-CBC, SHA1, SHA256, HMAC-SHA1, HMAC-SHA256
- Возможности, которые находятся в стадии бета тестирования:

- HTTP Server
- TLS 1.2 Server

- Обработка ошибок и отладка. 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
- RTC: STM32F1, STM32F2, STM32F4, STM32L0
- WDT: STM32F1, STM32F2, STM32F4, STM32L0
- EEPROM: LPC11Uxx
- I2C: LPC1Uxx
- ADC, DAC: STM32F1
- USB: STM32F1_CL, STM32L0, LPC11Uxx
- Битбанг: STM32F1, STM32F2, STM32F4, STM32L0, LPC11Uxx
- 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.

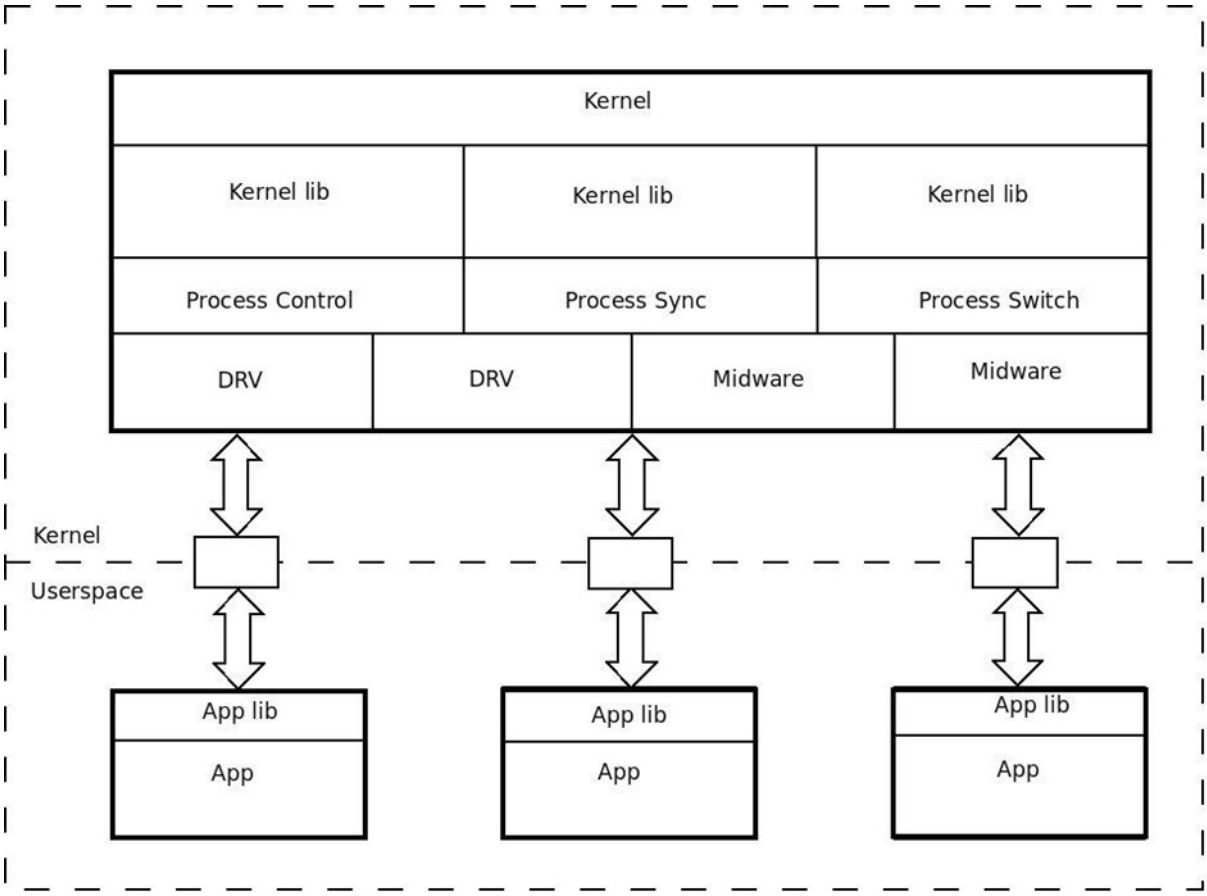


Рис. 1. Монолитное ядро

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.

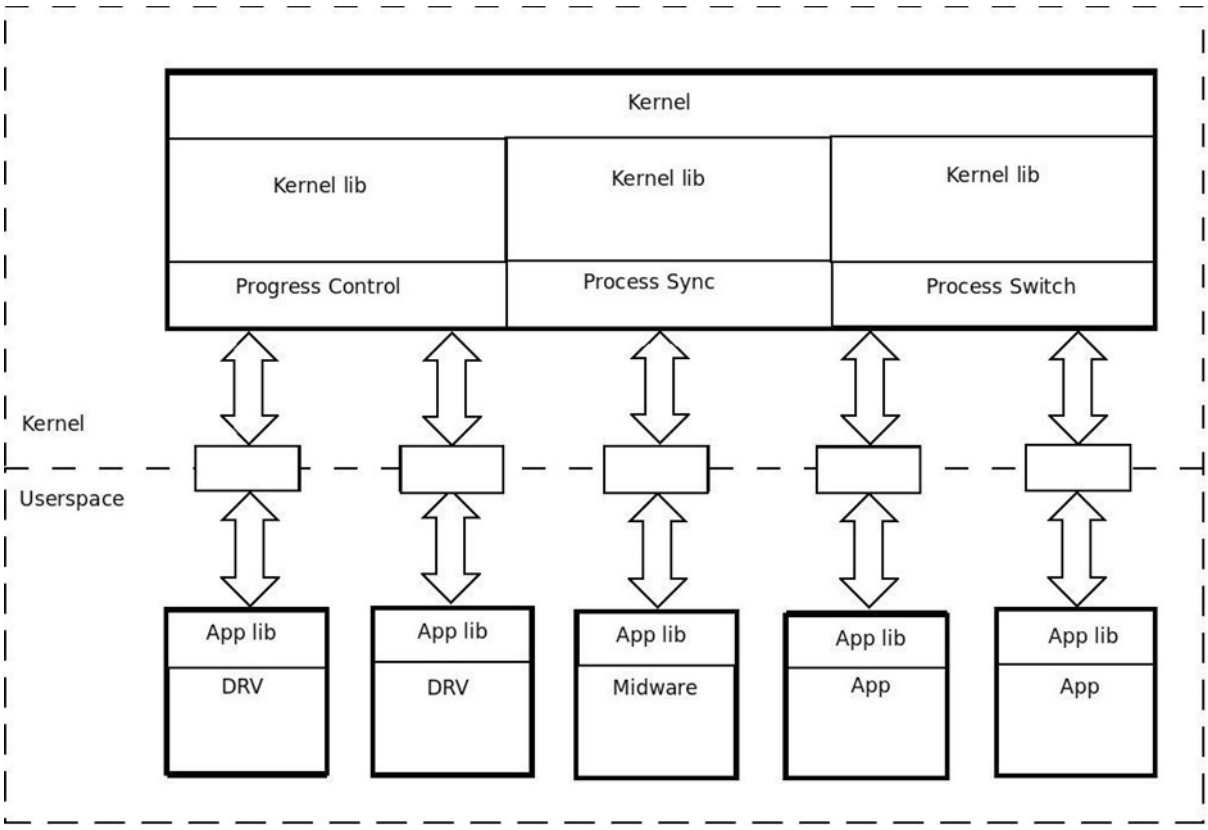


Рис. 2. Микроядро

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.

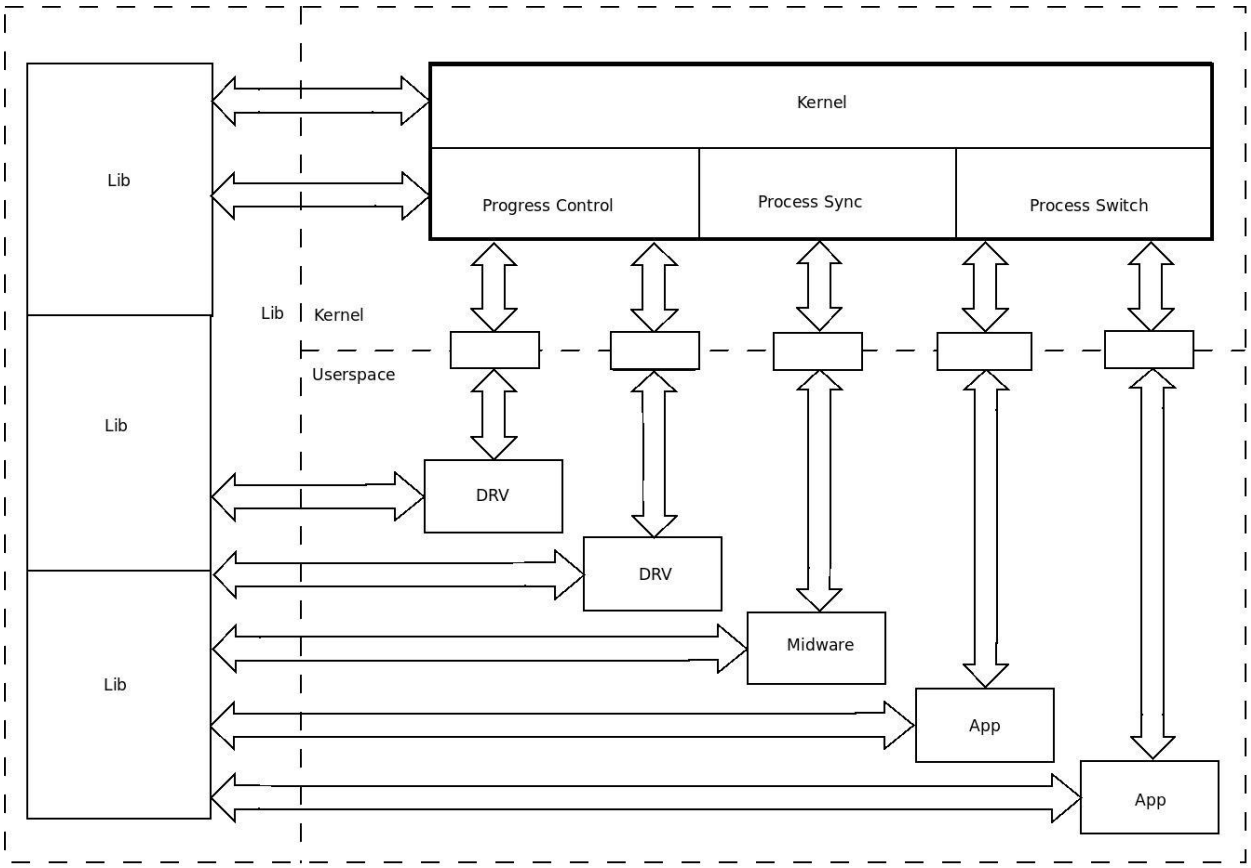


Рис. 3. Экзоядро

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. Высокопрецизионная архитектура таймера

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

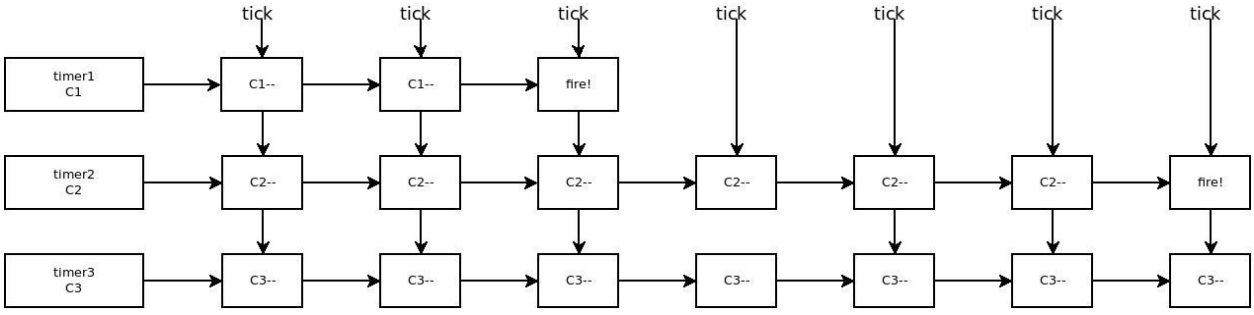


Рис. 4. Системный таймер

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.

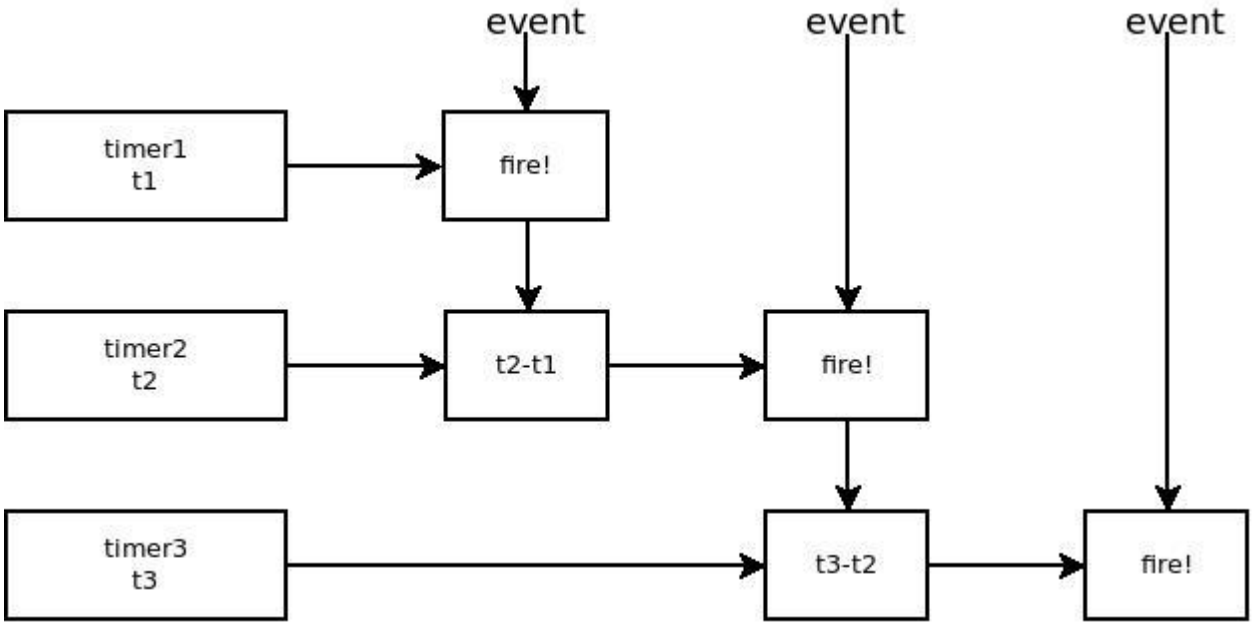


Рис. 5. Системный таймер REx

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, который формирует список «таймеров секунды» и высокопрецизионный таймер, которому задается дельта времени между двумя событиями. Диапазон в секунду выбран не случайно — как правило, во многих микроконтроллерах уже есть RTC, которые имеют секундный импульс.

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-based ОСРВ являются мьютексы. Рассмотрим на примере.

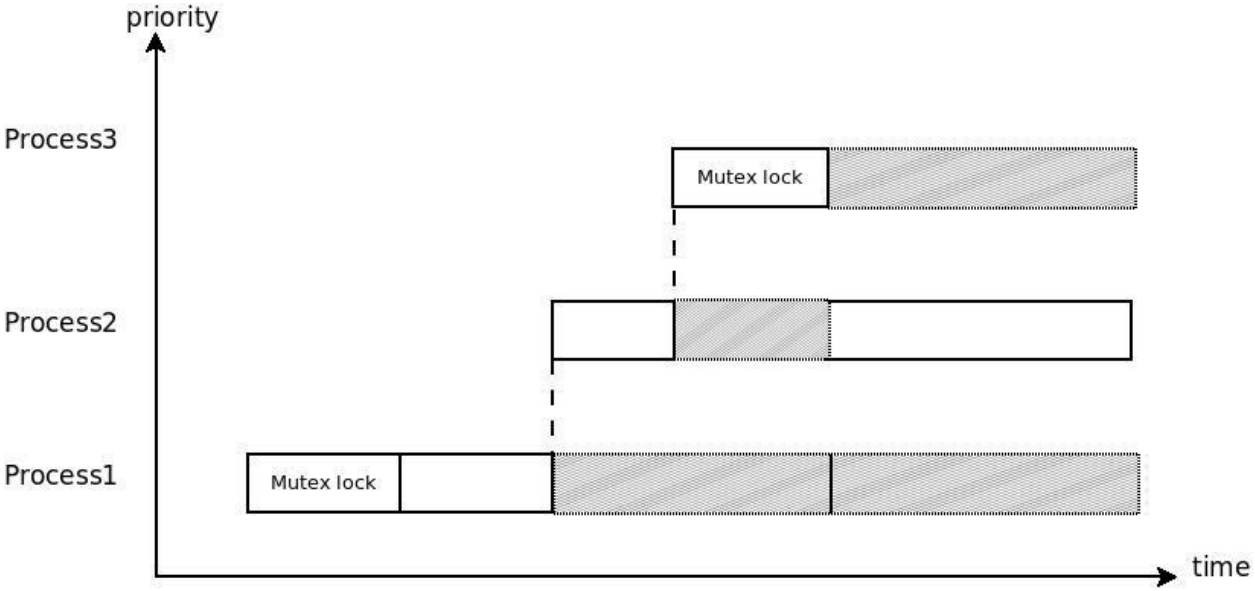


Рис. 6. Блокировка мьютексов

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 хендлеров 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:

- скрипт 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 реалтаймовость 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 wildcard.
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;
    unsigned int ipc_size;
    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.
ipc_size: размер очереди сообщений
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.

6.1. 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);
    }
}
```

6.2. Завершение процесса

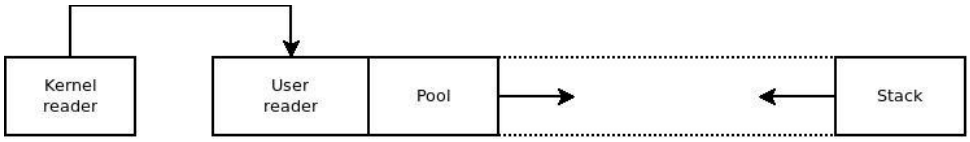
Процесс не может выйти за пределы бесконечного цикла. Чтобы завершить процесс, нужно вызвать специальную команду.

`void process_destroy(HANDLE process)`
Завершить процесс.
process: хэндл процесса

`void process_exit()`
Завершить текущий процесс.

6.3. 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.

6.4. 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 (TIME* 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 вида:.

time: указатель на структуру TIME;

ms: относительно время в миллисекундах;

us: относительное время в микросекундах.

6.5. 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.

Типы сообщений

IPC сообщений условно можно разделить на два типа:

1. Сообщение с запросом результата выполнения.
2. Сообщение 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.

Формирование и разбор кода сообщения

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" IPC используется функция `call`. Функция `ipc_post` работает в асинхронном режиме, другими словами, процесс-отправитель после отправки сообщения другому процессу может параллельно заниматься другими процессами и задачами, а после заняться обработкой ответа на сообщение (если ответ будет готов). Но хотелось бы обратить внимание на то, что асинхронные сообщения могут усложнить `state machine` рабочего цикла приема/отправки сообщений процесса-отправителя. Функция `call` работает в синхронном режиме. Синхронные вызовы могут облегчить программирование процесса. Один из примеров упрощение `state machine` рабочего цикла приема/отправки сообщений процесса-отправителя. Как правило, ответ на "вызов" записывается в один или несколько параметров `(param1, param2, param3)` сообщения. Хотелось бы добавить, что в случае если операция (которую мы отправили приемнику) займет продолжительное время, то процесс-приемник может принять решение о том, что это синхронный ответ. 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 RExOS 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 немного позже.

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 хендла 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 выпольнения 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. Результатом возврата будет HANDLE объект, а в случае ошибки 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 вNVALID_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

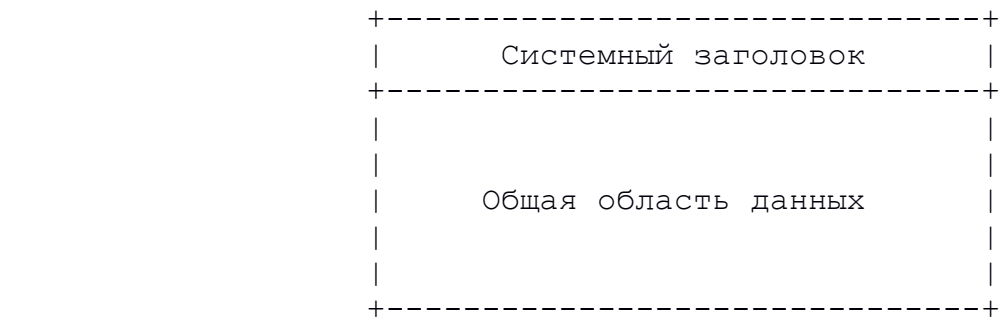
Для процессов, работающих с потоками, могут также использоваться запросы get для получения хэндла потока:
```
get (process, IPC_GET_RX_STREAM, handle, 0, 0);
get (process, IPC_GET_TX_STREAM, handle, 0, 0);
```
где:
process: процесс-владелец потока;
handle: пользовательский хэндл, идентифицирующий поток.

7.3. Блоки IO

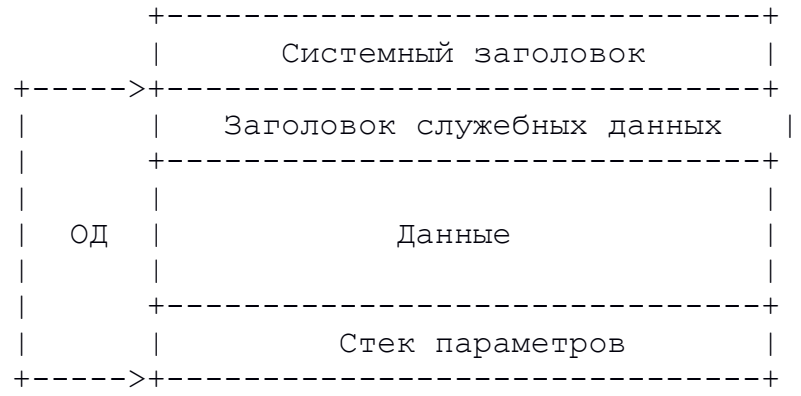
Когда необходима передача больших объемов данных между процессами, в REx используются блоки IO.

Структура и предназначение

Блок IO можно представить следующим рисунком.

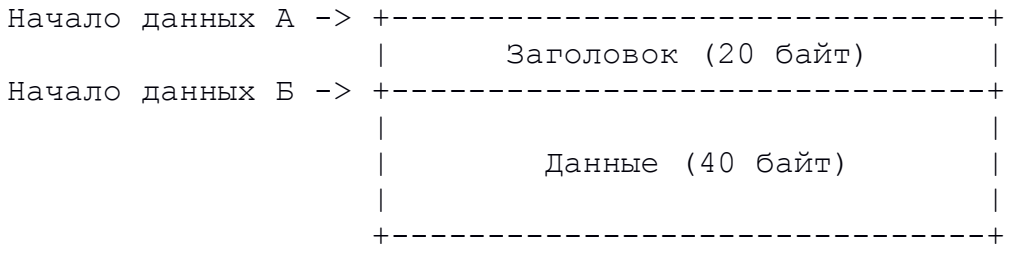


Но при необходимости общая область данных (для сокращения назовем ее ОД) может представлять собой три логические группы: заголовок служебных данных (СД), стек параметров (СП) и непосредственно сами данные (ДН), и тогда IO будет иметь следующий вид.



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 RExOS. 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)
```

Вернет разницу с момента запуска системы и `from`. Результат будет записан в `res`.

```
unsigned int systime_elapsed_ms(SYSTIME* from)
```

Аналогична предыдущей, но результат вернет в миллисекундах.

```
unsigned int systime_elapsed_us(SYSTIME* from)
```

Аналогична предыдущей, но результат вернет в микросекундах.

```
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 (описанная в userspace/ipc.h). Возвращает хэндл таймера или `INVALID_HANDLE` в случае ошибки.

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

Запускает таймер с тайм-аутом `time` в формате `SYSTIME`, если произойдет тайм-аут, 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_start_ms(HANDLE timer, unsigned int time_ms)`

Аналогична `timer_start(HANDLE timer, SYSTIME* time)`, но в качестве тайм-аута принимает миллисекунды.

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

Аналогична `timer_start(HANDLE timer, SYSTIME* time)`, но в качестве тайм-аута принимает микросекунды.

`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 основной 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)
```

Данные 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 `NULL-terminated` 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 `NULL-terminated` строку из `stdin`.

Данные 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 Поддержка кольцевых буферов

Данная библиотека реализует не сами кольцевые буферы как таковые, а только предоставляет поддержку для реализации различного рода очередей сообщений, буферов приёма-передачи различных коммутационных интерфейсов и т. д.
"Кольцевые буферы" представляют собой следующую структуру:

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

`head`: голова.
`tail`: хвост.

`size`: размер буфера.

Как видно из структуры `RB`, библиотека кольцевых буферов в RExOS реализует не хранение определенного типа данных, а хранения индексов некоего массива данных. А прикладной процесс уже сам решает, какие конкретно данные хранятся в массиве.

API

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

Функция инициализирует буфер `rb` размером `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 индекса.

`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 RExOS 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 RExOS 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 GPIO

С точки зрения защиты памяти реализация работы с прерываниями и регистрами должна быть выделена в отдельный процесс. Однако существуют задачи, в которых требуется управление периферией непосредственно через GPIO без аппаратного управления. Например, это медленная периферия или непосредственная работа с A/D шинами без аппаратной поддержки со стороны микроконтроллера. Решение по этому вопросу — предоставлять или нет прямой доступ к периферии из приложения — лежит полностью на пользователе и может быть отключено в настройках.
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` - порт ножек (`A`, `B`, ...), `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` - порт ножек (`A`, `B`, ...), `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` - порт ножек (`A`, `B`, ...), `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` - порт ножек (`A`, `B`, ...), `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` - порт ножек (`A`, `B`, ...), `mask` - маска.

```
void gpio_set_data_out(unsigned int port, unsigned int wide)
```

Переключает ножки в режим вывода по маске, где `port` - порт ножек (`A`, `B`, ...), `wide` - маска.

`void gpio_set_data_in(unsigned int port, unsigned int wide)`

Переключает ножки в режим ввода по маске, где `port` порт ножек (A, B, ...), `wide` маска.

10. Kernel config

`#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 высокопрецизионный системный таймер в режиме free run (интервал больше 1 секунды, по умолчанию 2 секунды), и запущен таймер RTC. После срабатывания таймера RTC будет перезапущен высокопрецизионный 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 высокопрецизионный 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.