Операционная система uOS

Руководство программиста

Версия документа 2011-01-30.

Оглавление

АННОТАЦИЯ

В данном документе описываются состав операционной системы uOS, её установка, настройка и программный интерфейс.

1. Общие сведения об операционной системе uOS

uOS представляет собой переносимую масштабируемую встраиваемую операционную систему для промышленных применений и систем реального времени. Свободная лицензия (GPL с дополнением) позволяет использовать uOS в коммерческих приложениях.

Операционная система uOS может применяться в промышленных и коммуникационных системах с самым широким диапазоном ресурсов, от 8-битных микроконтроллеров с 16 килобайтами ПЗУ и 2 килобайтами ОЗУ, до 32-битных микропроцессоров. Система поддерживает неограниченное количество задач, приоритетов и мутексов.

Система uOS построена по модульному принципу. Базовый модуль ядра для архитектуры MIPS32 занимает около 8 килобайт ПЗУ и 2 кбайт ОЗУ. Набор используемых модулей может наращиваться в соответствии с потребностями конкретного применения. В перечень модулей входят драйверы устройств, диспетчер памяти, сетевые протоколы. Среда разработки uOS работает в операционных системах Linux, FreeBSD, Mac OS X и Windows (с использованием

Cygwin или MinGW).

Поддерживаемые архитектуры:

- Atmel AVR;
- Texas Instruments MSP430;
- ARM: Samsung S3C4530A, Atmel AT91SAM7, AT91SAM9. Режимы ARM и Thumb;
- MIPS32: Элвис MC-24;
- Intel i386: стандартные PC-совместимые компьютеры с шиной PCI и видеоадаптером VESA. Используется загрузчик Grub 0.97;
- Linux 386, в режиме виртуальной машины, с поддержкой TCP/IP. Может применяться для отладки машинно-независимых частей разрабатываемых систем: алгоритмов обработки данных, диспетчеров памяти, сетевых протоколов.

Характеристики uOS:

- вытесняющая многозадачность;
- отсутствие ограничений на количество запускаемых задач;
- отсутствие ограничений на количество уровней приоритета;
- единый механизм синхронизации "мутекс", с дополнительной функциональностью передачи сообщений;
- реализация аппаратных прерываний через сообщения;
- возможность "быстрой" обработки прерываний, без задействования механизма синхронизации;
- наследование приоритета (priority inheritance);
- простота и расширяемость системы за счет организации в виде независимых модулей;
- малый размер обязательной части системы;
- высокая степень переносимости благодаря четкому выделению архитектурно-зависимых частей;
- возможность обнаружения переполнения стека;
- сетевой стек TCP/IP v4, включая протоколы Telnet и NNTP;
- модуль протокола SNMP для применения в качестве встраиваемого агента;
- интерпретатор языка Embedded TCL для приложений, требующих командной строки;
- свободные средства разработки (GCC), в том числе для платформы Windows;
- набор примеров для каждой поддерживаемой архитектуры.

2. Условия распространения

Операционная система uOS является свободным программным обеспечением. Исходные тексты, документацию и примеры можно скачать с сайта проекта: http://code.google.com/p/uos-embedded/

uOS распространяется на условиях лицензии GPL (русский перевод: http://www.infolex.narod.ru/gpl_gnu/gplrus.html), с одним дополнением: при связывании файлов с файлами uOS получаемый в результате бинарный файл не обязан подчиняться требованиям лицензии GPL.

В других словах:

- Вы можете применять uOS для разработки встраиваемых систем. На распространение таких систем (с бинарным кодом uOS внутри) не накладывается никаких ограничений. В

частности, Вы не обязаны выдавать покупателю Вашей системы её исходные тексты. Данный пункт относится только к распространению системы в виде выполняемых кодов.

- Если Вы хотите распространять модифицированную версию uOS или другую инструментальную систему, в состав которой входит uOS (либо ее существенная часть), в виде текстов или объектных файлов, Вы обязаны делать это на условиях лицензии GPL, т.е. с предоставлением полных исходных текстов.

3. Используемые технические средства

Для установки среды разработки uOS необходим компьютер с операционной системой Windows или Linux и следующими минимальными характеристиками:

- Процессор: 2 GHz Intel Pentium 4
- Оперативная память: 512 Мб
- Жёсткий диск: 2.5 Гб

4. Состав и функции

В состав uOS входят:

- базовая библиотека (runtime):
  - старт и начальная инициализация аппаратуры;
  - отладочная печать;
  - вектора прерываний;
  - inline-функции доступа к аппаратным регистрам процессора;
  - обработка строк;
  - сортировка и поиск;
  - математические функции;
- ядро (kernel):
  - создание и управление задачами (потоками);
  - мутексы как механизм блокирования задач;
  - мутексы как механизм обмена сообщениями;
  - группы мутексов как механизм множественного ожидания;
  - реализация аппаратных прерываний как сообщений;
  - наследование приоритета во избежание инверсии приоритетов;
- потоковый ввод/вывод (stream) в стиле POSIX printf/scanf;
- простой генератор случайных чисел (random);
- библиотека строковых регулярных выражений (regexp);
- менеджер динамической памяти (mem), аналог malloc/free;
- драйвер асинхронных каналов (uart);
- драйвер таймера (timer);
- менеджер динамических буферов (buf) – цепочек областей памяти для пакетной передачи данных;
- сетевой стек TCP/IP (net);
- сетевой протокол SNMP v2 (snmp);
- интерпретатор скриптового языка Embedded TCL (tcl);
- драйверы сетевых контроллеров Ethernet (cs8900, enc28j60, s3c4530, Linux tun/tap).

5. Установка системы

5.1. Развёртывание исходных текстов uOS

Исходные тексты системы можно скачать из сети в виде единого архива, например uos-embedded.tar.gz, и распаковать. Либо можно загрузить последнюю версию из репозитория SVN командой:

```
svn checkout http://uos-embedded.googlecode.com/svn/trunk/ uos-embedded
```

В основном калатоге uOS присутствуют три подкаталога:

- sources – исходные тексты ядра, базовой библиотеки и основных модулей системы;
- contrib – исходные тексты дополнительных модулей, например графической оконной системы Nano-X;
- examples – примеры сборки простых приложений для различных платформ.

5.2. Установка компилятора

Бинарные версии компилятора GCC для различных платформ можно скачать с сайта uOS по ссылке: http://code.google.com/p/uos-embedded/downloads/list

Например, для архитектуры MIPS32:

- gcc441-mipsel-mingw.zip – компилятор GCC версии 4.4.1, для Windows с пакетом MinGW 5.1 и Msys 1.0.11. Пакет MinGW можно скачать таам же под именем mingw51.zip. Необходимо распаковать mingw51.zip в “c:\”, образуются два каталога “c:\MinGW” и “c:\Msys”. Файл gcc441-mipsel-mingw.zip надо распаковать в каталог “c:\Msys\1.0\local”.
- gcc441-mipsel-cygwin.zip – компилятор GCC версии 4.4.1, для Windows с пакетом Cygwin 1.7. Пакет Cygwin можно скачать по ссылке: ftp://ftp.vak.ru/windows/cygwin171.zip. Распакуйте архив, образуется каталог cygwin-dist. Выполните скрипт install.bat. Файлы будут переписаны в каталог c:\Cygwin. Для проверки вызовите c:\Cygwin\Cygwin.bat. Появится текстовое окно с приглашением "$". Компилятор надо распаковать как c:/Cygwin/usr/local/mipsel441.
- gcc441-mipsel-linux.tgz – компилятор GCC версии 4.4.1, для Linux. Распакуйте его как /usr/local/mipsel441.
- gcc441-mipsel-macosx.tgz – компилятор GCC версии 4.4.1, для Mac OS X. Необходимо распаковать его как /usr/local/mipsel441.

6. Изменение конфигурации системы

Функциональность операционной системы uOS разбита на отдельные независимые компоненты, называемые модулями. Исходные тексты модулей хранятся в отдельных каталогах, например sources/runtime, sources/kernel и т.д. При сборке конкретного проекта в файле target.cfg следует указать список требуемых модулей, например:

```
MODULES  = runtime stream kernel random mem timer uart regexp tcl
```

Единственный обязательный модуль — runtime — содержит стартовый код и функции из базовой библиотеки libc. Следует отметить, что модуль ядра системы (kernel) не является обязательным, что позволяет разрабатывать встраиваемые приложения, не требующие многозадачности и обработки прерываний.

Пользователь может добавлять свои модули в дерево исходных текстов uOS, сопровождая их файлом module.cfg и добавляя в список MODULES. Каталог текстов модуля должен также содержать include-файлы, необходимые для вставки в программу пользователя, например:

```
#include <runtime/lib.h>
```

Модули, не требующие присутвия ядра, фактически являются библиотеками подпрограмм, обеспечивающими определённую функциональность:

- runtime — базовая библиотека, единственный обязательный модуль
- stream — потоковый ввод-вывод: putc(), printf() и т.п.
- regexp — сравнение и замена текстовых строк по шаблонам
- random — генератор псевдослучайных чисел
- crc — вычисление контрольных сумм CRC

Модули, работа которых касается управления потоками или обработки прерываний, требуют присутсвия модуля kernel:

- kernel — ядро, управляющее переключением задач и синхронизацией
- timer — аппаратный таймер системы
- uart — консоль, порты RS-232
- mem — динамическое выделение и освобождение ОЗУ с поддержкой нескольких диапазонов и пулов памяти
- buf — управление памятью в виде пакетов — цепочек буферов, предназначенных для передачи данных
- nvram — управление неразрушаемой памятью NVRAM, EEPROM
- net — реализация сетевого стека TCP/IP
- snmp — сетевой протокол управления SNMP
- tcl — встраиваемый язык для скриптов и командных оболочек
- adc — интерфейс к АЦП
- cs8900 — 8/16-битный контроллер Ethernet 10Base-T
- enc28i60 — контроллер Ethernet 10Base-T с интерфейсом SPI
- i8042 — клавиатура и мышь для платформы i386
- lcd2 — символьный индикатор 2x16
- gpanel — графический индикатор
- elvees — драйверы Ethernet, PCI, LPORT, внешнего UART для процессоров НПЦ Элвис
- microchip — драйвер USB для процессора Microchip PIC32
- milandr — драйверы UART, CAN, SPI, Ethernet 5600ВГ1 для процессоров Миландр 1986BE91 с архитектурой ARM Cortex-M3
- s3c4530 — драйверы Ethernet, HDLC, I2C, NVRAM, GPIO для процессора Samsung ARM7 S3C4530A
- tap — виртуальный адаптер Ethernet для отладки под Linux386

6.1. Настройка uOS для конкретной целевой платформы

Для сборки uOS под конкретную целевую платформу необходимо создать рабочий каталог и

поместить в него два файла: target.cfg и Makefile.

6.1.1. target.cfg

Файл target.cfg задаёт архитектуру целевого процессора, список компилируемых модулей, стартовый файл, а также имя компилятора и все параметры, необходимые для сборки. Пример для архитектуры MIPS32:

```
ARCH     = mips32
MODULES  = runtime stream kernel random mem timer uart regexp tcl
OPTIMIZE = -Os -DNDEBUG

BINDIR   = /usr/local/mipsel441/bin
CC       = $(BINDIR)/mipsel-elf32-gcc -mips32 -Wall
CFLAGS   = $(OPTIMIZE) -I$(OS)/sources -DMIPS32 -DELVEES_MC24 -Werror \
           -fno-builtin -fsigned-char
ASFLAGS  = -I$(OS)/sources -DMIPS32 -DELVEES_MC24 -DELVEES_FPU_EPC_BUG
DEPFLAGS = -MT $@ -MD -MP -MF .deps/$*.dep
LDFLAGS  = -nostdlib startup.o
LIBS     = -L$(TARGET) -luos -lgcc
STARTUP  = startup-mc24.S
AR       = $(BINDIR)/mipsel-elf32-ar
SIZE     = $(BINDIR)/mipsel-elf32-size
OBJDUMP  = $(BINDIR)/mipsel-elf32-objdump -mmips:isa32r2 -D
OBJCOPY  = $(BINDIR)/mipsel-elf32-objcopy

CFLAGS   += -DKHZ=60000 -DELVEES_CLKIN=10000
```

Переменная ARCH может иметь значения arm, avr, mips32, msp430, i386 или linux386.

Переменная MODULES содержит список требуемых модулей из каталога uos/sources или uos/contrib. Если у Вас имеется свой каталог дополнительный модулей (вместо uos/contrib), Вы можете использовать его, установив переменную CONTRIB, например:

```
CONTRIB = $(HOME)/Project/mcu
```

Если для архитектуры имеется несколько разных стартовых файлов, Вы можете выбрать нужный, задав его имя в переменной STARTUP.

6.1.2. Makefile

В файле Makefile необходимо:

- определить переменную TARGET, содержащую имя рабочего каталога;
- определить переменную OS, задающую абсолютный путь к текстам uOS;
- подключить файл target.cfg параметров компиляции;
- подключить файл targets/rules.cfg из каталога uOS, содержащий правила компиляции, с учётом параметров;
- задать цели all и clean.

Вы можете также добавить правила для сборки Вашего приложения.

Пример:

```
TARGET  = $(CURDIR)
```

```
OS      = $(HOME)/Project/uos

include target.cfg

all:    startup.o libuos.a

clean:
        rm -rf *.[oasi] *~ $(MODULES)

include $(OS)/sources/rules.mak
```

По команде $ make произойдет компиляция всех требуемых модулей, и будут созданы статовый файл startup.o и библиотека uOS libuos.a.

6.2. Компиляция модулей uOS

Система состоит из набора модулей, которые компилируются независимо, в отдельных подкаталогах. При сборке пользователь указывает в файле target.cfg список модулей, которые необходимы для данного конкретного проекта.

Параметры конкретного модуля содержатся в его каталоге в файле module.cfg. Необходимо определить:

- VPATH - список каталогов с исходными текстами модуля;
- OBJS - список объектных файлов модуля.

Модуль получает параметры:

- MODULEDIR - имя каталога с текстами модуля;
- ARCH - архитектура процессора для сборки;
- TARGET - путь к рабочему каталогу компиляции;
- OS - путь к текстам uOS;
- MODULE - название компилируемого модуля.

Также модулю доступны все переменные, установленные в файле target.cfg. Таким образом, можно параметры компиляции всех модулей держать в одном месте, изменяя их при необходимости при сборке конкретной системы.

Пример:

```
VPATH   = $(MODULEDIR) $(MODULEDIR)/$(ARCH)
OBJS    = halt.o itake.o irelease.o lock.o lsignal.o ltry.o\
          main.o tcreate.o tdelete.o texit.o tname.o tprio.o\
          tsetprio.o tstack.o twait.o lgroup.o machdep.o\
          tprivate.o tsetprivate.o tyield.o
```

Для сборки модуля вызывается утилита make, используя общий скрипт uos/sources/module.mak.

7. Ядро системы

Ядро системы оперирует объектами двух основных типов: 'задача' и 'мутекс'.

Задача представляет собой поток управления (thread). Каждая задача имеет отдельный стек.

В процессе выполнения задача может захватывать необходимые мутексы. При попытке захватить мутекс, занятый другой задачей, задача блокируется до момента освобождения мутекса. Таким образом, каждая задача может находиться в одном из двух состояний: выполняемом или заблокированном.

Каждая задача имеет целочисленную характеристику - приоритет. Если в выполняемом состоянии находится более одной задачи, будет выполняться задача с более высоким (большим) приоритетом. Приоритет задается при создании задачи и может изменяться по ходу выполнения. Нулевой, самый низкий приоритет присваивается фоновой задаче.

Для облегчения отладки каждая задача имеет также имя - текстовую строку.

Мутекс представляет собой метод взаимодействия задач. Можно считать мутексы обобщением семафоров и почтовых ящиков. Кроме захвата и освобождения мутексов, задачи имеют возможность обмениваться сообщениями. Задача (или несколько задач) может ожидать сообщения от мутекса, при этом задача блокируется. Другая задача может послать сообщение мутексу, при этом все задачи, ожидающие сообщения от этого мутекса, переходят в выполняемое состояние и получают посланное сообщение. Посылающая задача не блокируется. Если ни одна задача не ждет сообщения, оно теряется.

В качестве сообщения используется произвольный указатель, обычно ссылающийся на структуру данных, содержащую требуемую информацию.

Этот же механизм сообщений применяется для обработки аппаратных прерываний. При захвате мутекса задача может присвоить ему номер аппаратного прерывания, и ожидать сообщения. При возникновении прерывания задача получит сообщение (пустое).

Для каждого аппаратного прерывания, требующего обслуживания, следует создать отдельную задачу. Такая задача обычно в бесконечном цикле ожидает сообщения о прерывании, выполняет необходимые действия с аппаратурой и посылает сообщения другим мутексам, содержащие результаты обработки.

7.1. Старт системы

При старте системы вызывается функция пользователя uos_init(), которая посредством task_create() создает необходимое количество задач. После завершения функции uos_init() запускается планировщик задач, открываются прерывания и наиболее приоритетная из созданных задач получает управление.

7.1.1. Функция uos_init()

```
void uos_init (void);
```

Функция пользователя, вызываемая системой на этапе инициализации. Ваша программа должна содержать ровно одну функцию uos_init(). Главная её цель -- создать необходимое количество задач. К этому моменту механизм синхронизации задач еще не функционирует, поэтому единственная функция ядра, которую можно (и нужно) вызывать -- это task_create(). Функция uos_init() не должна вызывать никакие другие функции ядра (task_xxx(), mutex_xxx()) ни непосредственно, ни посредством вызова других модулей. В частности, нельзя инициализировать и обращаться к модулю управления памятью (mem_alloc() и пр.).

Во время работы uos_init() прерывания запрещены.

После завершения функции uos_init() запускается планировщик задач, открываются прерывания и наиболее приоритетная из созданных задач получает управление.

Пример:

```
char stack [400];

void hello (void *data)
{
    debug_puts ("Hello, World!\n");
    uos_halt ();
}

void uos_init (void)
{
    task_create (hello, 0, "hello", 1, stack, sizeof (stack));
}
```

7.1.2. Функция uos_halt()

```
void uos_halt (int dump_flag);
```

Завершение работы отлаживаемой системы. Если параметр dump_flag имеет ненулевое значение, то на отладочный терминал выводится список задач и дамп стека вызовов. Затем, для встроенной системы, выполняется бесконечный пустой цикл. При отладке про управлением инструментальной операционной системы, например Linux, завершает задачу вызовом exit(0).

Пример:

```
char stack [400];

void hello (void *data)
{
    debug_puts ("Hello, World!\n");
    uos_halt (0);
}

void uos_init (void)
{
    task_create (hello, 0, "hello", 1, stack, sizeof (stack));
}
```

7.2. Задачи

Задача представляет собой поток управления (thread). Каждая задача имеет отдельный стек.

В процессе выполнения задача может захватывать необходимые мутексы. При попытке захватить мутекс, занятый другой задачей, задача блокируется до момента освобождения мутекса. Таким образом, каждая задача может находиться в одном из двух состояний: выполняемом или заблокированном.

Каждая задача имеет целочисленную характеристику - приоритет. Если в выполняемом

состоянии находится более одной задачи, будет выполняться задача с более высоким (большим) приоритетом. Приоритет задается при создании задачи и может изменяться по ходу выполнения. Нулевой, самый низкий приоритет присваивается фоновой задаче.

Для облегчения отладки каждая задача имеет также имя - текстовую строку.

7.2.1. Тип task_t

```
typedef struct _task_t {
    ...
} task_t;

task_t *task_create (void (*func)(void*), void *arg, char *name,
    int priority, char *stack, int stacksz);
void task_exit (void *message);
void task_delete (task_t *task, void *message);
void *task_wait (task_t *task);
int task_stack_avail (task_t *task);
char *task_name (task_t *task);
int task_priority (task_t *task);
void task_set_priority (task_t *task, int priority);
```

Структура, описывающая задачу (поток управления). Каждая задача выполняется независимо от других. Операционная система производит переключение между задачами, всякий раз выбирая для выполнения задачу, имеющую наивысший приоритет (priority). Для упрощения отладки каждой задаче присваивается имя (name). Имеется специальная задача с именем "idle" и приоритетом 0, получающая управление при отсутствии готовых к выполнению задач.

Каждая задача имеет отдельный стек (stack), использующийся при выполнении для хранения адресов возврата и параметров вызываемых функций. В стеке также сохраняется содержимое регистров процессора при переключении задач. Память для стека содержится в структуре task_t.

Для работы с задачами применяются функции, описанные в таблице 2.

Таблица 2 - Функции для работы с задачами

Функция	Описание
task_create ()	Создание задачи
task_exit ()	Завершение текущей задачи
task_delete ()	Принудительное завершение задачи
task_wait ()	Ожидание завершения задачи
task_stack_avail ()	Вычисление размера стека, не использованного задачей
task_name ()	Запрос имени задачи
task_priority ()	Запрос приоритета задачи
task_set_priority ()	Установка приоритета задачи

7.2.2. Функция task_create()

```
task_t *task_create (void (*func)(void*), void *arg, char *name,
    int prio, char *stack, int stacksz);
```

Создание новой задачи (потока) и помещение ее в очередь готовых к выполнению задач. Переключение задач не происходит (Рис. 23).

Параметры:

- func - Функция, которая будет вызвана при первом переключении на новую задачу. Не должна возвращать управление. Для завершения задачи следует применять функцию task_exit().
- arg - Данное значение будет передано в качестве аргумента при вызове функции func.
- name - Текстовая строка, обозначающая "имя" данной задачи. Используется при выдаче отладочных сообщений. Доступно посредством функции task_name.
- prio - Приоритет задачи, положительное целое значение. Задача с большим значением prio имеет более высокий приоритет.
- stack - Массив памяти, в которой будут размещаться структура данных задачи (task_t) и стек.
- stacksz - Размер массива stack в байтах. Минимальное значение зависит от процессора и выполняемой задачи. Например, для процессора AVR, рекомендуемый размер составляет 200-300 байтов. При отладке рекомендуется задать размер стека с запасом, затем с помощью функции task_stack_avail определить требуемый расход памяти для каждой задачи, и при компоновке реальной системы использовать оптимизированные значения. Для экономии стека не рекомендуется исмользовать в функциях локальные массивы и вызов alloca().

Возвращаемое значение: указатель на структуру task_t, расположенную в массиве stack.

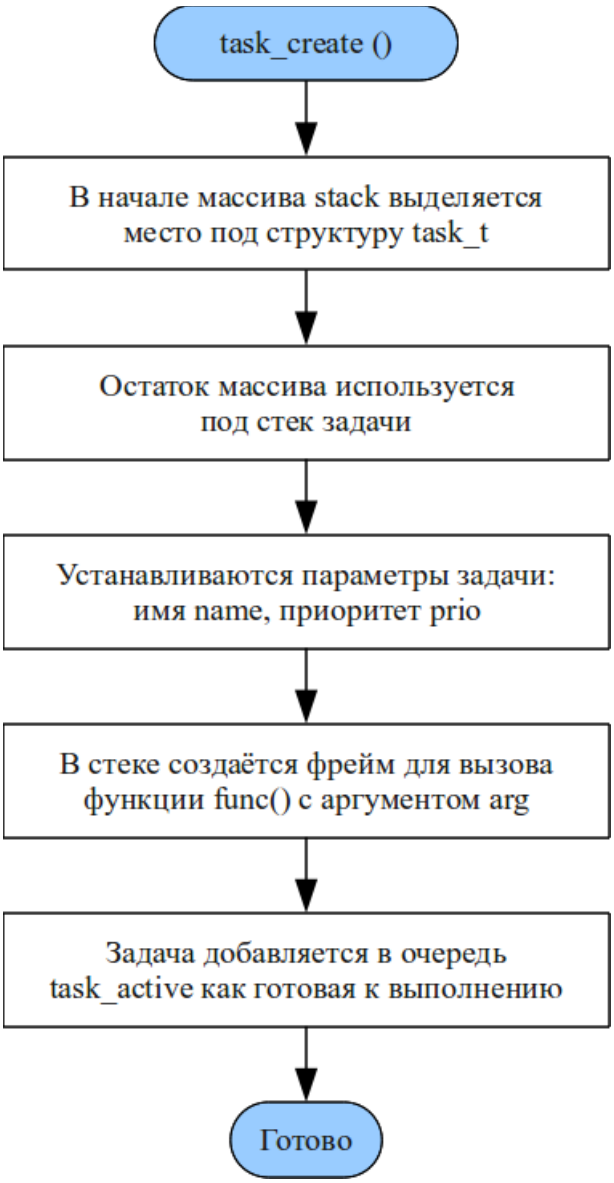


Рис. 23

Пример:

```
char stack [400];

void hello (void *data)
{
    debug_puts ("Hello, World!\n");
    uos_halt ();
}

void uos_init (void)
{
    task_create (hello, 0, "hello", 1, stack, sizeof (stack));
}
```

7.2.3. Функция task_exit()

```
void task_exit (void *status);
```

Завершение текущей задачи со статусом status. Происходит переключение задач.

Если некая задача ожидает посредством функции task_wait() завершения текущей задачи, она получит status в качестве возвращаемого значения.

Параметры:

- status --- Произвольный указатель, передаваемый в качестве сообщения функции task_wait().

Пример:

```
task_t *task2;
char stack1 [400], stack2 [400];

void main1 (void *data)
{
    void *msg;

    /* Задача 1 имеет более высокий приоритет и стартует раньше */
    debug_puts ("Task 1: waiting for task2\n");
    msg = task_wait (task2);
    debug_printf ("Task 1: task 2 returned '%s'\n", msg);
    uos_halt ();
}

void main2 (void *data)
{
    debug_puts ("Task 2: returning 'Hello'\n");
    task_exit ("Hello");
}

void uos_init (void)
{
    task_create (main1, 0, "task1", 2, stack1, sizeof (stack1));
    task2 = task_create (main2, 0, "task2", 1, stack2, sizeof (stack2));
}
```

7.2.4. Функция task_delete()

```
void task_delete (task_t *task, void *status);
```

Принудительное завершение указанной задачи со статусом status. Если завершаемая задача является текущей, происходит переключение задач (Рис. 24).

Если некая задача ожидает посредством функции task_wait() завершения текущей задачи, она получит status в качестве возвращаемого значения.

Параметры:

- task - Указатель на структуру данных завершаемой задачи.

- status - Произвольный указатель, передаваемый в качестве сообщения функции task_wait().

Рис. 24

Пример:

```
task_t *task2;
mutex_t event;
char stack1 [400], stack2 [400], stack3 [400];
```

```
void main1 (void *data)
{
    void *msg;

    /* Задача 1 имеет самый высокий приоритет и стартует раньше.
     * Ждем завершения задачи 2. */
    debug_puts ("Task 1: waiting for task2\n");
    msg = task_wait (task2);
    debug_printf ("Task 1: task 2 returned '%s'\n", msg);
    uos_halt ();
}

void main2 (void *data)
{
    /* Задача 2 стартует второй и ждет некоего события. */
    debug_puts ("Task 2: waiting for some event\n");
    mutex_wait (&event);
}

void main3 (void *data)
{
    /* Задача 3 стартует последней и завершает задачу 2. */
    debug_puts ("Task 3: killing task 2\n");
    task_delete (task2, "Killed");
    task_exit (0);
}

void uos_init (void)
{
    task_create (main1, 0, "task1", 3, stack1, sizeof (stack1));
    task2 = task_create (main2, 0, "task2", 2, stack2, sizeof (stack2));
    task_create (main3, 0, "task3", 1, stack3, sizeof (stack3));
}
```

7.2.5. Функция task_wait()

```
void *task_wait (task_t *task);
```

Ожидание завершения указанной задачи. Текущая задача останавливается и происходит переключение задач (Рис. 25).

Параметры:

- task - Задача, завершения которой ожидается.

Возвращаемое значение: статус завершения задачи, установленный вызовами task_exit() или task_delete().

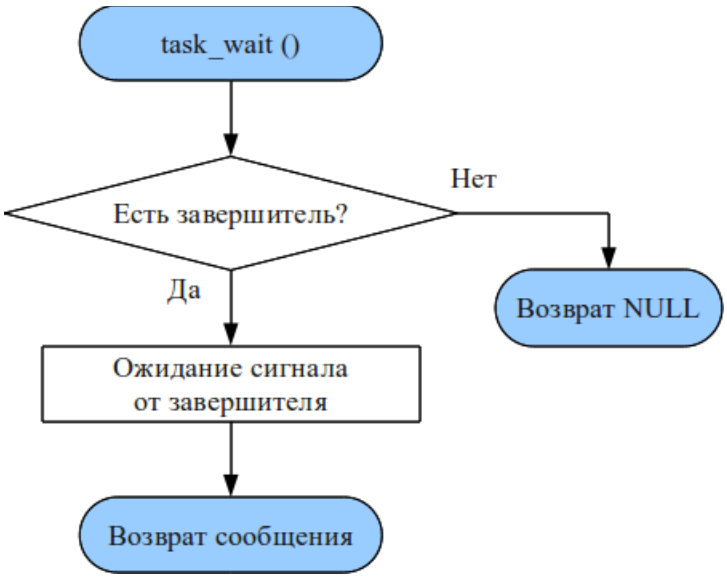


Рис. 25

Пример:

```
task_t *task2;
char stack1 [400], stack2 [400];

void main1 (void *data)
{
    void *msg;

    /* Задача 1 имеет более высокий приоритет и стартует раньше */
    debug_puts ("Task 1: waiting for task2\n");
    msg = task_wait (task2);
    debug_printf ("Task 1: task 2 returned '%s'\n", msg);
    uos_halt ();
}

void main2 (void *data)
{
    debug_puts ("Task 2: returning 'Hello'\n");
    task_exit ("Hello");
}

void uos_init (void)
{
    task_create (main1, 0, "task1", 2, stack1, sizeof (stack1));
    task2 = task_create (main2, 0, "task2", 1, stack2, sizeof (stack2));
}
```

7.2.6. Функция task_stack_avail()

```
int task_stack_avail (task_t *task);
```

Запрос размера неиспользованной части стека указанной задачи. Переключение задач не

происходит.

Вычислить размера стека, необходимый для конкретной задачи, довольно непросто. Рекомендуется при отладке задавать размер стека с запасом, затем с помощью функции task_stack_avail определить требуемый расход памяти для каждой задачи, и при компоновке реальной системы использовать оптимизированные значения. Для экономии стека не рекомендуется исмользовать в функциях локальные массивы и вызов alloca().

Параметры:

- task --- Задача, размер стека которой требуется опросить.

Возвращаемое значение: положительное целое значение -- размер в байтах неиспользованной части стека указанной задачи.

Пример:

```
char stack [400];
task_t *task;

void hello (void *data)
{
    int unused;

    unused = task_stack_avail (task);
    debug_printf ("Stack %d bytes\n", sizeof (stack));
    debug_printf ("Unused %d bytes\n", unused);
    uos_halt ();
}

void uos_init (void)
{
    task = task_create (hello, 0, "hello", 1, stack, sizeof (stack));
}
```

7.2.7. Функция task_name()

```
char *task_name (task_t *task);
```

Запрос имени указанной задачи. Имя задается при создании задачи и используется для отладочной печати. Переключение задач не происходит.

Параметры:

- task - Задача, имя которой требуется запросить.

Возвращаемое значение: указатель на текстовую строку - имя задачи.

Пример:

```
char stack [400];
task_t *task;

void hello (void *data)
{
```

```
    debug_printf ("Task name = '%s'\n", task_name (task));
    uos_halt ();
}

void uos_init (void)
{
    task = task_create (hello, 0, "hello", 1, stack, sizeof (stack));
}
```

7.2.8. Функция task_priority()

```
int task_priority (task_t *task);
```

Запрос приоритета указанной задачи. Приоритет задается при создании задачи и может изменяться функцией task_set_priority(). Переключение задач не происходит.

Параметры:

- task - Задача, приоритет которой требуется запросить.

Возвращаемое значение: целое положительное число - приоритет указанной задачи.

Пример:

```
char stack [400];
task_t *task;

void hello (void *data)
{
    debug_printf ("Task priority = %d\n", task_priority (task));
    uos_halt ();
}

void uos_init (void)
{
    task = task_create (hello, 0, "hello", 123, stack, sizeof (stack));
}
```

7.2.9. Функция task_set_priority()

```
void task_set_priority (task_t *task, int priority);
```

Изменение приоритета указанной задачи. Если новое значение приоритета превышает приоритет текущей задачи, может произойти переключение задач.

Параметры:

- task - Задача, приоритет которой требуется изменить.

Пример:

```
char stack [400];
task_t *task;
```

```
void hello (void *data)
{
    debug_printf ("Task priority = %d\n", task_priority (task));
    task_set_priority (task, 456);
    debug_printf ("New priority = %d\n", task_priority (task));
    uos_halt ();
}

void uos_init (void)
{
    task = task_create (hello, 0, "hello", 123, stack,
sizeof (stack));
}
```

7.3. Мутексы

Мутекс представляет собой метод взаимодействия задач. Можно считать мутексы обобщением семафоров и почтовых ящиков. Рекомендуется использовать мутексы для защиты структур данных, доступ к которым производится из нескольких задач (критические области).

Кроме захвата и освобождения мутексов, задачи имеют возможность обмениваться сообщениями. Задача (или несколько задач) может ожидать сообщения от мутекса, при этом задача блокируется. Другая задача может послать сообщение мутексу, при этом все ожидающие сообщения задачи переходят в выполняемое состояние и получают посланное сообщение. Посылающая задача не блокируется. Если ни одна задача не ждет сообщения, оно теряется.

В качестве сообщения используется произвольный указатель, обычно ссылающийся на структуру данных, содержащую требуемую информацию.

7.3.1. Тип mutex_t

```
typedef struct _mutex_t {
    ...
} mutex_t;

void mutex_lock (mutex_t *lock);
void mutex_unlock (mutex_t *lock);
int mutex_trylock (mutex_t *lock);
void mutex_signal (mutex_t *lock, void *message);
void *mutex_wait (mutex_t *lock);
```

Для работы с мутексами применяются функции, описанные в таблице 3.

Таблица 3 - Функции для работы с мутексами

Функция	**Описание**
`mutex_lock ()`	Захват мутекса
`mutex_unlock ()`	Освобождение мутекса
`mutex_trylock ()`	Попытка захвата мутекса

`mutex_signal ()`	Посылка сообщения
`mutex_wait ()`	Ожидание сообщения

7.3.2. Функция mutex_lock()

```
void mutex_lock (mutex_t *lock);
```

Захват мутекса. Если мутекс свободен, переключение задач не происходит. Если мутекс занят другой задачей, текущая задача блокируется до освобождения мутекса (Рис. 26).

Если мутекс связан с аппаратным прерыванием, обработка прерывания блокируется на время удержания мутекса. Отложенное прерывание будет обработано при освобождении мутекса вызовом mutex_unlock() или mutex_wait().

Параметры:

- lock - Мутекс, который требуется захватить.

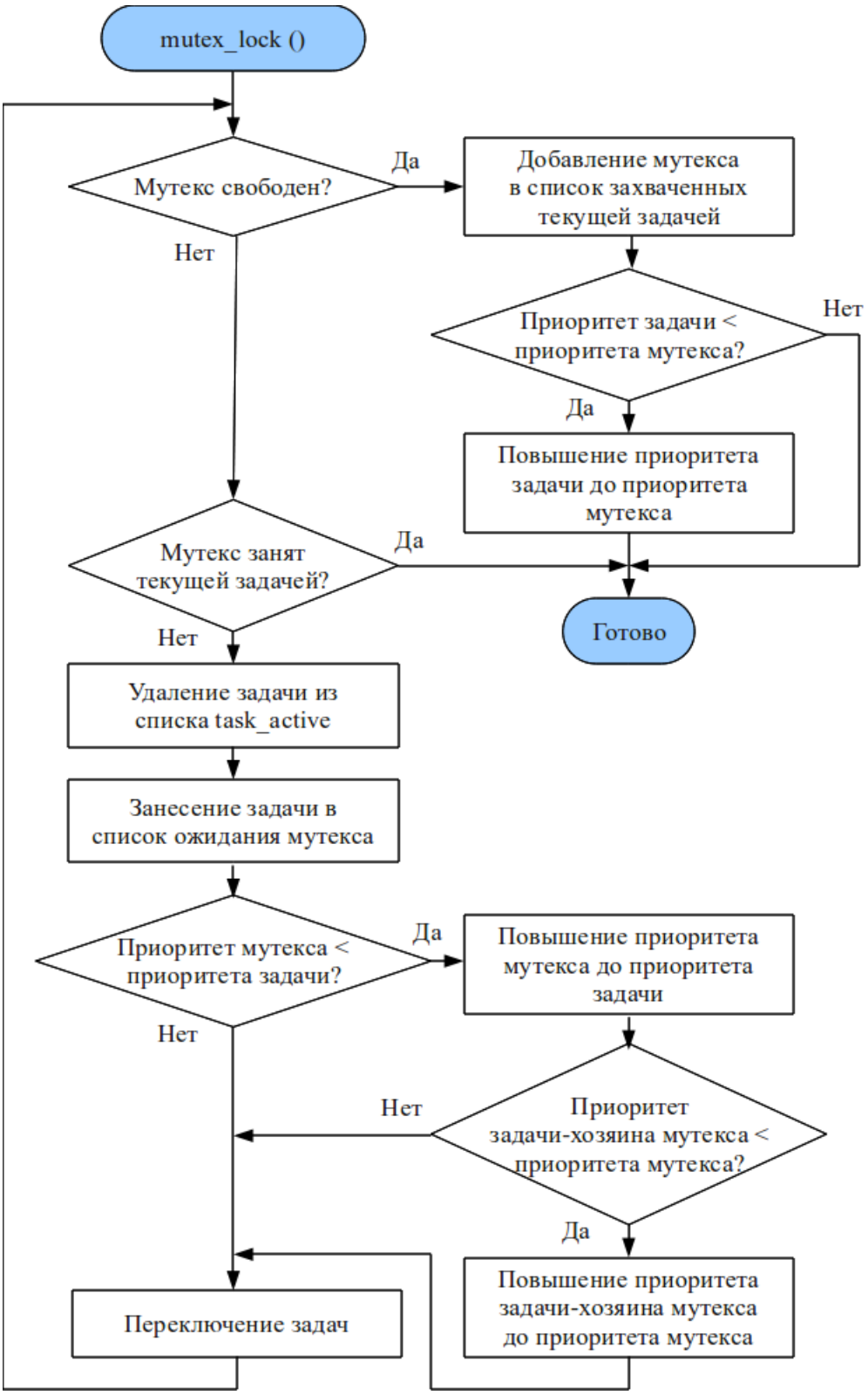


Рис. 26

Пример:

```
mutex_t lock;
timer_t timer;
char stack1 [4000], stack2 [4000];
```

```
void main1 (void *data)
{
    /* Задача 1 имеет более высокий приоритет и стартует раньше */
    debug_puts ("Task 1: taking the lock\n");
    mutex_lock (&lock);

    debug_puts ("Task 1: sleeping 1 second\n");
    timer_delay (&timer, 1000);

    debug_puts ("Task 1: releasing the lock\n");
    mutex_unlock (&lock);

    task_exit (0);
}

void main2 (void *data)
{
    /* Задача 2 приостанавливается до освобождения мутекса */
    debug_puts ("Task 2: taking the lock\n");
    mutex_lock (&lock);

    debug_puts ("Task 2: got the lock\n");
    mutex_unlock (&lock);
    uos_halt ();
}

void uos_init (void)
{
    task_create (main1, 0, "task1", 2, stack1, sizeof (stack1));
    task_create (main2, 0, "task2", 1, stack2, sizeof (stack2));
    timer_init (&timer, KHZ, 10);
}
```

7.3.3. Функция mutex_unlock()

```
void mutex_unlock (mutex_t *lock);
```

Освобождение мутекса. Если мутекс связан с аппаратным прерыванием, и за время удержания мутекса возникло прерывание, производится его обработка. Может произойти переключение задач (Рис. 27).

Параметры:

- lock - Мутекс, который требуется освободить.

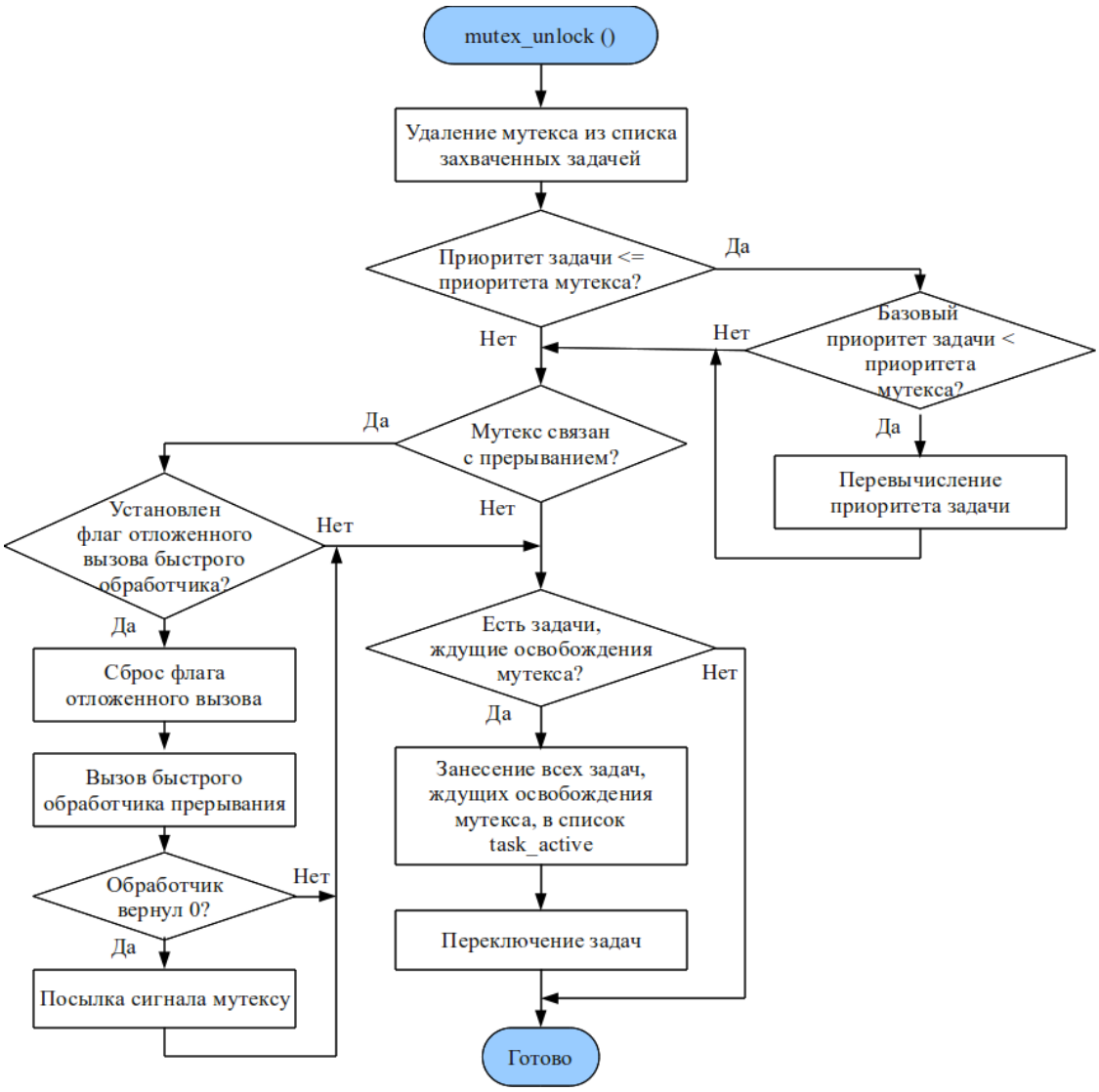


Рис. 27

Пример:

```
mutex_t lock;
timer_t timer;
char stack1 [4000], stack2 [4000];

void main1 (void *data)
{
    /* Задача 1 имеет более высокий приоритет и стартует раньше */
    debug_puts ("Task 1: taking the lock\n");
    mutex_lock (&lock);

    debug_puts ("Task 1: sleeping 1 second\n");
    timer_delay (&timer, 1000);

    debug_puts ("Task 1: releasing the lock\n");
    mutex_unlock (&lock);
```

```
    task_exit (0);
}

void main2 (void *data)
{
    /* Задача 2 приостанавливается до освобождения мутекса */
    debug_puts ("Task 2: taking the lock\n");
    mutex_lock (&lock);

    debug_puts ("Task 2: got the lock\n");
    mutex_unlock (&lock);
    uos_halt ();
}

void uos_init (void)
{
    task_create (main1, 0, "task1", 2, stack1, sizeof (stack1));
    task_create (main2, 0, "task2", 1, stack2, sizeof (stack2));
    timer_init (&timer, KHZ, 10);
}
```

7.3.4. Функция mutex_trylock()

```
int mutex_trylock (mutex_t *lock);
```

Попытка захвата мутекса. Если мутекс свободен, он захватывается. Переключение задач не происходит (Рис. 28).

Параметры:

- lock - Мутекс, который требуется захватить.

Возвращаемое значение:

Возвращает 1, если мутекс успешно захвачен. В случае неуспеха возвращается 0.

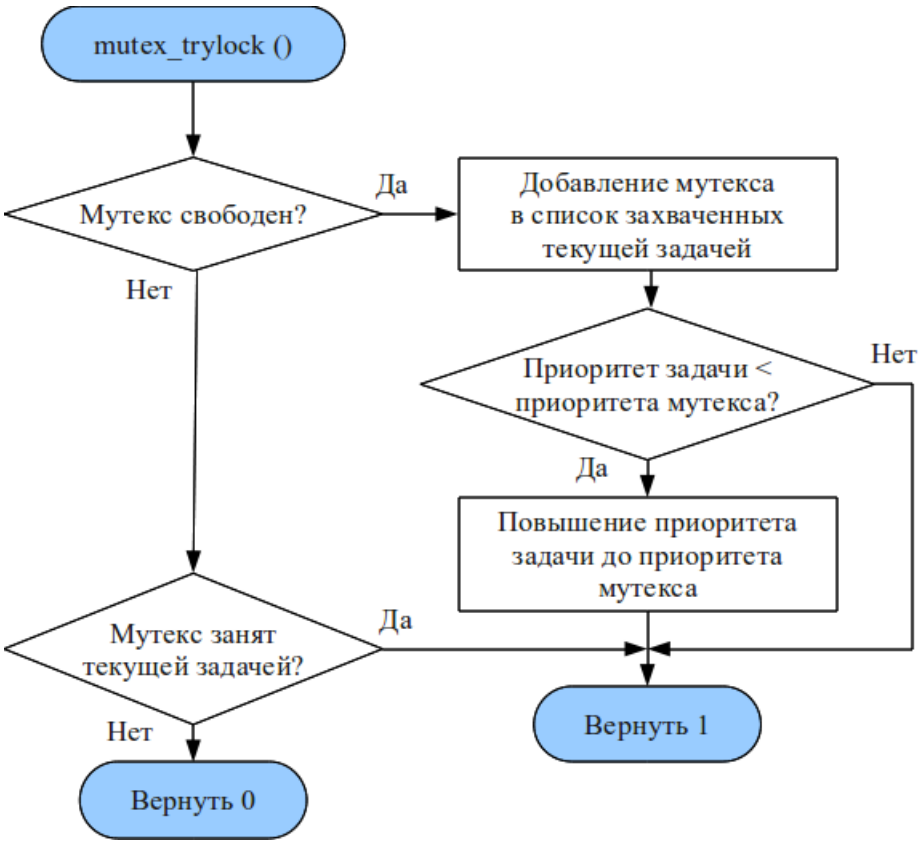


Рис. 28

7.3.5. Функция mutex_signal()

```
void mutex_signal (mutex_t *lock, void *message);
```

Посылка сообщения мутексу. Может произойти переключение задач. Если ни одна задача не ждет сообщения, оно теряется (Рис. 29).

Параметры:

- lock - Мутекс, которому посылается сообщение.
- message - Сообщение - произвольный указатель.

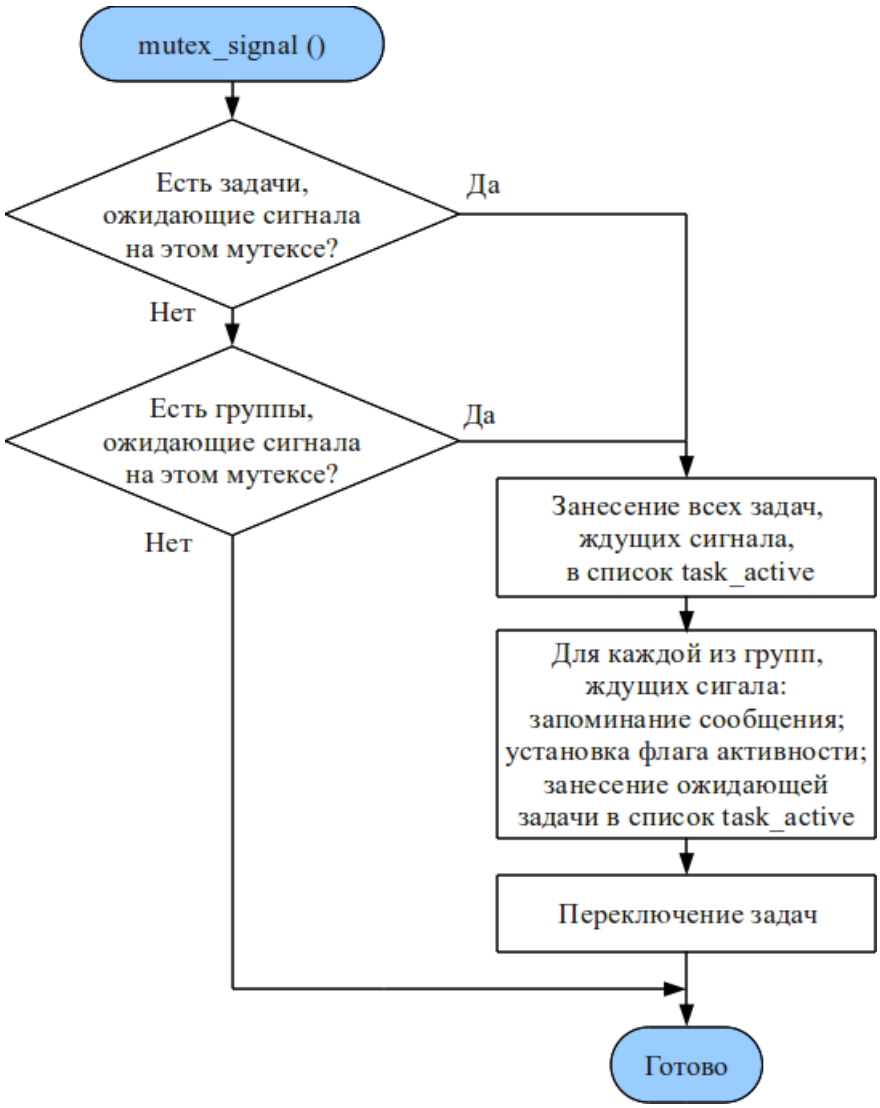


Рис. 29

Пример:

```
mutex_t lock;
char stack1 [4000], stack2 [4000];
struct signal1_t
{
    char str_val[10];
    int int_val;
};

void main1 (void *data)
{
    /* Задача 1 имеет более высокий приоритет и стартует раньше */
    struct signal1_t *sig; /* Здесь будет указатель на принятое сообщение */
    debug_puts ("Task 1: Waiting for signal\n");
    sig = (struct signal1_t *) mutex_wait (&lock);

    debug_puts ("Task 1: str_val = ");
    debug_puts (sig->str_val);
    debug_puts ("\n");
```

```
    uos_halt ();
}

void main2 (void *data)
{
    /* Задача 2 посылает сообщение */
    struct signal_t sign; /* Сообщение для Task 1 */
    sign.str_val = "Hi, Task 1";

    debug_puts ("Task 2: Sending signal to Task 1\n");
    mutex_signal (&lock, (void *) &sign);

    debug_puts ("Task 2: Sent signal to Task 1, exiting\n");
    task_exit (0);
}

void uos_init (void)
{
    task_create (main1, 0, "task1", 2, stack1, sizeof (stack1));
    task_create (main2, 0, "task2", 1, stack2, sizeof (stack2));
}
```

7.3.6. Функция mutex_wait()

```
void *mutex_wait (mutex_t *lock);
```

Ожидание сообщения. Текущая задача блокируется до момента, пока другая задача не пошлет сообщение мутексу вызовом mutex_signal(). Если мутекс был захвачен текущей задачей, на время ожидания он освобождается. Происходит переключение задач (Рис. 30).

Для надежной доставки сообщений требуется, чтобы ожидающая задача предварительно захватила мутекс. Возможно ожидание сообщений на незахваченном мутексе, но при этом надежная доставка сообщений не гарантируется.

Если мутекс связан с аппаратным прерыванием, и за время удержания мутекса возникло прерывание, производится его обработка, затем производится аппаратное разрешение (открытие маски) данного прерывания в контроллере прерываний.

Параметры:

- lock - Мутекс, для которого ожидается сообщение.

Возвращаемое значение: указатель-сообщение, посланное вызовом mutex_signal().

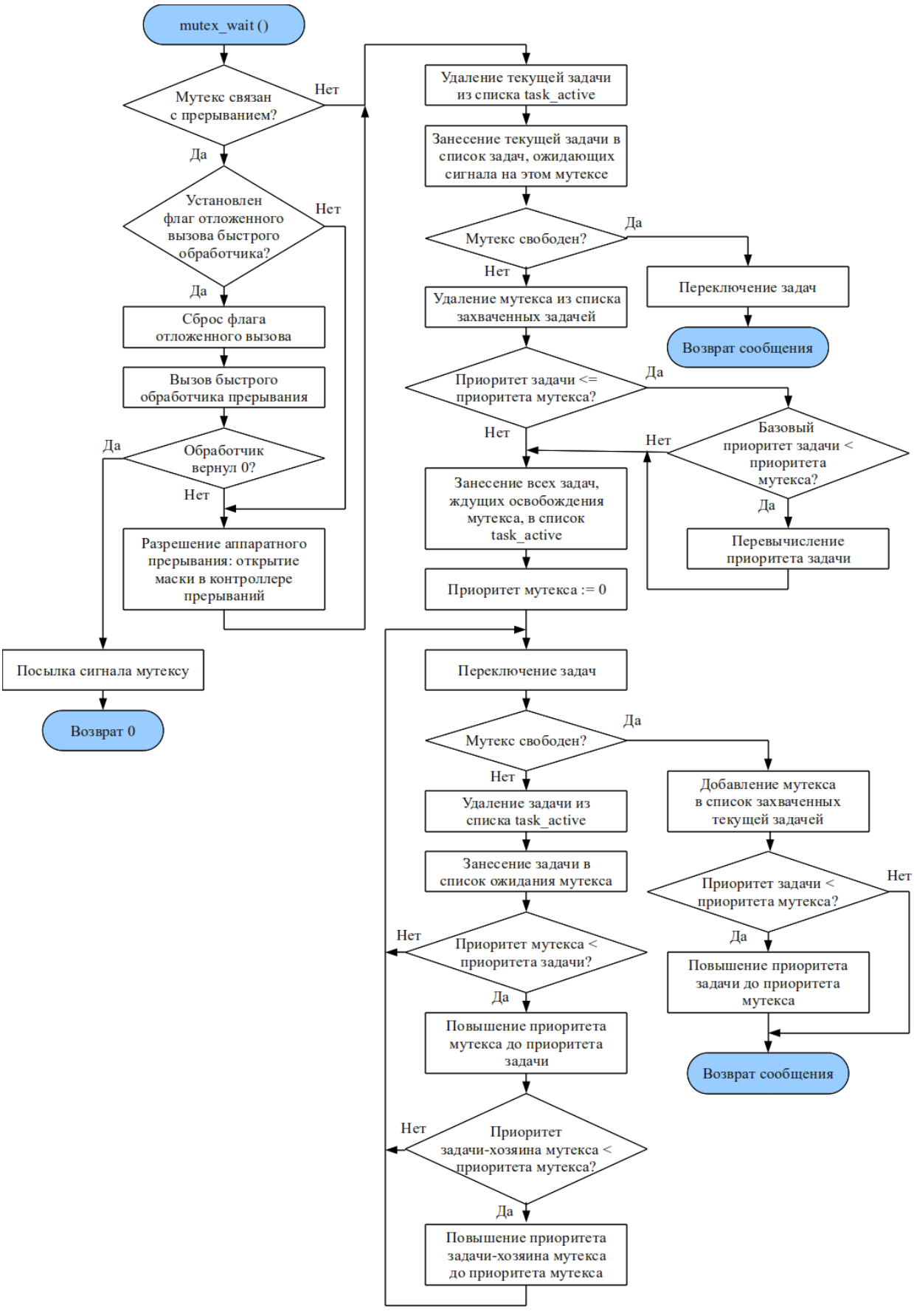

mutex_wait ()
Мутекс связан с прерыванием?
Нет
Да
Установлен флаг отложенного вызова быстрого обработчика?
Нет
Да
Сброс флага отложенного вызова
Вызов быстрого обработчика прерывания
Обработчик вернул 0?
Да
Нет
Разрешение аппаратного прерывания: открытие маски в контроллере прерываний
Посылка сигнала мутексу
Возврат 0
Удаление текущей задачи из списка task_active
Занесение текущей задачи в список задач, ожидающих сигнала на этом мутексе
Мутекс свободен?
Да
Нет
Переключение задач
Возврат сообщения
Удаление мутекса из списка захваченных задачей
Приоритет задачи <= приоритета мутекса?
Да
Нет
Базовый приоритет задачи < приоритета мутекса?
Нет
Да
Перевычисление приоритета задачи
Занесение всех задач, ждущих освобождения мутекса, в список task_active
Приоритет мутекса := 0
Переключение задач
Мутекс свободен?
Да
Нет
Добавление мутекса в список захваченных текущей задачей
Удаление задачи из списка task_active
Занесение задачи в список ожидания мутекса
Приоритет задачи < приоритета мутекса?
Нет
Да
Повышение приоритета задачи до приоритета мутекса
Возврат сообщения
Приоритет мутекса < приоритета задачи?
Нет
Да
Повышение приоритета мутекса до приоритета задачи
Приоритет задачи-хозяина мутекса < приоритета мутекса?
Нет
Да
Повышение приоритета задачи-хозяина мутекса до приоритета мутекса

Рис. 30

7.4. Прерывания

Для обработки аппаратных прерываний применяется механизм сообщений. При захвате мутекса задача может присвоить ему номер аппаратного прерывания, и ожидать сообщения. При возникновении прерывания задача получит сообщение (пустое).

После вызова mutex_lock_irq() аппаратное прерывание считается связанным с указанным мутексом. Для каждого аппаратного прерывания, требующего обслуживания, следует создать отдельную задачу. Такая задача обычно в бесконечном цикле ожидает сообщения о прерывании, выполняет необходимые действия с аппаратурой и посылает сообщения другим мутексам, содержащие результаты обработки.

Такой метод обработки прерываний обладает простотой и наглядностью, но имеет один недостаток: он требует переключения задач на каждое прерывание. Для тех случаев, когда нужна более быстрая реакция на прерывания, применяется дополнительный механизм быстрой обработки.

Быстрый обработчик представляет собой функцию, которая регистрируется при захвате мутекса прерывания, и вызывается ядром при наступлении прерывания, обеспечивая максимально высокую скорость реакции. Обслужив прерывание, быстрый обработчик принимает решение и возвращает ядру признак: следует ли посылать основной задаче-обработчику сообщение о прерывании. Таким образом, работа по обработке разделяется между основной задачей прерывания и быстрым обработчиком: срочные действия выполняет быстрый обработчик, а "медленные" - основная задача.

Пока мутекс захвачен функцией mutex_lock(), быстрый обработчик не может быть вызван "немедленно", так как это привело бы к конфликту совместного доступа к данным. В этом случае вызов быстрого обработчика откладывается до освобождения мутекса функциями mutex_unlock() или mutex_wait().

Поскольку быстрый обработчик выполняется в обход механизма синхронизации задач, на него накладываются некоторые ограничения. Во избежание конфликта с другими задачами, он имеет право работать только с данными, защищенными мутексом прерывания. Он не должен вызывать никакие функции ядра (task_xxx(), mutex_xxx()) ни непосредственно, ни посредством вызова других модулей. В частности, он не может обращаться к модулю управления памятью (mem_alloc() и пр.). Подобные действия должны перекладываться на задачу-обработчик.

При возникновении прерывания выполняются следующие действия:

- Аппаратный запрет прерывания (закрывается маска в контроллере прерываний).
- При наличии быстрого обработчика:
- Если мутекс занят - вызов быстрого обработчика откладывается до освобождения мутекса. Обработка прерывания закончена.
- Вызов быстрого обработчика. Если обработчик вернул ненулевой код - обработка прерывания закончена.
- Посылка сообщения мутексу.
- Переключение задач.

Аппаратное разрешение прерывания (открытие маски в контроллере прерываний) производится

при вызове функции mutex_wait().

```
typedef int (*handler_t) (void*);

void mutex_lock_irq (mutex_t *lock, int irq, handler_t func, void *arg);
void mutex_unlock_irq (mutex_t *lock);
```

Для работы с прерываниями применяются функции, описанные в таблице 4.

Таблица 4 - Функции для работы с прерываниями

Функция	**Описание**
mutex_lock_irq ()	Захват прерывания
mutex_unlock_irq ()	Освобождение прерывания
mutex_wait ()	Ожидание прерывания

7.4.1. Функция mutex_lock_irq()

```
void mutex_lock_irq (mutex_t *lock, int irq, handler_t func, void *arg);
```

Захват мутекса и привязка его к аппаратному прерыванию с указанным номером. Если мутекс свободен, переключение задач не происходит. Если мутекс занят другой задачей, текущая задача блокируется до освобождения мутекса (Рис. 31).

После вызова mutex_lock_irq() аппаратное прерывание считается связанным с указанным мутексом. При возникновении прерываний будет вызван быстрый обработчик, а мутекс будет получать сообщение. При последующих захватах мутекса функцией mutex_lock() вызов быстрого обработчика откладывается на время удержания мутекса. Отложенное прерывание будет обработано при освобождении мутекса вызовом mutex_unlock(), или mutex_wait().

Для каждого аппаратного прерывания, требующего обслуживания, следует создать отдельную задачу. Такая задача должна работать по следующему алгоритму:

- захватить прерывание вызовом mutex_lock_irq();
- инициализировать обслуживаемую аппаратуру;
- в цикле ожидать прерывания вызовом mutex_wait();
- выполнить необходимые действия с аппаратурой, сохранить данные и т.п.;
- при необходимости послать сообщения другим мутексам вызовом mutex_signal();
- перейти к п.3, продолжив цикл ожидания прерывания.

Параметры:

- lock - Мутекс, который требуется захватить.
- irq - Номер аппаратного прерывания, который будет привязан к данному мутексу.

- func - Указатель на функцию - быстрый обработчик прерывания. Необязательный параметр. Если нет необходимости использовать быстрый обработчик, следует указать параметр func равным 0.
- arg - Аргумент, передаваемый быстрому обработчику прерывания.

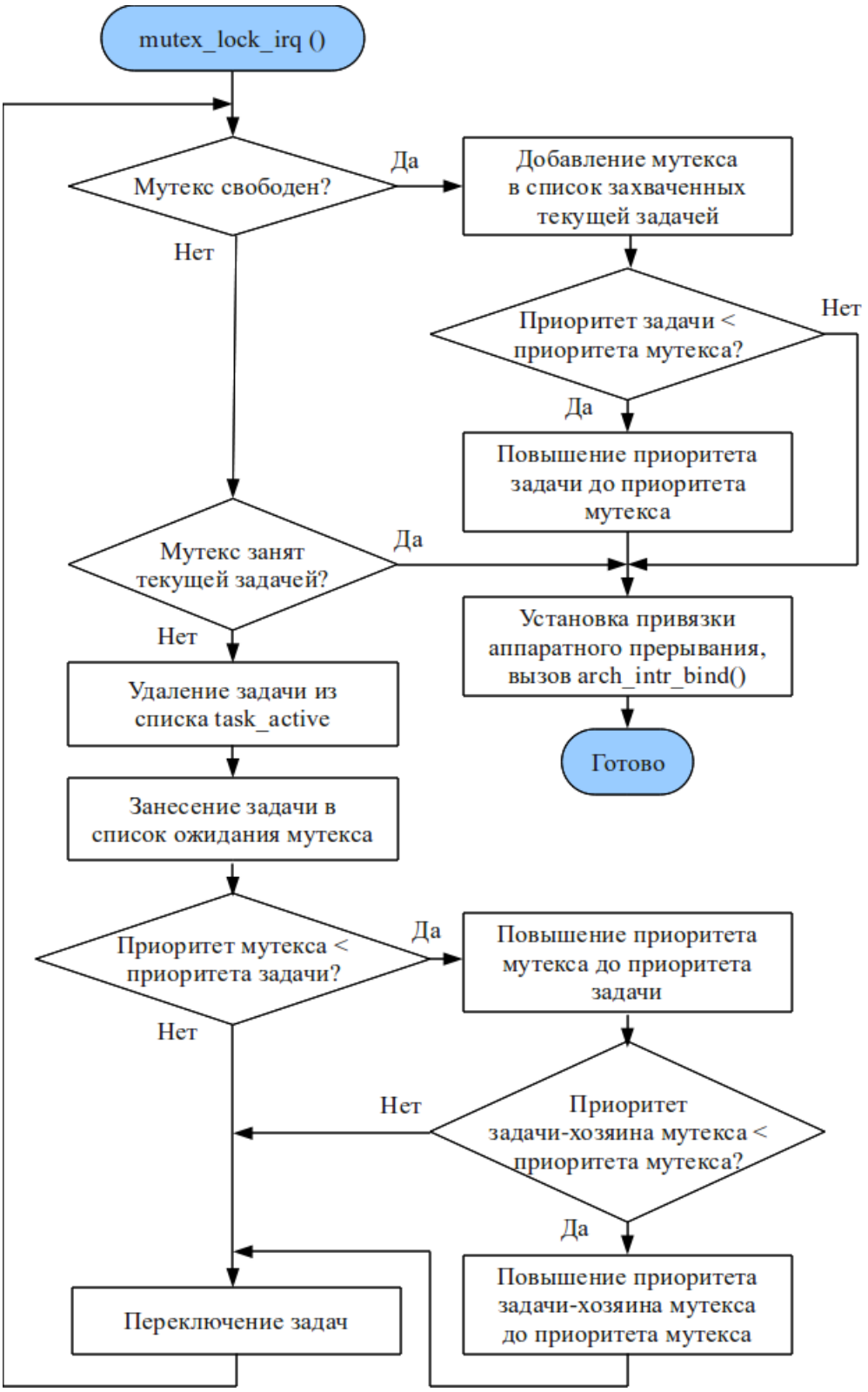

mutex_lock_irq ()
Мутекс свободен?
Да
Нет
Добавление мутекса в список захваченных текущей задачей
Приоритет задачи < приоритета мутекса?
Нет
Да
Повышение приоритета задачи до приоритета мутекса
Мутекс занят текущей задачей?
Да
Нет
Установка привязки аппаратного прерывания, вызов arch_intr_bind()
Готово
Удаление задачи из списка task_active
Занесение задачи в список ожидания мутекса
Приоритет мутекса < приоритета задачи?
Да
Нет
Повышение приоритета мутекса до приоритета задачи
Приоритет задачи-хозяина мутекса < приоритета мутекса?
Нет
Да
Повышение приоритета задачи-хозяина мутекса до приоритета мутекса
Переключение задач

Рис. 31

7.4.2. Функция mutex_unlock_irq()

```
void mutex_unlock_irq (mutex_t *lock);
```

Освобождение мутекса, отмена привязки аппаратного прерывания. Прерывание блокируется (запрещается). Может произойти переключение задач (Рис. 32).

Следует заметить, что функция mutex_unlock() не отменяет привязку и не блокирует прерывание.

Параметры:

- lock - Мутекс, который требуется освободить.

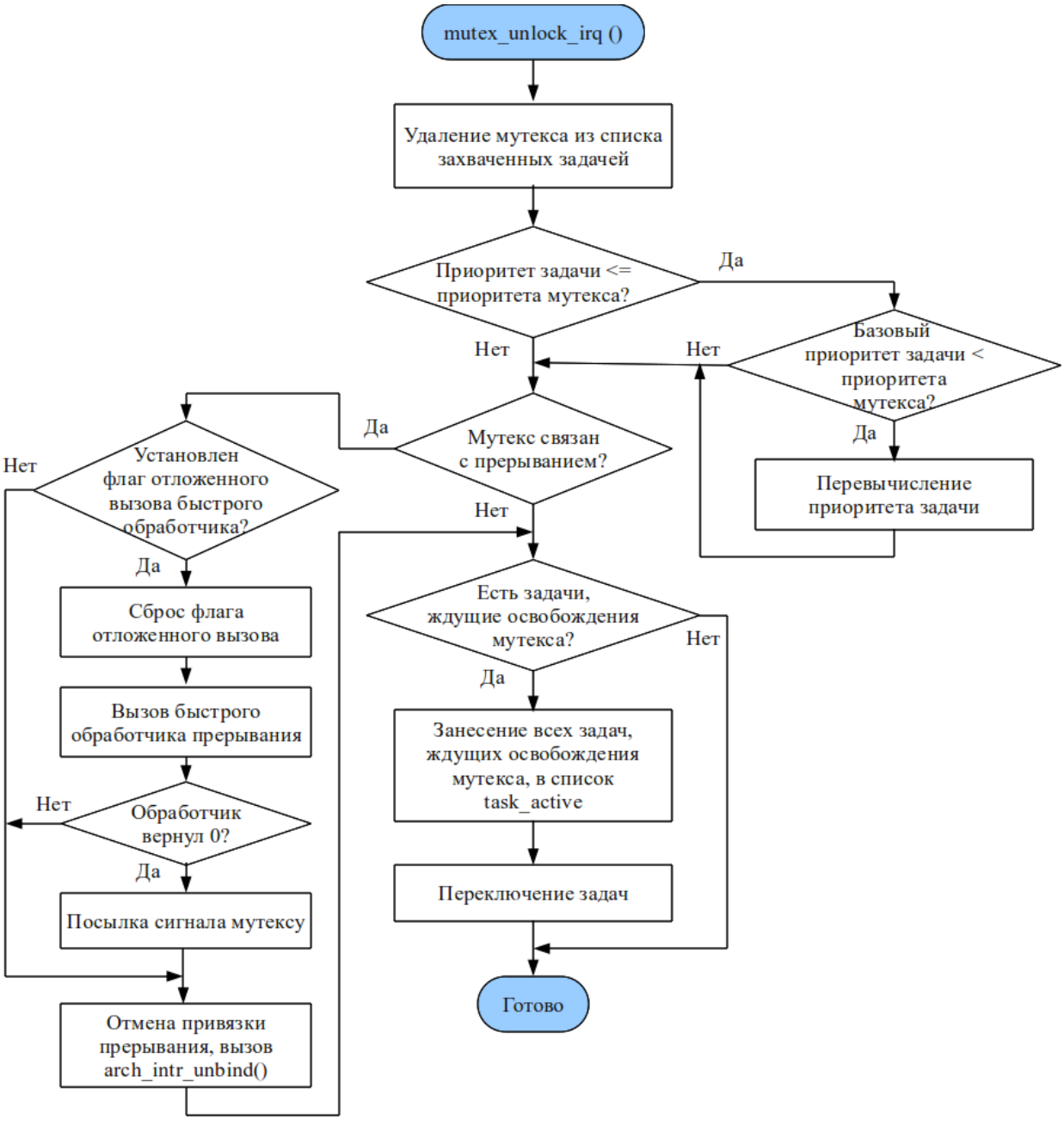


Рис. 32

7.4.3. Тип handler_t

```
typedef int (*handler_t) (void *arg);
```

Тип - указатель на функцию быстрой обработки прерываний.

Стандартный метод обработки прерываний требует переключения задач на каждое прерывание. Для тех случаев, когда нужна более быстрая реакция на прерывания, применяется дополнительный механизм быстрой обработки.

Быстрый обработчик представляет собой функцию, которая регистрируется при захвате мутекса прерывания, и вызывается ядром при наступлении прерывания, обеспечивая максимально высокую скорость реакции. Обслужив прерывание, быстрый обработчик принимает решение и возвращает ядру признак: следует ли посылать основной задаче-обработчику сообщение о прерывании. Таким образом, работа по обработке разделяется между основной задачей прерывания и быстрым обработчиком: срочные действия выполняет быстрый обработчик, а "медленные" - основная задача.

Поскольку быстрый обработчик выполняется в обход механизма синхронизации задач, на него накладываются некоторые ограничения. Во избежание конфликта с другими задачами, он имеет право работать только с данными, защищенными мутексом прерывания. Он не должен вызывать никакие функции ядра (task_xxx(), mutex_xxx()) ни непосредственно, ни посредством вызова других модулей. В частности, он не может обращаться к модулю управления памятью (mem_alloc() и пр.). Подобные действия должны перекладываться на задачу-обработчик.

Параметры:

- arg - Аргумент, передаваемый быстрому обработчику прерывания. Задается при захвате прерывания функцией mutex_lock_irq().

8. Модуль runtime — базовая библиотека

Это единственный обязательный модуль. Он содержит стартовый код системы, векторы прерываний, а также выполняет начальную инициализацию процессора (в частности, частоты PLL) и периферийных блоков, достаточную для функционирования отладочной печати и запуска программы пользователя. Также содержит стандартные макроопределения и функции, относящиеся к библиотекам libc и libm. Программа пользователя должна включать обязательную строку:

```
#include <runtime/lib.h>
```

Если нужно использовать функции с плавающей точкой, то следует дополнительно добавить строку:

```
#include <runtime/math.h>
```

Если коняигурация target.cfg не содержит модуля kernel, то выполнение приложения начинается с функции main(). Функция не должна завершаться. Пример:

```
#include <runtime/lib.h>

int main (void)
{
        for (;;) {
                debug_puts ("Hello, World!\n");
        }
}
```

Модуль runtime содержит следующие наборы функций:

- Общие типы и определения
- Обработка строк: strcpy(), memcpy() и прочие из библиотеки <string.h>
- Сортировка qsort(), atoi() и прочие из библиотеки <stdlib.h>
- Классификация символов: isdigit() и прочие из библиотеки <ctype.h>
- Макрос верификации assert()
- Математические функции и константы из библиотеки <math.h>
- Нелокальные переходы setjmp(), longjmp()
- Отладочная печать debug_printf() и другие
- Обращение к аппаратным портам микроконтроллера: inb(), outb() и их аналоги
- Задержка времени udelay(), mdelay() с калибровкой по частоте процессора
- Двусвязные списки

8.1. Общие типы и определения

Макрос __BYTE_ORDER служит для ветвления в зависимости от порядка байтов в слове. Рекомендуемое применение:

```
#if __BYTE_ORDER == __LITTLE_ENDIAN
    (от младшего байта к старшему)
#endif

#if __BYTE_ORDER == __BIG_ENDIAN
    (от старшего байта к младшему)
```

```
#endif
```

Макрос __FLOAT_WORD_ORDER полезен для ветвления в зависимости от порядка 32-битных слов в 64-битном вещественном значении:

```
#if __FLOAT_WORD_ORDER == __LITTLE_ENDIAN
    (сначала младшая половина вещественного значения, затем старшая)
#endif

#if __FLOAT_WORD_ORDER == __BIG_ENDIAN
    (сначала старшая половина вещественного значения, затем младшая)
#endif
```

Для преобразования 16-битных и 32-битных значений из локального порядка байтов в “сетевой” порядок служат макросы HTONS(), HTONL() (то же самое NTOHS(), NTOHL()). На процессорах с архитектурой big-endian эти макросы не изменяют значение своего аргумента. Для архитектуры little-endian они переставляют порядок байтов, например:

```
unsigned short a = NTOHS (0x1234);        // результат 0x3412
unsigned long  b = NTOHL (0x12345678);    // результат 0x78563412
```

Логический тип **bool_t** предназначен для оперирования логическими значениями **TRUE** (1) и **FALSE**. Он эквивалентен наименьшему целочисленному типу со знаком, эффективному для данной архитектуры.

Целые типы фиксированного размера:

int8_t	8 бит, со знаком
uint8_t	8 бит, без знака
int16_t	16 бит, со знаком
uint16_t	16 бит, без знака
int32_t	32 бит, со знаком
uint32_t	32 бит, без знака
int64_t	64 бит, со знаком
uint64_t	64 бит, без знака

Размер этих типов жестко задан и не зависит от архитектуры целевого процессора. Их рекомендуется использовать в структурах данных, а также при вычислениях, если результат зависит от количества разрядов. Типы int8_t, uint8_t, int16_t и uint16_t не очень актуальны, так как вместо них можно использовать стандартные signed char, unsigned char, signed short и unsigned short соответственно. Если требуется целочисленный тип, обеспечивающий не менее 16 бит, следует использовать обычный int или unsigned int. Если требуется тип, обеспечивающий не менее 32 бит, следует использовать int32_t или uint32_t.

Целые типы нежёсткого размера:

small_int_t	не менее 8 бит, со знаком
small_uint_t	не менее 8 бит, без знака

Эти типы рекомендуется применять в случаях, когда точный размер целого числа не имеет большого значения, например для счетчика цикла. На некоторых архитектурах это более эффективно, чем тип int или unsigned int (например Atmel AVR).

Беззнаковый тип **size_t** служит для хранения и арифметического преобразования адресов оперативной памяти.

Сравнение целочисленных типов для разных архитектур:

Тип	AVR	MSP430	ARM, MIPS, i386	ia64
int8_t	signed char	signed char	signed char	signed char
uint8_t	unsigned char	unsigned char	unsigned char	unsigned char
int16_t	int	int	short	short
uint16_t	unsigned int	unsigned int	unsigned short	unsigned short
int32_t	long	long	int	int
uint32_t	unsigned long	unsigned long	unsigned int	unsigned int
int64_t	long long	long long	long long	long
uint64_t	unsigned long long	unsigned long long	unsigned long long	unsigned long
small_int_t	signed char	int	int	int
small_uint_t	unsigned char	unsigned int	unsigned int	unsigned int
bool_t	signed char	int	int	int
size_t	unsigned int	unsigned int	unsigned int	unsigned long

8.2. Обработка строк

Модуль runtime содержит стандартные функции работы с байтовыми массивами и строковыми значениями:

```
void *memcpy (void *to, const void *from, size_t n);
```

Перепись N байтов из массива FROM в массив TO.

```
void *memset (void *s, unsigned char c, size_t n);
```

Роспись N байтов массива S значением C.

```
small_int_t memcmp (const void *s1, const void *s2, size_t n);
```

Сравнение N байтов массивов S1 и S2. Возвращает -1 если массив S1 меньше, 1 если массив S1 больше, или 0 если массивы равны.

```
void *memmove (void *dest, const void *src, size_t n);
```

Перепись N байтов из массива FROM в массив TO. Гарантируется корректность, даже если массивы перекрываются.

```
void *memchr (const void *s, unsigned char c, size_t n);
```

Поиск байта C в массиве S размером N байтов.

```
size_t strlen (const unsigned char *s);
```

Вычисление длины строки S.

```
size_t strnlen (const unsigned char *string, size_t maxlen);
```

Вычисление длины строки S с ограничением не более MAXLEN байтов.

```
unsigned char *strcpy (unsigned char *dest, const unsigned char *src);
```

Перепись строки из SRC в DEST.

```
unsigned char *strncpy (unsigned char *dest,
    const unsigned char *src, size_t n);
```

Перепись не более чем N байтов из строки из SRC в DEST.

```
unsigned char *strcat (unsigned char *dest, const unsigned char *src);
```

Дозапись строки SRC в конец строки DEST.

```
unsigned char *strncat (unsigned char *dest,
    const unsigned char *src, size_t n);
```

Дозапись не более чем N байтов из строки SRC в конец строки DEST.

```
small_int_t strcmp (const unsigned char *s1, const unsigned char *s2);
```

Сравнение строк S1 и S2. Возвращает -1 если строка S1 меньше, 1 если строка S1 больше, или 0 если строка равны.

```
small_int_t strncmp (const unsigned char *s1,
    const unsigned char *s2, size_t n);
```

Сравнение не более чем N байтов строк S1 и S2. Возвращает -1 если строка S1 меньше, 1 если строка S1 больше, или 0 если строка равны.

```
unsigned char *strchr (const unsigned char *s, unsigned char c);
```

Поиск байта C в строке S.

```
unsigned char *strrchr (const unsigned char *src, unsigned char c);
```

Поиск последнего появления байта C в строке S.

```
unsigned char *strstr (const char *s1, const unsigned char *s2);
```

Поиск строки S2 в строке S1.

```
int strspn (const unsigned char *s1, const unsigned char *s2);
```

Пропуск символов из строки S2 в строке S1. Возвращает количество пропущенных символов.

```
int strcspn (const unsigned char *s1, const unsigned char *s2);
```

Пропуск символов, не входящих в строку S2, в строке S1. Возвращает количество пропущенных символов.

```
const unsigned char *strmatch (const unsigned char *s,
    const unsigned char *pattern);
```

Сравнение строки S с шаблоном PATTERN. Шаблон может содержать специальные элементы * ? [x-y] [^x-y].

8.3. Сортировка и преобразование строк

Модуль runtime содержит стандартные функции сортировки и поиска:

```
void qsort (void *a, size_t n, size_t es,
    int (*cmp)(const void*, const void*));
```

Сортировка массива A по возрастанию, используя функцию сравнения CMP. Массив A содержит N элементов. Каждый элемент имеет размер ES байтов.

```
void *bsearch (const void *key, const void *a, size_t n,
    size_t es, int (*compar) (const void *, const void *));
```

Поиск элемента KEY в массиве A, используя функцию сравнения CMP. Массив A содержит N элементов. Каждый элемент имеет размер ES байтов.

Также реализованы стандартные функции преобразования строки в число:

```
int atoi (const unsigned char *str);
```

Преобразование строки STR в целое число со знаком. Десятичная система счисления.

```
long atol (const unsigned char *str);
```

Преобразование строки STR в длинное целое число со знаком. Десятичная система счисления.

```
long strtol (const unsigned char *str,
    unsigned char **endptr, int base);
```

Преобразование строки STR в длинное целое число со знаком. Если указатель ENDPTR ненулевой, в нём сохраняется адрес символа, следующего за последним прочитанным. Значение BASE задаёт основание системы счисления, от 2 до 36. Если BASE равно 0, распознаются префиксы в стиле языка Си: 0 для восьмеричного числа и 0x для шестнадцатеричного.

```
unsigned long strtoul (const unsigned char *str,
    unsigned char **endptr, int base);
```

Преобразование строки STR в длинное беззнаковое целое число. Если указатель ENDPTR ненулевой, в нём сохраняется адрес символа, следующего за последним прочитанным. Значение BASE задаёт основание системы счисления, от 2 до 36. Если BASE равно 0, распознаются префиксы в стиле языка Си: 0 для восьмеричного числа и 0x для шестнадцатеричного.

8.4. Классификация символов

Для быстрой классификации символов имеется набор эффективных inline-функций. Они возвращают ненулевое значение, если символ соответствует требуемому типу. В качестве аргумента можно использовать только значения из диапазона -1...255.

`small_uint_t isdigit (int c);`	Является ли символ цифрой 0...9.
`small_uint_t isxdigit (int c);`	Является ли символ шестнадцатеричной цифрой, т.е. одним из 0...9 a...f A...F.
`small_uint_t isalpha (int c);`	Проверяет символ на принадлежность к алфавитным символам a...z A...Z \240...\377.
`small_uint_t islower (int c);`	Является ли символ буквой нижнего регистра a...z.
`small_uint_t isupper (int c);`	Является ли символ буквой верхнего регистра A...Z.
`small_uint_t isalnum (int c);`	Проверяет символ на принадлежность к текстовым символам 0...9 a...z A...Z \240...\377.
`small_uint_t toupper (int c);`	Преобразует символ в заглавный. Если это не символ a...z, результат непредсказуем.
`small_uint_t tolower (int c);`	Преобразует символ в строчный. Если это не символ A...Z, результат непредсказуем.
`small_uint_t isspace (int c);`	Проверяет, принадлежит ли символ к разряду пробелов, а именно: пробел, символ перевода страницы \f, "новая строка" \n, "перевод каретки" \r, "горизонтальная табуляция" \t и "вертикальная табуляция" '\v'.
`small_uint_t ispunct (int c);`	Проверяет, принадлежит ли символ к знакам пунктуации: отображаемый, но не буква и не цифра.
`small_uint_t iscntrl (int c);`	Проверяет, является ли символ управляющим, т.е. из диапазона \0...\37 \177...\237, но не пробел.
`small_uint_t isprint (int c);`	Проверяет, является ли символ печатаемым (включая пробел). То есть не управляющий и не -1 (EOF).
`small_uint_t isgraph (int c);`	Проверяет, является ли символ печатаемым (не пробелом). То есть не пробел, не управляющий и не -1 (EOF).

8.5. Макрос верификации

Макрос assert() для проверки логических условий в процессе выполнения программы. С точки зрения программиста макрос выглядит как функция:

```
void assert (int expression);
```

При выполнении вычисляется значение выражения **`expression`** и, если оно ложно (то есть

равно 0), выполнение программы останавливается и в поток отладочной печати (debug) выдаётся следующая информация:

- текст выражения **expression**
- имя файла с исходным кодом
- номер строки файла с исходным кодом
- имя текущей выполняемой функции языка Си

Чтобы отключить проверку, не требуется убирать вызов assert() из кода программы. Достаточно пересобрать приложение, добавив для компилятора флаг **-DNDEBUG**. Если же проверка должна выполняться всегда, независимо от режима компиляции NDEBUG, следует использовать другой макрос:

```
void assert_always (int expression);
```

Пример использования. Предположим, мы имеем файл “myalloc.c”:

```
#include <runtime/lib.h>
#include <mem/mem.h>

void myalloc (mem_pool_t *pool)
{
    char *data = mem_alloc (pool, 100);
    assert (data != 0);
    ...
}
```

В данном примере, при отсутствии места для выделения памяти, assert() сработает и выдаст следующую информацию:

```
Assertion failed in function `myalloc':
myalloc.c, 7: data != 0
```

8.6. Математические функции и константы

Часто употребляемые математические константы:

Тип double	Тип long double	Значение
M_E	M_El	e
M_LOG2E	M_LOG2El	log_2 e
M_LOG10E	M_LOG10El	log_10 e
M_LN2	M_LN2l	log_e 2
M_LN10	M_LN10l	log_e 10
M_PI	M_PIl	pi
M_PI_2	M_PI_2l	pi/2
M_PI_4	M_PI_4l	pi/4
M_1_PI	M_1_PIl	1/pi
M_2_PI	M_2_PI	2/pi
M_2_SQRTPI	M_2_SQRTPIl	2/sqrt(pi)
M_SQRT2	M_SQRT2l	sqrt(2)
M_SQRT1_2	M_SQRT1_2l	1/sqrt(2)

Граничные значения вещественных чисел:

MAXFLOAT	Максимальное значение вещественного числа одинарной точности.
HUGE_VAL	“Плюс бесконечность” двойной точности.
HUGE_VALF	“Плюс бесконечность” одинарной точности.
HUGE_VALL	“Плюс бесконечность” повышенной точности.
INFINITY	“Плюс бесконечность” двойной точности.
NAN	Не-число, обозначает неопределённый результат вычисления.

Математические функции:

`int abs (int x);`	Абсолютное значение (модуль) целого числа.
`slong labs (long x);`	Абсолютное значение (модуль) длинного целого числа.
`double pow (double x, double y);`	Возведение числа X в степень Y.
`double sqrt (double x);`	Вычисление квадратного корня двойной точности.
`float sqrtf (float x);`	Вычисление квадратного корня одинарной точности.
`double modf (double x, double *intp);`	Разделение числа X на целую и дробную части.
`int isnan (double x);`	Проверка числа на неопределённое значение.
`int isinf (double x);`	Проверка числа на бесконечность.

8.7. Нелокальные переходы

Функции setjmp() и longjmp() обеспечивают возможность выполнения нелокального перехода и обычно используются для передачи управления к обработке ошибок в ранее вызванной процедуре.

```
int setjmp (jmp_buf label);
```

При вызове функция setjmp() создаёт нелокальную метку: сохраняет состояние стека и регистров в массиве LABEL и возвращает значение 0. Позже состояние выполнения может быть возвращено к этой точке посредством использования функции longjmp(). В этом случае функция setjmp() вернёт ненулевое значение.

```
void longjmp (jmp_buf label, int val);
```

Функция longjmp() выполняет переход к нелокальной метке: восстанавливает состояние стека, ранее сохраненное в массиве LABEL функцией setjmp(). Значение VAL будет возвращено функцией setjmp() в качестве результата. Если VAL равно 0, будет возвращено значение 1.

Пример:

```
#include <runtime/lib.h>

jmp_buf label;

int main()
{
    int error = setjmp (label);
    if (error != 0) {
        printf ("fatal error %d", error);
        uos_halt (1);
    }
    …
    func();
    …
```

```
}

void func()
{
    int error = 0;
    …
    if (error != 0)
        longjmp (label, error);
    …
}
```

8.8. Отладочная печать

Систем апредоставляет возможность вывода отладочной текстовой информации на консольный порт UART. Обычно этот порт оснащается интерфейсом RS-232. Формат выдачи асинхронный, 8 битов в байте, без чётности, один стоповый бит. Скорость обычно 115200 бит/сек (зависит от типа процессора). Отладочный вывод можно также перенаправить на произвольное логическое устройство, например в графическое окно или виртуальный последовательный порт USB. Функции отладочной печати можно вызывать как с открытыми, так и с закрытыми прерываниями, например из быстрого обработчика прерываний.

Функции вывода:

```
void debug_putc (char c);
```

Выдача одного байта в консольный порт.

```
void debug_puts (const char *str);
```

Выдача строки символов.

```
int debug_printf (const char *fmt, ...);
```

Выдача строки по формату с параметрами.

```
int debug_vprintf (const char *fmt, va_list args);
```

Выдача строки по формату с параметрами по указателю **va_list**.

```
void debug_dump (const char *caption, void *data, unsigned len);
```

Печать произвольного байтового массива.

Функции ввода:

```
unsigned short debug_getchar (void);
```

Ввод символа из консольного порта с ожиданием.

```
int debug_peekchar (void);
```

Ввод символа без ожидания. Если символ отсутствует, возвращается значение -1.

Функции управления:

```
extern bool_t debug_onlcr;
```

Флаг трансляции \n → \r\n при выводе. По умолчанию установлен. Трансляцию можно отменить, установив флаг в 0.

```
void debug_redirect (void (*func) (void *arg, short c), void *arg);
```

Перенаправления отладочного вывода. Для каждого выдаваемого байта будет вызываться функция FUNC. Первым аргументов вызова будет параметр ARG, вторым аргументом – выдаваемый символ.

Для работы с отладочным портом можно также использовать функции модуля stream (см. раздел “Модуль stream – потоковый ввод-вывод”). Структура данных отладочного потока задана глобальной переменной:

```
extern stream_t debug;
```

Пример выдачи:

```
printf (&debug, "Hello World\n");
```

8.9. Обращение к аппаратным портам и специальныи регистрам микроконтроллера

Для архитектуры MIPS обращение к специальныи регистрам процессора выполняется посредством следующих inline-функций:

`void *mips_get_stack_pointer (void);`	Чтение регистра SP – указателя стека.
`void mips_set_stack_pointer (void *x);`	Запись в регистр SP – указатель стека.
`int mips_read_c0_register (int reg);`	Чтение произвольного регистра управления сопроцессора 0.
`void mips_write_c0_register (int reg, int value);`	Запись в произвольный регистр управления сопроцессора 0.
`int mips_read_fpu_register (int reg);`	Чтение произвольного регистра данных процессора с плавающей точкой.
`void mips_write_fpu_register (int reg, int value);`	Запись в произвольный регистр данных процессора с плавающей точкой.
`int mips_read_fpu_control (int reg);`	Чтение произвольного регистра управления процессора с плавающей точкой.
`void mips_write_fpu_control (int reg, int value);`	Запись в произвольный регистр управления процессора с плавающей точкой.
`void mips_intr_disable (int *x);`	Общий запрет прерываний. Предыдущее значение режима прерываний сохраняется в указанной переменной (регистр C0_STATUS).
`void mips_intr_restore (int x);`	Восстановление режима прерываний из указанной переменной.
`void mips_intr_enable (void);`	Общее разрешение прерываний.
`int mips_count_leading_zeroes (unsigned x);`	Вычисление количества нулевых битов в левой части 32-битного значения. Используется инструкция процессора CLZ.

Для непосредственного доступа к аппаратным портам микроконтроллера имеются соответствующие макросы. Пример записи в регистры CSCON для настройки внешней памяти процессора NVCom-01:

```
/* Внешняя Flash-память шириной 32 бита на сигнале выборки nCS3. */
MC_CSCON3 = MC_CSCON_WS (3);        /* Количество wait states */

/* Внешняя память SDRAM шириной 32 бита на nCS0. */
MC_CSCON0 = MC_CSCON_E |            /* Разрешение nCS0 */
      MC_CSCON_T |                  /* Синхронная память */
      MC_CSCON_CSBA (0x00000000) |  /* Базовый адрес */
      MC_CSCON_CSMASK (0xF8000000); /* Маска адреса */

MC_SDRCON = MC_SDRCON_PS_512 |      /* Размер страницы SDRAM */
      MC_SDRCON_CL_3 |              /* Задержка CAS = 3 такта */
      MC_SDRCON_RFR (64000000/8192, /* Период refresh */
            MPORT_KHZ);             /* Частота внешней шины */

MC_SDRTMR = MC_SDRTMR_TWR (2) |     /* Write recovery delay */
      MC_SDRTMR_TRP (2) |           /* Минимальный период Precharge */
      MC_SDRTMR_TRCD (2) |          /* Между Active и Read/Write */
      MC_SDRTMR_TRAS (5) |          /* Между * Active и Precharge */
      MC_SDRTMR_TRFC (15);          /* Интервал между Refresh */

MC_SDRCSR = 1;                      /* Инициализация SDRAM */
```

Полный список аппаратных портов и их битовых полей для процессоров НТЦ «Элвис» находится в следующих файлах:

sources/runtime/mips/io-elvees.h	Регистры, общие для всех процессоров НТЦ «Элвис» с архитектурой MIPS
sources/runtime/mips/io-mc24.h	Регистры для процессора MC-24
sources/runtime/mips/io-nvcom01.h	Регистры для процессора NVCom-01
sources/runtime/mips/io-nvcom02.h	Регистры для процессора NVCom-02

8.10. Задержка времени

Для реализации точных задержек времени используется функции udelay() и mdelay().

```
void udelay (small_uint_t usec);
```

Задержка выполнения на указанное количество микросекунд.

```
void mdelay (small_uint_t msec);
```

Задержка выполнения на указанное количество миллисекунд.

Функции реализуют максимально возможную точность, достижимую на данном типе процессора. Рекомендуется применять их в драйверах и других аппаратно-ориентированных алгоритмах. Не следует применять их в программах уровня пользователя, так как при выполнении задержки процессорное время не освобождается для других потоков. В таких случаях нужно использовать вызов timer_delay() (см. раздел “Модуль timer – аппаратный таймер”).

8.11. Двусвязные списки

Для организации произвольных двусвязных списков реализован набор inline-функций и макросов. Все операции над списками выполняются очень быстро и не требуют динамического выделения памяти.

`typedef struct _list_t {`
`  struct _list_t *next;`
`  struct _list_t *prev;`
`} list_t;`
	Заголовок структуры данных, содержащий указатели на следующий и предыдущий элементы списка.

`void list_init (list_t *l);`	Инициализация списка.
`void list_prepend (list_t *l, list_t *elem);`	Вставка элемента в начало списка.
`void list_append (list_t *l, list_t *elem);`	Вставка элемента в конец списка.
`void list_unlink (list_t *elem);`	Удаление элемента из списка.
`bool_t list_is_empty (const list_t *l);`	Состоит ли элемент в каком-либо списке.
`list_t *list_first (const list_t *l);`	Получение первого элемента списка.
`list_iterate (i, head) { ... }`	Цикл прохода по списку от начала к концу.
`list_iterate_backward (i, head) { ... }`	Цикл прохода по списку от конца к началу.

Чтобы некоторую структуру данных можно было включать в список, она должна начинаться со специального поля item типа list_t. Пример:

```
#include <runtime/lib.h>

struct mydata {
    list_t item;
    int x;
};

list_t list;
struct mydata a, b, c;

int main()
{
    list_init (&list);                  // инициализируем пустой список
    list_init (&a.item); a.x = 'a';     // инициализируем элементы
    list_init (&b.item); b.x = 'b';
    list_init (&c.item); c.x = 'c';

    list_append (&list, &a.item);       // добавляем элементы
    list_append (&list, &b.item);
    list_prepend (&list, &c.item);      // список: c, a, b

    struct mydata *i;
    list_iterate (i, &list) {           // проход по списку
        debug_printf («%c», i->x);      // получим «cab»
    }
    return 0;
}
```

9. Модуль stream — потоковый ввод-вывод

Модуль stream реализует форматный и посимвольный ввод-вывод в стиле <stdio.h>.

`void putchar (void *stream, short c);`	Посылка одного байта.
`unsigned short getchar (void *stream);`	Ожидание и приём одного байта.
`int peekchar (void *stream);`	Приём одного байта без ожидания.
`void fflush (void *stream);`	Ожидание выдачи всех буферизованных символов.
`bool_t feof (void *stream);`	Выяснение, достигнут ли конец файла ввода.
`mutex_t *freceiver (void *stream);`	Адрес мутекса, получающего сигнал при приёме новых днных.
`int puts (void *stream, const char *str);`	Посылка строки.
`unsigned char *gets (void *u, unsigned char *str, int len);`	Приём строки.
`int printf (void *stream, const char *fmt, ...);`	Форматный вывод.
`int scanf (void *stream, const char *fmt, ...);`	Форматный ввод.
`int vprintf (void *stream, const char *fmt, va_list args);`	Форматный вывод с аргументами в виде списка stdarg.
`int vscanf (void *stream, const char *fmt, va_list args);`	Форматный ввод с аргументами в виде списка stdarg.

В качестве первого аргумента для функций stream можно использовать:

- Адрес стандартного внешнего объекта debug. Будет выполняться обращение к отладочному консольному порту. Например, вызов “`putchar (&debug, 'z');`” эквивалентен “`debug_putc ('z');`”.
- Адрес структуры данных драйвера UART.
- Адрес сокета TCP.
- Адрес структуры данных драйвера графического дисплея gpanel.
- Адрес master или slave, полученные при вызове pipe_init() - описаны ниже.
- Адрес структуры stream_buf_t – описана ниже.

Можно применять функции stream для вывода в текстовую строку или ввода из текстовой строки. Для этого необходимо использовать объект типа strem_buf_t и привязать его в конкретному строковому буферу функциями stropen/strclose. Например:

```
#include <runtime/lib.h>
#include <stream/stream.h>

void func ()
{
    stream_buf_t stream;
    unsigned char buf [100];

    stropen (&stream, buf, sizeof (buf));
    printf (&stream, «Example output: %c %c %c», 'a', 'b', 'c');
    strclose (&stream);
}
```

Имеются также более простые функции для форматного вывода в текстовую строку или ввода из текстовой строки:

```
int snprintf (unsigned char *buf,
    int size, const char *fmt, ...);
int vsnprintf (unsigned char *buf,
    int size, const char *fmt,
    va_list args);
int sscanf (const unsigned char *buf,
    const char *fmt, ...);
```

Форматный вывод в строку указанного размера.

Форматный вывод в строку указанного размера с аргументами в виде списка stdarg .

Форматный ввод из строки.

Функции stream можно использовать для обмена данными между задачами посредством двунаправленного потока байтов, похожего на каналы Unix. Создание такого канала делается вызовом функции:

```
pipe_t *pipe_init (char *buf, int bytes,
                   stream_t **master, stream_t **slave);
```

Здесь buf - буферный массив, в котором будут храниться пересылаемые данные, nbytes – его размер. После выполнения функции по указателям master и slave будут записаны адреса объектов stream_t, которые можно использовать в функциях putchar, printf, getchar и т.п.

Пример:

```
#include <runtime/lib.h>
#include <stream/stream.h>
#include <stream/pipe.h>

#define NPIPE_BYTES     32          // желаемый размер буфера

char pipe_buf [sizeof(pipe_t) + NPIPE_BYTES];
stream_t *master, *slave;

void func ()
{
    …
    pipe_init (pipe_buf, sizeof (pipe_buf), &master, &slave);
    …
    puts (master, «hello»);
    …
    c = getchar (slave);
    …
}
```

Полный текст примера можно найти в файле examples/linux386/test_pipe.c.

Можно использовать функции stream для работы со структурами данных пользователя или произвольного драйвера. Для этого структура данных должна первым элементов иметь указатель на специальную таблицу функций, например:

```
#include <runtime/lib.h>
#include <stream/stream.h>

struct userdata {
    const stream_interface_t *interface;  // интерфейс stream
    …                                     // данные пользователя
};

static void userdata_putchar (struct userdata *u, short c)
```

```
{
    …                                       // вывод байта
}

static unsigned short userdata_getchar (struct userdata *u)
{
    …                                       // ввод байта
}

static int userdata_peekchar (struct userdata *u)
{
    …                                       // опрос ввода байта
}

static void userdata_flush (struct userdata *u)
{
    …                                       // завершение вывода
}

static bool_t userdata_eof (struct userdata *u)
{
    …                                       // опрос конца ввода
}

static void userdata_close (struct userdata *u)
{
    …                                       // завершение коммуникации
}

const stream_interface_t userdata_interface = {
        .putc = (void (*) (stream_t*, short))    userdata_putchar,
        .getc = (unsigned short (*) (stream_t*)) userdata_getchar,
        .peekc = (int (*) (stream_t*))           userdata_peekchar,
        .flush = (void (*) (stream_t*))          userdata_flush,
        .eof = (bool_t (*) (stream_t*))          userdata_eof,
        .close = (void (*) (stream_t*))          userdata_close,
};

void userdata_init (struct userdata *u)
{
    u->interface = &userdata_interface;  // начальная инициализация
    ...
}
```

В качестве примера реализации интерфейса stream рекомендуется изучить файл sources/stream/pipe.c.

10. Модуль random — генератор псевдослучайных чисел

Простой генератор псевдослучайных чисел. Получаемые значения равномерно распределены в диапазоне 0...32767.

```
short rand15 (void);
```

Вычисление следующего 15-битного псевдослучайного числа.

```
void srand15 (unsigned short);
```

Установка начального значения генератора.

11. Модуль crc — вычисление контрольных сумм

Набор функций для вычисления контрольных сумм.

`#include <crc/crc8-atm.h>`
 	
`unsigned char crc8_atm (`
`  unsigned const char *buf,`
`  unsigned char len);`
	8-битная контрольная сумма, применяемая в сетях ATM.
	Полином: $x^8 + x^2 + x + 1$.

`#include <crc/crc8-dallas.h>`
 	
`unsigned char crc8_dallas (`
`  unsigned const char *buf,`
`  unsigned char len);`
	8-битная контрольная сумма, применяемая в устройствах iButton фирмы Dallas Semiconductor.
	Полином: $x^8 + x^5 + x^4 + 1$.

`#include <crc/crc16.h>`
 	
`unsigned short crc16 (`
`  unsigned short sum,`
`  unsigned const char *buf,`
`  unsigned short len);`
 	
`unsigned short crc16_byte (`
`  unsigned short sum,`
`  unsigned char byte);`
	16-битная контрольная сумма. Применяется в протоколах Bisync, Modbus, USB.
	Полином: $x^{16} + x^5 + x^2 + 1$.

`#include <crc/crc16-ccitt.h>`
 	
`unsigned short crc16_ccitt (`
`  unsigned short sum,`
`  unsigned const char *buf,`
`  unsigned short len);`
 	
`unsigned short crc16_ccitt_byte (`
`  unsigned short sum,`
`  unsigned char byte);`
	16-битная контрольная сумма, применяемая в сетях HDLC и X.21.
	Полином: $x^{16} + x^{12} + x^5 + 1$.

`#include <crc/crc16-inet.h>`
 	
`unsigned short crc16_inet (`
`  unsigned short sum,`
`  unsigned const char *buf,`
`  unsigned short len);`
 	
`unsigned short crc16_inet_header (`
`  unsigned char *src,`
`  unsigned char *dest,`
`  unsigned char proto,`
`  unsigned short proto_len);`
 	
`unsigned short crc16_inet_byte (`
`  unsigned short sum,`
`  unsigned char byte);`
	16-битная контрольная сумма для протокола IP.
	Полином: $x^{16} + 1$.

`#include <crc/crc32-vak.h>`
 	
`unsigned long crc32_vak (`
`  unsigned long sum,`
`  unsigned const char *buf,`
`  unsigned short len);`
	Эффективная 32-битная хэш-функция.

`unsigned long crc32_vak_byte (`
`  unsigned long sum,`
`  unsigned char byte);`	

12. Модуль regexp — сопоставление текстовых строк

Модуль regexp предназначен для сравнения и преобразования текстовых строк в соответствии с заданным шаблоном. Шаблон представляет собой регулярное выражение, соответствующее стандарту POSIX.

`typedef struct _regexp_t regexp_t;`	Тип для структуры данных регулярного выражения. Подробности скрыты от пользователя.
`unsigned regexp_size (` `  const unsigned char *pattern);`	Вычисление обьёма памяти, необходимого для регулярного выражения.
`bool_t regexp_compile (regexp_t *re,` `  const unsigned char *pattern);`	Компиляция регулярного выражения и помещение результата в заданную структуру regexp_t. Необходимо, чтобы в структуре имелось достаточное количество памяти, вычисленное с помощью функции regexp_size().
`bool_t regexp_execute (regexp_t *re,` `  const unsigned char *str);`	Сравнение строки с регулярным выражением. Регулярное выражение должно быть предварительно скомпилировано функцией regexp_compile(). Возвращает 1 в случае успеха, иначе 0.

`bool_t regexp_substitute (`
`  const regexp_t *re,`
`  const unsigned char *src,`
`  unsigned char *dst);`
	Выполнение замены после успешного сравнения. Возвращает 1 в случае успеха, иначе 0. Строка src должна содержать ссылки на найденные значения вида & \1 \2 \3 … \9. Они заменяются на соответствующие фрагменты из строки сравнения, обозначенные круглыми скобками “()”. Результат помещается в строку dst.

Регулярные выражения могут содержать следующие метасимволы:

`.`	Соответствует любому символу.
`*`	Соответствует повторению предыдущего символа нуль или более раз.
`+`	Соответствует повторению предыдущего символа один или более раз.
`?`	Соответствует повторению предыдущего символа нуль или один раз.
`^`	Соответствует началу строки.
`$`	Соответствует концу строки.
`[xyz]`	Соответствует любому символу из заключенных в квадратные скобки.
`[^xyz]`	Соответствует любому символу, кроме заключенных в квадратные скобки.
`[a-z]`	Соответствует любому символу в указанном диапазоне.
`[^a-z]`	Соответствует любому символу, кроме лежащих в указанном диапазоне.
`(pattern)`	Соответствует строке pattern и запоминает найденное соответствие.
`x|y`	Соответствует x или y.
`\<`	Левая граница слова.

\>	Правая граница слова.
\s	Отмена специального значения символа s.

13. Модуль kernel — задачи и мутексы

Модуль kernel реализует задачи, мутексы и обработку прерываний. Его функциональность подробно описана в разделе «7. Ядро системы».

Файл объявлений:

```
#include <kernel/uos.h>
```

Типы данных:

task_t	Задача, представляющая собой отдельный поток управления, со своим стеком и приоритетом.
mutex_t	Мутекс, служащий для синхронизации задач.
mutex_group_t	Група мутексов, предназначенная для ожидания готовности одного из мутексов в списке.
handler_t	Тип функции — быстрого обработчика прерываний.
array_t	Обобщённый массив памяти.

Тип array_t предназначен для выделения статической памяти под задачи и группы мутексов. Гарантируется корректное выравнивание на границу слова. Для объявления переменной нужно использовать макрос ARRAY(name,nbytes), например:

```
#include <runtime/lib.h>
#include <kernel/uos.h>

ARRAY (task, 1000);                         // выделение памяти для задачи
ARRAY (group, sizeof(mutex_group_t) +       // группа для четырех мутексов
              4 * sizeof(mutex_slot_t));
```

Управление задачами:

`task_t *task_create (void (*func)(void*), void *arg, const char *name, int priority, array_t *stack, unsigned stacksz);`	Создание задачи.
`void task_exit (void *status);`	Завершение текущей задачи.
`void task_delete (task_t *task, void *status);`	Принудительное завершение задачи.
`void *task_wait (task_t *task);`	Ожидание завершения задачи.
`int task_stack_avail (task_t *task);`	Определение размера части стека, не ипользуемой задачей.
`const char *task_name (task_t *task);`	Запрос имени задачи.
`int task_priority (task_t *task);`	Запрос приоритета задачи.
`void task_set_priority (task_t *task, int priority);`	Установка приоритета задачи.
`void *task_private (task_t *task);`	Запрос приватных данных.
`void task_set_private (task_t *task,`	Установка приватных данных задачи.

```
    void *privatep);
void task_yield (void);
```
Переключение на следующую задачу с тем же приоритетом (кооперативная многозадачность).

```
void task_print (stream_t *stream,
    task_t *t);
```
Отладочная печать информации о задаче.

```
unsigned task_fpu_control (task_t *t,
    unsigned mode, unsigned mask);
```
Управление режимами сопроцессора с плавающей точкой (FPU) для данной задачи.

Управление мутексами:

```
void mutex_lock (mutex_t *m);
```
Захват мутекса.

```
void mutex_unlock (mutex_t *m);
```
Освобождение мутекса.

```
bool_t mutex_trylock (mutex_t *m);
```
Захват мутекса без ожидания.

```
void mutex_signal (mutex_t *m,
    void *message);
```
Посылка сигнала мутексу.

```
void *mutex_wait (mutex_t *m);
```
Ожидание сигнала от мутекса.

Управление прерываниями:

```
void mutex_lock_irq (mutex_t *m,
    int irq, handler_t func, void *arg);
```
Захват мутекса и привязка к номеру аппаратного прерывания. Задание быстрого обработчика прерывания (необязательно).

```
void mutex_unlock_irq (mutex_t *m);
```
Освобождение мутекса и прекращение обслуживания аппаратного прерывания.

```
void mutex_attach_irq (mutex_t *m,
    int irq, handler_t func, void *arg);
```
Привязка мутекса к номеру аппаратного прерывания (без захвата). Задание быстрого обработчика прерывания (необязательно).

Управление группами мутексов:

```
mutex_group_t *mutex_group_init (
    array_t *buf, unsigned buf_size);
```
Инициализация группы мутексов.

```
bool_t mutex_group_add (
    mutex_group_t *group,
    mutex_t *mutex);
```
Добавление мутекса в группу.

```
void mutex_group_listen (
    mutex_group_t *group);
```
Начало ожидания сигнала от группы мутексов.

```
void mutex_group_unlisten (
    mutex_group_t *group);
```
Прекращение ожидания сигнала от группы мутексов.

```
void mutex_group_wait (
    mutex_group_t *group,
    mutex_t **pmutex,
    void **pmessage);
```
Получение сообщения от группы мутексов. Указатели на мутекс и сообщение помещаются в переменные *mutex и *message.

Выполнение приложения начинается с функции uos_init(), определяемой пользователем. Эта функция должна создать требуемое количество задач пользователя и инициализировать все подсистемы и драйверы, нужные для приложения. После завершения функции uos_init() включается многозадачное ядро и начинается выполнение задач,в соответствии с их приоритетом.

14. Модуль timer — аппаратный таймер

Модуль timer предоставляет возможность учёта времени жизни системы и переключения задач по истечению требуемого количества миллисекунд. Таймер не является обязательной частью системы. Для учёта времени используются прерывания от аппаратного таймера, встроенного в микроконтроллер. Периодичность прерываний определяется пользователем. Запуск таймера производится вызовом соответствующей функции на этапе инициализации (из функции uos_init()). Например:

```
#include <runtime/lib.h>
#include <timer/timer.h>

timer_t timer;                      // структура данных драйвера

#define MSEC_PER_TICK   100         // период между прерываниями

void uos_init()
{
    timer_init (&timer, KHZ, MSEC_PER_TICK);
    ...
}
```

Здесь KHZ — рабочая частота процессора в килогерцах. Третий параметр определяет гранулярность таймера: период между прерываниями в миллисекундах.

В структуре данных таймера есть два мутекса: **lock** и **decisec**. Мутекс **timer.lock** получает сигнал каждые **MSEC_PER_TICK** миллисекунд. Мутекс **timer.decisec** получает сигнал 10 раз в секунду. Можно использовать их для получения задержек на известное время в задачах пользователя, например:

```
for (;;) {
    mutex_wait (&timer.decisec);
    debug_puts («10 раз в секунду\n»);
}
```

Функции таймера:

```
void timer_init (timer_t *t,
    unsigned long khz,
    small_uint_t msec_per_tick);
```
Инициализация таймера.

```
void timer_delay (timer_t *t,
    unsigned long msec);
```
Задержка выполнения текущей задачи на указанное количество миллисекунд. Процессор освобождается для выполнения других задач.

```
unsigned long timer_milliseconds (
    timer_t *t);
```
Запрос времени жизни системы в миллисекундах.

```
unsigned int timer_days (timer_t *t,
    unsigned long *milliseconds);
```
Запрос времени жизни системы в сутках и миллисекундах.

```
bool_t timer_passed (timer_t *t,
    unsigned long t0, unsigned int msec);
```
Запрос, прошло ли **msec** миллисекунд с момента времени **t0**.

```
bool_t interval_greater_or_equal (
    long interval, long msec);
```
Параметр **interval** содержит разность двух значений времени в миллисекундах (возможно, отрицательный из-за переполнения). Функция определяет, превышает ли этот интервал указанной значение миллисекунд.

15. Модуль uart — асинхронные порты RS-232

Драйвер предназначен для обслуживания асинхронного приёмопередатчика UART, встроенного в процессор. Возможны два режима использования: двунаправленный поток быйтов или передача сетевых пакетов по протоколу SLIP.

Для работы в побайтовом режиме при старте системы из функции uos_init() необходимо вызвать функцию uart_init(). Она выполнит инициализацию аппаратных регистров приёмопередатчика, а также создаст задачу для обработки прерываний.

Функция инициализации:

```
void uart_init (uart_t *u, small_uint_t port, int prio,
            unsigned int khz, unsigned long baud);
```

Параметры:

- u – Структура данных для работы драйвера. Должна быть инициализирована нулевым значением.
- port – Номер порта UART, начиная с 0.
- prio – Приоритет для задачи обработки прерываний. Должен быть выше, чем у других задач, обращающихся к данному драйверу.
- khz – Базовая частота опорного генератора.
- baud – Требуемая скорость данных, бит/сек.

Пример:

```
…
#include <uart/uart.h>

uart_t uart;

void uos_init (void)
{
    …
    uart_init (&uart, 0, 50, KHZ, 9600);
}
```

Для выдачи и приёма данных следует использовать стандартные функции интерфейса stream_t:

- putchar (поток, символ) – посылка одного байта
- getchar (поток) – ожидание и приём одного байта
- peekchar (поток) – приём одного байта без ожидания
- fflush (поток) – ожидание выдачи всех буферизованных символов
- puts (поток, строка) – посылка строки
- gets (поток, буфер, размер) – приём строки
- printf (поток, формат, …) - форматный вывод
- vprintf (поток, формат, аргументы) – форматный вывод stdarg

Пример:

```
puts (&uart, "Hello, World!\r\n");
printf (&uart, "UART frame errors = %u\r\n", uart.frame_errors);
```

Для работы по протоколу SLIP при старте системы из функции uos_init() необходимо вызвать функцию иниуиализации slip_init(). Она выполнит инициализацию аппаратных регистров

приёмопередатчика, а также создаст две задачи для обработки прерываний по приёму и передаче.

Функция инициализации:

```
void slip_init (slip_t *u, small_uint_t port, const char *name, int prio,
            mem_pool_t *pool, unsigned int khz, unsigned long baud);
```

Параметры:

- u – Структура данных для работы драйвера. Должна быть инициализирована нулевым значением.
- port – Номер порта UART, начиная с 0.
- name – Имя сетевого интерфейса, например “sl0”.
- prio – Приоритет для задачи обработки прерываний. Должен быть выше, чем у других задач, обращающихся к данному драйверу.
- pool – Область памяти для размещения сетевых пакетов.
- khz – Базовая частота опорного генератора.
- baud – Требуемая скорость данных, бит/сек.

Функция приёма пакета:

```
buf_t *slip_recv (slip_t *u);
```

Функция передачи пакета:

```
bool_t slip_recv (slip_t *u, buf_t *pkt);
```

16. Модуль mem — динамическое выделение памяти

Модуль mem обеспечивает динамическое выделение памяти. Он не является обязательной частью системы: в частности, модули runtime и kernel не используют динамическое выделение памяти.

Память выделяется из ***пула памяти***, выполненного в виде структуры **`mem_pool_t`**. Можно иметь в системе несколько независимых пулов памяти. Обычно в начале работы (из функции uos_init()) вся доступная память организуется в один или несколько пулов вызовом функции mem_init().

`void mem_init (mem_pool_t *region, size_t start, size_t stop);`	Добавление области свободной памяти к пулу. Пул может состоять из нескольких областей памяти. Добавлять их следует в порядке увеличения адресов.
`void *mem_alloc (mem_pool_t *region, size_t bytes);`	Выделение участка памяти из пула. Если памяти недостаточно, возвращает 0. Выделенная память расписывается нулём.
`void *mem_xalloc (mem_pool_t *region, size_t bytes, const char *title);`	Выделение участка памяти из пула. Если памяти недостаточно, останавливает работу системы вызовом uos_halt(). Никогда не возвращает 0. Выделенная память расписывается нулём.
`void *mem_alloc_dirty (mem_pool_t *region, size_t bytes);`	Выделение участка памяти из пула. Если памяти недостаточно, возвращает 0. Выделенная память содержит мусор.

`void *mem_realloc (void *block, size_t bytes);`	Изменение размера выделенного ранее участка памяти. Если памяти недостаточно, возвращает 0. Выделенная память расписывается нулём.
`void mem_truncate (void *block, size_t bytes);`	Укорачивание выделенного ранее участка памяти.
`void mem_free (void *block);`	Освобождение выделенного ранее участка памяти.
`size_t mem_available (mem_pool_t *region);`	Запрос размера оставшейся свободной памяти в пуле.
`size_t mem_size (void *block);`	Запрос размера выделенного ранее участка памяти.
`mem_pool_t *mem_pool (void *block);`	Запрос указателя на пул, соответствующий выделенному ранее участку памяти.
`unsigned char *mem_strdup ( mem_pool_t *region, const unsigned char *s);`	Создание копии текстовой строки. Если памяти недостаточно, возвращает 0.
`unsigned char *mem_strndup ( mem_pool_t *region, const unsigned char *s, size_t n);`	Создание копии текстовой строки с ограничением размера. Если памяти недостаточно, возвращает 0.

Пример:

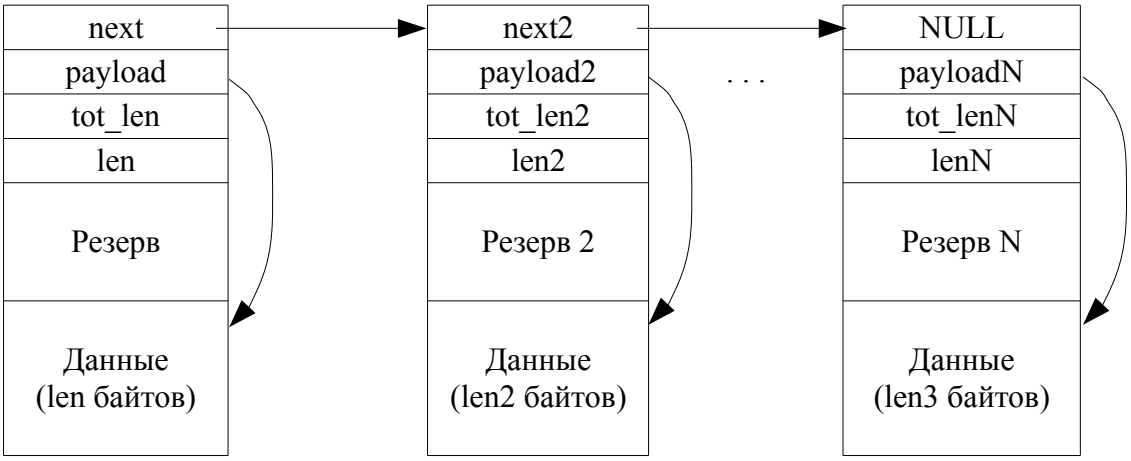
```
…
#include <mem/mem.h>

mem_pool_t pool;

void uos_init (void)
{
    …
    mem_init (&pool, 0xA0000000, 0xA4000000);  // 64 Мбайта внешней SRAM
    …
}
```

17. Модуль buf — управление цепочками буферов памяти

Модуль buf реализует управление памятью в виде списка участков памяти. Отдельные участки памяти выделяются динамически из пула. Такое представление используется стеком TCP/IP для хранения сетевых пакетов. Файлы заголовков:

`#include <buf/buf.h>`	Тип **`buf_t`** и функции работы с буферами.
`#include <buf/buf-queue.h>`	Тип **`buf_queue_t`** и функции работы с очередью буферов.
`#include <buf/buf-prio.h>`	Тип **`buf_prio_queue_t`** и функции работы с приоритезированной очередью буферов.

Буфер представляет собой односвязный список участков памяти с заголовками типа **`buf_t`**:



Поля структуры **`buf_t`**:

`buf_t *next;`	Указатель на следующий участок памяти.
`unsigned char *payload;`	Указатель на поле данных текущего участка памяти. Между заголовком **`buf_t`** и полем данных может располагаться резервная область.
`unsigned short tot_len;`	Суммарная длина данных этого и последующих участков списка.
`unsigned short len;`	Длина поля данных этого участка памяти.

Функции работы с буферами:

`buf_t *buf_alloc (struct _mem_pool_t *m,` **`  unsigned short size,`** **`  unsigned short reserved);`**	Динамическое выделение буфера из указанного пула памяти. Длина поля данных задаётся параметром size, размер резерва – параметром reserved. Память выделяется одним участком. Если памяти недостаточно, возвращается 0.
`small_int_t buf_free (buf_t *p);`	Освобождение всей цепочки памяти буфера. Возвращает количество участков памяти в цепочке.
`void buf_truncate (buf_t *p,` **`  unsigned short size);`**	Укорачивание буфера до указанной длины.
`bool_t buf_add_header (buf_t *p,`	Изменение резервной области буфера. При

`short size);`	положительном значении параметра size резервная область уменьшается, а поле данных увеличивается; при отрицательном - наоборот. Если операция невозможна, возвращается 0.
`void buf_chain (buf_t *h, buf_t *t);`	Добавление буфера t в конец буфера h.
`buf_t *buf_dechain (buf_t *p);`	Удаление первого участка буфера. Возвращает указатель на новый заголовок укороченного буфера.
`buf_t *buf_make_continuous (buf_t *h);`	Преобразование буфера в единый непрерывный участок памяти. Если операция невозможна, возвращается 0 и память буфера освобождается.
`buf_t *buf_copy (buf_t *h);`	Создание копии буфера в виде единого непрерывного участка памяти. Если операция невозможна, возвращается 0.
`small_int_t buf_chain_len (buf_t *p);`	Вычисление количества участков в цепочке буфера.
`unsigned short buf_chksum (buf_t *p, unsigned short sum);`	Вычисление контрольной суммы данных буфера для протокола IP.
`void buf_print (buf_t *p);`	Отладочная печать буфера.
`void buf_print_data (unsigned char *data, int len);`	Отладочная печать массива байтов.
`void buf_print_ethernet (buf_t *p);`	Отладочная печать сетевого пакета Ethernet.
`void buf_print_ip (buf_t *p);`	Отладочная печать сетевого пакета IP.
`void buf_print_tcp (buf_t *p);`	Отладочная печать сетевого пакета TCP.

При работе с сетевыми пакетами удобно хранить их в виде очереди FIFO. Модуль buf содержит реализацию очереди буферов в виде структуры **`buf_queue_t`**. Функции для работы с очередью буферов:

`void buf_queue_init (buf_queue_t *q, buf_t **buf, int bytes);`	Начальная инициализация очереди. Параметр buf задаёт массив указателей на буферы для хранения очереди, параметр bytes – его размер.
`void buf_queue_put (buf_queue_t *q, buf_t *b);`	Занесение буфера в очередь. Очередь не должна быть заполненной, иначе последует непредсказуемое поведение.
`buf_t *buf_queue_get (buf_queue_t *q);`	Извлечение буфера из очереди. Если очередь пуста, возвращает 0.
`bool_t buf_queue_is_full (buf_queue_t *q);`	Проверка, заполнена ли очередь.
`bool_t buf_queue_is_empty (buf_queue_t *q);`	Проверка, пуста ли очередь.
`void buf_queue_clean (buf_queue_t *q);`	Опустошение очереди, освобождение памяти имеющихся буферов.

Обработка сетевых пакетов IP или Ethernet часто требует разделять сетевые пакеты по уровню приоритета QOS (quality of service). В модуле buf имеется реализация очереди буферов с поддержкой восьми уровней приоритета. Функции для работы с приоритезарованной очередью буферов:

`void buf_prio_queue_init (buf_prio_queue_t *q,`	Начальная инициализация очереди. Параметр

`buf_t **buf, int bytes);`	buf задаёт массив указателей на буферы для хранения очереди, параметр bytes – его размер.
`buf_t *buf_prio_queue_get (` `buf_prio_queue_t *q);`	Извлечение из очереди буфера с наибольшим приоритетом. Если очередь пуста, возвращает 0.

`int buf_prio_queue_put (`
`buf_prio_queue_t *q,`
`buf_t *p, int prio);`
	Занесение в очередь буфера с указанным приоритетом. Если в очереди нет места, возвращает 0. Если пакет поставлен в очередь, возвращает 1. Если для занесения пакета пришлось удалить другой пакет с меньшим (или таким же) приоритетом, возвращает 2.

`buf_t *buf_prio_queue_exchange (`
`buf_prio_queue_t *q,`
`buf_t *p, int prio);`
	Обмен пакета на другой с большим приоритетом, или на более старый с тем же приоритетом. Функция всегда возвращает ненулевой указатель.

18. Модуль net — реализация сетевого стека TCP/IP

Модуль net реализует сетевой стек протоколов IPv4, ICMP, ARP, UDP, TCP, Telnet. За основу взят пакет lwIP, созданный Адамом Дункельсом (Adam Dunkels).

Файлы заголовков сетевого стека:

`#include <net/ip.h>`	Протоколы IP, ICMP.
`#include <net/udp.h>`	Протокол UDP.
`#include <net/tcp.h>`	Протокол TCP.
`#include <net/telnet.h>`	Протокол Telnet.
`#include <net/arp.h>`	Протокол ARP.
`#include <net/netif.h>`	Обобщённый интерфейс к драйверу сетевого адаптера.
`#include <net/route.h>`	Работа с таблицей маршрутизации.

Типы данных сетевого стека:

Тип	**Файл**	**Описание**
`ip_t`	`#include <net/ip.h>`	Данные сетевого стека IP.
`udp_socket_t`	`#include <net/udp.h>`	Сеанс (сокет) протокола UDP.
`tcp_socket_t`	`#include <net/tcp.h>`	Сеанс (сокет) протокола TCP.
`tcp_stream_t`	`-- // --`	Stream-интерфейс к сокету TCP.
`arp_t`	`#include <net/arp.h>`	Таблица протокола ARP.
`netif_t`	`#include <net/netif.h>`	Данные обобщённого сетевого адаптера.
`netif_interface_t`	`-- // --`	Интерфейс обобщённого сетевого адаптера.
`route_t`	`#include <net/route.h>`	Строка таблицы маршрутизации.

Инициализация стека IP:

```
void ip_init (ip_t *ip, mem_pool_t *pool, int prio,
    timer_t *timer, arp_t *arp, mutex_group_t *g);
```

При инициализации создаётся отдельная задача с указанным приоритетом, отвечающая за обработку принятых пакетов и срабатывание таймаутов ARP и TCP.

Параметры:

- ip – Структура данных для работы драйвера. Должна быть инициализирована нулевым значением.
- pool – Пул памяти для размещения сетевых пакетов.
- prio – Приоритет для задачи обработки прерываний. Должен быть ниже, чем у сетевых адаптеров, но выше, чем у задач пользователя.
- timer – Таймер для отслеживания таймаутов.
- arp – Таблица протокола ARP (или 0 если протокол ARP не используется).
- g – Группа мутексов, содержащая все сетевые адаптеры и таймер.

Пример старта сетевого стека с адаптером Ethernet:

```
#include <runtime/lib.h>
#include <mem/mem.h>
#include <buf/buf.h>
#include <timer/timer.h>
#include <net/ip.h>
#include <net/route.h>
#include <elvees/eth.h>

#define SDRAM_START     0x80000000        // Начало внешней памяти
#define SDRAM_SIZE      (64*1024*1024)    // Всего 64 мегабайта

#define PRIO_IP   40    // Приоритет задачи IP
#define PRIO_ETH  70    // Приоритет драйвера Ethernet

mem_pool_t pool;        // Пул памяти
timer_t    timer;       // Системный таймер
ip_t       ip;          // Протокол IP
arp_t     *arp;         // Протокол ARP
eth_t      eth;         // Драйвер Ethernet
route_t    route;       // Маршрут для локальной сети

// IP-адрес данного устройства.
static unsigned char ip_addr [4] = { 192, 168, 20, 222 };

// MAC-адрес для сетевого адаптера Ethernet.
// Должен быть индивидуальным у каждого экземпляра устройства!
const unsigned char mac_addr [6] = { 0, 9, 0x94, 0xf1, 0xf2, 0xf3 };

// Группа мутексов – список сетевых адаптеров.
ARRAY (group, sizeof(mutex_group_t) + 4 * sizeof(mutex_slot_t));

// Таблица протокола ARP.
ARRAY (arp_data, sizeof(arp_t) + 10 * sizeof(arp_entry_t));

void uos_init (void)
{
    ...
    // Инициализация пула памяти.
    mem_init (&pool, SDRAM_START, SDRAM_START + SDRAM_SIZE);

    // Запуск системного таймера.
    timer_init (&timer, KHZ, 50);

    // Группа из двух мутексов: таймер и адаптер Ethernet.
```

```
        mutex_group_t *g = mutex_group_init (group, sizeof(group));
        mutex_group_add (g, &eth.netif.lock);
        mutex_group_add (g, &timer.decisec);

        // Инициализация таблицы ARP.
        arp = arp_init (arp_data, sizeof(arp_data), &ip);

        // Запуск протокола IP.
        ip_init (&ip, &pool, PRIO_IP, &timer, arp, g);

        // Драйвер адаптера Ethernet.
        eth_init (&eth, "eth0", PRIO_ETH, &pool, arp, mac_addr);

        // Установка маршрута для локальной сети.
        route_add_netif (&ip, &route, ip_addr, 24, &eth.netif);
        ...
    }
```

Другие функции протокола IP обычно не предназначены для вызова пользователем:

```
void ip_input (ip_t *ip, buf_t *p,
    netif_t *inp);
```
Обработка принятого пакета IP.

```
bool_t ip_output (ip_t *ip, buf_t *p,
    unsigned char *dest,
    unsigned char *src,
    small_uint_t proto);
```
Отправка IP-пакета. Параметр dest задаёт IP-адрес получателя, параметр src – IP-адрес отправителя (необязательный), параметр proto - протокол. Нужный сетевой интерфейс определяется по таблице маршрутизации. Функция возвращает 1, если пакет успешно поставлен в очередь на передачу. Если нет маршрута или очередь переполнена, пакет удаляется и функция возвращает 0.

```
bool_t ip_output_netif (ip_t *ip,
    buf_t *p, unsigned char *dest,
    unsigned char *src,
    small_uint_t proto,
    unsigned char *gateway,
    netif_t *netif,
    unsigned char *netif_ipaddr);
```
Аналогично ip_output(), но сетевой адаптер, его IP-адрес и адрес маршрутизатора явно заданы в параметрах. Поиск по таблице маршрутизации не производится.

Протокол IP в процессе работы собирает статистическую информацию, накапливаемую в полях структуры **ip_t**:

`unsigned long in_receives;`	IP: всего принято пакетов.
`unsigned long in_hdr_errors;`	IP: принято пакетов с ошибками в заголовке.
`unsigned long in_addr_errors;`	IP: принято пакетов с неизвестным адресом назначения.
`unsigned long in_discards;`	IP: потеряно пакетов из-за нехватки памяти.
`unsigned long in_unknown_protos;`	IP: принято пакетов с неизвестным протоколом.
`unsigned long in_delivers;`	IP: количество успешно доставленных пакетов.
`unsigned long out_requests;`	IP: всего запросов на передачу пакета.
`unsigned long out_discards;`	IP: потеряно передач из-за нехватки памяти.
`unsigned long out_no_routes;`	IP: потеряно пакетов из-за неизвестного адреса назначения.
`unsigned long forw_datagrams;`	IP: количество маршрутизированных пакетов.

`unsigned long icmp_in_msgs;`	ICMP: всего принято пакетов.
`unsigned long icmp_in_errors;`	ICMP: количество ошибок приёма.
`unsigned long icmp_in_echos;`	ICMP: принято эхо-запросов.
`unsigned long icmp_out_msgs;`	ICMP: всего передано пакетов.
`unsigned long icmp_out_errors;`	ICMP: количество ошибок передачи.
`unsigned long icmp_out_dest_unreachs;`	ICMP: потеряно пакетов из-за неизвестного адреса назначения.
`unsigned long icmp_out_time_excds;`	ICMP: потеряно пакетов из-за исчерпания времени жизни.
`unsigned long icmp_out_echo_reps;`	ICMP: передано эхо-ответов.
`unsigned long udp_out_datagrams;`	UDP: всего передано пакетов.
`unsigned long udp_in_datagrams;`	UDP: всего принято пакетов.
`unsigned long udp_in_errors;`	UDP: количество ошибок приёма.
`unsigned long udp_no_ports;`	UDP: потеряно пакетов из-за неизвестного номера порта.
`unsigned long tcp_out_datagrams;`	TCP: всего передано пакетов.
`unsigned long tcp_out_errors;`	TCP: количество ошибок передачи.
`unsigned long tcp_in_datagrams;`	TCP: всего принято пакетов.
`unsigned long tcp_in_errors;`	TCP: количество ошибок приёма.
`unsigned long tcp_in_discards;`	TCP: потеряно входящих соединений из-за нехватки памяти.

Структура **`netif_t`** содержит данные, общие для всех видов сетевых адаптеров. Каждый драйвер сетевого адаптера должен в своей структуре данных первым элементом иметь «**`netif_t netif;`**». В структуре **`netif_t`** имеется поле «**`netif_interface_t *interface;`**», которое должно указывать на интерфейс (список функций) конкретного адаптера.

Функции для работы с адаптерами:

`bool_t netif_output (netif_t *netif, buf_t *p, unsigned char *dest, unsigned char *src);`	Отправка IP-пакета. Параметр dest задаёт IP-адрес получателя, параметр src – IP-адрес отправителя (необязательный). MAC-адрес получателя определяется по ARP-таблице. Функция возвращает 1, если пакет успешно поставлен в очередь на передачу. Если нет MAC-адреса получателя или очередь переполнена, пакет удаляется и функция возвращает 0.
`bool_t netif_output_prio (netif_t *netif, buf_t *p, unsigned char *dest, unsigned char *src, small_uint_t prio);`	Аналогично netif_output(), но с указанием уровня приоритета QOS, от 0 до 7.
`buf_t *netif_input (netif_t *netif);`	Извлечение следующего принятого пакета из очереди адаптера. Возвращает 0, если нет принятых пакетов.
`void netif_set_address (netif_t *netif, unsigned char *macaddr);`	Установка локального MAC-адреса.

Сетевой адаптер в процессе функционирования должен накапливать статистическую информацию, которая содержится полях структуры **`netif_t`**:

`unsigned long in_packets;`	Всего принято пакетов.
`unsigned long in_bytes;`	Всего принято байтов.
`unsigned long in_errors;`	Количество ошибок приёма.
`unsigned long in_discards;`	Потеряно пакетов из-за нехватки памяти.
`unsigned long in_unknown_protos;`	Принято пакетов с неизвестным протоколом.
`unsigned long in_mcast_pkts;`	Принято multicast-пакетов.
`unsigned long out_packets;`	Всего передано пакетов.
`unsigned long out_bytes;`	Всего передано байтов.
`unsigned long out_discards;`	Потеряно передач из-за нехватки памяти.
`unsigned long out_errors;`	Количество ошибок передачи.
`unsigned long out_collisions;`	Количество коллизий при передаче.
`unsigned long out_mcast_pkts;`	Передано multicast-пакетов.

При работе в сетях Ethetnet для преоразования IP-адресов в MAC-адреса неоходимо использовать протокол ARP. Пространство для таблица адресов необходимо выделять при старте системы (из функции uos_init()), инициализировать его в виде объекта **`arp_t`** и передавать в качестве параметра функциям ip_init() и eth_init(). Функция инициализации протокола ARP:

```
arp_t *arp_init (array_t *buf, unsigned bytes, ip_t *ip);
```

Функция возвращает указатель на объект **`arp_t`**.

Параметры:

- buf – Массив данных, в котором будет организована таблица ARP-адресов.
- bytes – Размер массива buf в байтах.
- ip – Указатель на структуру данных стека IP.

Пример:

```
…
ip_t       ip;          // Протокол IP
arp_t     *arp;         // Протокол ARP

// Таблица протокола ARP на десять адресов.
ARRAY (arp_data, sizeof(arp_t) + 10 * sizeof(arp_entry_t));

void uos_init (void)
{
    …
    arp = arp_init (arp_data, sizeof(arp_data), &ip);
    …
}
```

Остальные функции протокола IP обычно не предназначены для пользователя:

```
buf_t *arp_input (netif_t *netif,
    buf_t *p);
bool_t arp_request (netif_t *netif,
    buf_t *p,
    unsigned char *dest,
    unsigned char *src);
```

Обработка принятого пакета ARP.

Отправка ARP-запроса. Параметр dest задаёт IP-адрес получателя, параметр src – IP-адрес сетевого адаптера. Функция возвращает 1, если пакет успешно поставлен в очередь на передачу. Если очередь переполнена, пакет удаляется и

функция возвращает 0.

```
bool_t arp_add_header (netif_t *netif,
    buf_t *p,
    unsigned char *ipdest,
    unsigned char *ethdest);
```

Добавление MAC-заголовка к отправляемому пакету. Параметр ipdest задаёт IP-адрес получателя, параметр ethdest – его MAC-адрес. Размер пакета увеличивается на 14 байтов. Функция возвращает 1, если заголовок успешно добавлен. Если очередь переполнена, пакет удаляется и функция возвращает 0.

```
unsigned char *arp_lookup (
    netif_t *netif,
    unsigned char *ipaddr);
```

Поиск MAC-адреса по IP-адресу в ARP-таблице. Если такой IP-адрес не найден, функция возвращает 0.

```
void arp_timer (arp_t *arp);
```

Отслеживание и удаление устаревающих записей из ARP-таблицы. Функция вызывается по таймеру 10 раз в секунду.

Таблица маршрутизации хранится в виде односвязного списка сртуктур типа **route_t**. Для добавления очередной строки в таблицу маршрутизации следует использовать функции:

```
void route_add_netif (ip_t *ip, route_t *r,
        unsigned char *ipaddr, unsigned char masklen,
        netif_t *netif);

bool_t route_add_gateway (ip_t *ip, route_t *r,
        unsigned char *ipaddr, unsigned char masklen,
        unsigned char *gateway);
```

Первая функция регистрирует локальный IP-адрес указанного сетевого интерфейса.

Вторая функция создаёт запись для маршрутизации сети по маске на указанный IP-адрес маршрутизатора. Нужный сетевой интерфейс определяется по таблице маршрутизации. Функция возвращает 1, если строка маршрутизации успешно создана, или 0 если требуемый сетевой интерфейс не найден.

Параметры:

- ip – Структура данных IP-стека.
- r – Создаваемая строка маршрутизации.
- ipaddr, masklen – IP-адрес и длина маски для сравнения с адресом назначения.
- netif – Сетевой интерфейс назначения.
- gateway – IP-адрес маршрутизатора назначения.

Остальные функции мершрутизации предназначены для использования протоколом IP:

```
netif_t *route_lookup (ip_t *ip,
    unsigned char *ipaddr,
    unsigned char **gateway,
    unsigned char **netif_ipaddr);
```

Поиск маршрута по IP-адресу. Функция возвращает указатель на интерфейс назначения, его IP-адрес и IP-адрес маршрутизатора.

```
netif_t *route_lookup_self (ip_t *ip,
    unsigned char *ipaddr,
    unsigned char *broadcast);
```

Предполагая, что пакет предназначен для локального устройства, функция находит сетевой интерфейс по его локальному IP-адресу. Функция возвращает указатель на интерфейс назначения, а также флаг broadcast.

```
unsigned char *route_lookup_ipaddr (
    struct _ip_t *ip,
    unsigned char *ipaddr,
    netif_t *netif);
```

Сетевой интерфейс может иметь несколько IP-адресов. Функция находит для заданного интерфейса IP-адрес, “ближайший” к требуемому (с большей длиной маски).

Сокет (соединение) UDP реализован в виде типа **udp_socket_t**. Для работы по протоколу UDP программа пользователя может использовать следующие функции:

```
void udp_socket (udp_socket_t *s,
    ip_t *ip,
    unsigned short port);
```

Создание сокета с указанным номером порта UDP.

```
void udp_close (udp_socket_t *s);
```

Завершение работы сокета UDP.

```
bool_t udp_sendto (udp_socket_t *s,
    buf_t *p,
    unsigned char *dest,
    unsigned short port);
```

Отправка пакета UDP по указанным IP-адресу и номеру порта. Функция возвращает 1, если пакет успешно поставлен в очередь на передачу. В случае неудачи пакет удаляется и функция возвращает 0.

```
buf_t *udp_recvfrom (udp_socket_t *s,
    unsigned char *from_addr,
    unsigned short *from_port);
```

Приём пакета UDP. Текущая задача пристанавливается до получения пакета. IP-адрес и номер порта отправителя сохраняются в переменных from_addr и from_port.

```
buf_t *udp_peekfrom (udp_socket_t *s,
    unsigned char *from_addr,
    unsigned short *from_port);
```

Приём пакета UDP без ожидания. Если принятые пакеты отсутствуют, функция возвращает 0. Иначе она возвращает первый принятый пакет. IP-адрес и номер порта отправителя сохраняются в переменных from_addr и from_port.

```
void udp_connect (udp_socket_t *s,
    unsigned char *ipaddr,
    unsigned short port);
```

Запоминание IP-адреса и номера порта получателя в структуре сокета для последующего использования функцией udp_send().

```
bool_t udp_send (udp_socket_t *s,
    buf_t *p);
```

Отправка пакета UDP по адресу, хранящемуся в сокете. Функция возвращает 1, если пакет успешно поставлен в очередь на передачу. В случае неудачи пакет удаляется и функция возвращает 0.

```
buf_t *udp_recv (udp_socket_t *s);
```

Приём пакета UDP. Текущая задача пристанавливается до получения пакета.

```
buf_t *udp_peek (udp_socket_t *s);
```

Приём пакета UDP без ожидания. Функция извлекает из очереди первый принятый пакет. Если принятые пакеты отсутствуют, функция возвращает 0.

Функция приёма пакета UDP предназначена для использования протоколом IP:

```
buf_t *udp_input (ip_t *ip, buf_t *p,
    netif_t *inp, ip_hdr_t *iph);
```

Обработка принятого пакета UDP.

Пример сервера UDP:

```
void udp_server_example (ip_t *ip, int serv_port)
{
    // Создание сокета с указанным номером порта.
    udp_socket_t sock;
    udp_socket (&sock, ip, serv_port);

    for (;;) {
        // Ожидание пакета от клиента.
        unsigned char client_addr [4];
        unsigned short client_port;
        buf_t *p = udp_recvfrom (&sock, client_addr, &client_port);

        // Обработка пакета p, подготовка ответа r.
        buf_t *r = process (p);
        buf_free (p);

        // Отправка ответа клиенту.
        udp_sendto (&sock, r, client_addr, client_port);
    }
    udp_close (&sock);
}
```

Пример клиента UDP:

```
void udp_client_example (ip_t *ip, unsigned char *serv_addr,
     int serv_port, int client_port)
{
    // Создание сокета с указанным номером порта.
    udp_socket_t sock;
    udp_socket (&sock, ip, client_port);

    // Подготовка запроса p.
    buf_t *p = ...;

    // Отправка пакета серверу.
    if (! udp_sendto (&sock, p, serv_addr, serv_port)) {
        error ("Error writing to socket");
    } else {
        // Ожидание ответа от сервера.
        buf_t *r = udp_recv (&sock);

        // Обработка ответа r.
        consume (r);
        buf_free (r);
    }
    udp_close (&sock);
}
```

Протокол TCP реализован в виде сокетов (соединений) типа **tcp_socket_t**. Для работы по протоколу TCP программа пользователя может использовать следующие функции:

```
tcp_socket_t *tcp_connect (ip_t *ip,
    unsigned char *ipaddr,
    unsigned short port);
```

Создание сокета TCP, соединение с сервером с указанными удалёнными IP-адресом и номером порта. Если соединение установить не удаётся, функция возвращает 0.

```
tcp_socket_t *tcp_listen (ip_t *ip,
    unsigned char *ipaddr,
    unsigned short port);
```

Создание серверного сокета, принимающего входящие соединения с указанными

локальными IP-адресом и номером порта..

```
tcp_socket_t *tcp_accept (
    tcp_socket_t *s);
```
Приём входящего соединения, создание сокета для передачи данных.

```
int tcp_close (tcp_socket_t *s);
```
Завершение работы сокета TCP. После вызова необходимо освободить память сокета функцией mem_free().

```
void tcp_abort (tcp_socket_t *s);
```
Аварийное прекращение работы сокета TCP. После вызова необходимо освободить память сокета функцией mem_free().

```
int tcp_read (tcp_socket_t *s,
     void *dataptr, unsigned short len);
```
Чтение данных из сокета TCP. Текущая задача пристанавливается до получения непустого количества данных. Возвращает количество прочитанных байтов, от 1 до len. При ошибке сокета возвращает -1.

```
int tcp_read_poll (tcp_socket_t *s,
    void *dataptr, unsigned short len,
    int nonblock);
```
Чтение данных из сокета TCP без ожидания. Возвращает количество прочитанных байтов, от 0 до len. При ошибке сокета возвращает -1.

```
int tcp_write (tcp_socket_t *s,
    const void *dataptr,
    unsigned short len);
```
Запись данных в сокет TCP. Текущая задача пристанавливается до помещения всех данных в очередь сокета. Возвращает количество записанных байтов (len). При ошибке сокета возвращает -1.

```
unsigned long tcp_inactivity (
    tcp_socket_t *s);
```
Запрос периода неактивности сокета TCP, в секундах.

Можно работать с сокетами TCP посредством функций модуля stream (printf, puts, getchar и другие). Для этого необходимо создать переменную типа **`tcp_stream_t`** и привязать её к сокету функцией:

```
stream_t *tcp_stream_init (
    tcp_stream_t *u,
    tcp_socket_t *sock);
```
Создание stream-интерфейса к сокету TCP.

Пример:

```
…
tcp_socket_t *sock;
sock = tcp_accept (listen_sock);    // Имеем сокет данных TCP

tcp_stream_t tcp_stream;            // Создаём stream-интерфейс к сокету
stream_t *stream;
stream = tcp_stream_init (&tcp_stream, sock);

putchar (stream, '?');              // Передаём данные

int c = getchar (stream);           // Принимаем данные
…
```

Несколько функций TCP предназначены для использования протоколом IP:

```
void tcp_input (ip_t *ip, buf_t *p,
    netif_t *inp, ip_hdr_t *iph);
```
Обработка принятого пакета TCP.

```
void tcp_fasttmr (ip_t *ip);
```
Выполнение действий по быстрому таймеру

`void tcp_slowtmr (ip_t *ip);`	(200 миллисекунд). Выполнение действий по медленному таймеру (500 миллисекунд).

Пример сервера TCP:

```
void tcp_server_example (ip_t *ip, int serv_port)
{
    tcp_socket_t *lsock = tcp_listen (ip, serv_port);
    if (! lsock) {
        error ("Error on listen");
        return;
    }
    for (;;) {
        tcp_socket_t *sock = tcp_accept (lsock);
        if (! sock) {
            error ("Error on accept");
            break;
        }
        char buffer [256];
        int n = tcp_read (sock, buffer, 256);
        if (n < 0) {
            error ("Error reading from socket");
        } else {
            buffer[255] = 0;
            printf ("Here is the message: %s\n", buffer);
            n = tcp_write (sock, "I got your message", 19);
            if (n < 0)
                error ("Error writing to socket");
        }
        tcp_close (sock);
        mem_free (sock);
    }
    tcp_close (lsock);
    mem_free (lsock);
}
```

Пример клиента TCP:

```
void tcp_client_example (ip_t *ip, unsigned char *serv_addr,
    int serv_port, char *message)
{
    tcp_socket_t *sock = tcp_connect (ip, serv_addr, serv_port);
    if (! sock) {
        error ("Error connecting");
        return;
    }
    char buffer [256];
    strncpy (buffer, message, 256);
    buffer[255] = 0;
    int n = tcp_write (sock, buffer, strlen (buffer) + 1);
    if (n < 0) {
        error ("Error writing to socket");
    } else {
        n = tcp_read (sock, buffer, 256);
        if (n < 0) {
            error ("Error reading from socket");
```

```
            } else {
                buffer[255] = 0;
                printf ("%s\n", buffer);
            }
        }
        tcp_close (sock);
        mem_free (sock);
    }
```

Для работы по протоколу Telnet служит функция:

`stream_t *telnet_init (tcp_socket_t *sock);`	Инициация Telnet-сессии по указанному сокету. Возвращает указатель на интерфейс stream. Создание stream-интерфейса к сокету TCP. Обмен данными производится функциями printf, puts, getchar и т.п.

19. Модуль snmp — сетевой протокол управления

Модуль snmp реализует сетевой протокол управления SNMPv2c в соответствии с рекомендациями RFC1901, RFC1905 и RFC1906. Структура информации упраления соответствует SMIv1 (рекомендации RFC1155, RFC1212, RFC1215).

Файлы заголовков протокола SNMP:

`#include <snmp/asn.h>`	Функции кодирования информации в соответствии со стандартом ASN.1.
`#include <snmp/snmp.h>`	Основные функции протокола SNMP для пользователя.
`#include <snmp/snmp-var.h>`	Макросы для объявления переменных управления.
`#include <snmp/snmp-vardecl.h>`	Макросы для размещения переменных управления.
`#include <snmp/snmp-vardef.h>`	Макросы для инициализации таблицы переменных управления.
`#include <snmp/snmp-system.h>`	Список общесистемных переменных управления.
`#include <snmp/snmp-netif.h>`	Список переменных управления для сетевых интерфейсов.
`#include <snmp/snmp-ip.h>`	Список переменных управления для протокола IP.
`#include <snmp/snmp-snmp.h>`	Список переменных управления для протокола SNMP.
`#include <snmp/snmp-icmp.h>`	Список переменных управления для протокола ICMP.
`#include <snmp/snmp-udp.h>`	Список переменных управления для протокола UDP.

Типы данных протокола SNMP:

Тип	**Файл**	**Описание**
`snmp_t`	`#include <snmp/snmp.h>`	Данные протокола SNMP.
`snmp_var_t`	`-- // --`	Описатель переменной управления.
`asn_t`	`#include <snmp/asn.h>`	Универсальный тип для хранения знаковых и беззнаковых чисел, строк, идентификаторов и списков.

Инициализация протокола SNMP:

```
    void snmp_init (snmp_t *snmp, mem_pool_t *pool, ip_t *ip,
            const snmp_var_t *tab, unsigned tab_size,
```

```
    unsigned enterprise, unsigned char services,
    const char *descr, const char *object_id,
    const char *resource_descr, const char *resource_id);
```

При инициализации устанвливается пароль чтения “public” (get community).

Параметры:

- snmp – Структура данных для работы драйвера. Должна быть инициализирована нулевым значением.
- pool – Пул памяти для размещения сетевых пакетов.
- ip – Стек протокола IP.
- tab – Таблица описателей переменных управления.
- tab_size – Размер таблицы описателей переменных в байтах.
- enterprise – Уникальный номер изготовителя устройства из списка “http://www.iana.org/assignments/enterprise-numbers”.
- services – Значение для переменной sysServices. Битовая маска, описывающая набор сервисов, выполняемых устройством.
- descr – Значение для переменной sysDescr. Текстовое описание устройства.
- object_id – Идентификатор данного типа устройств от изготовителя в виде текстовой строки.
- resource_descr – Значение для переменной sysORDescr. Текстовое описание возможностей устройства.
- resource_id – Идентификатор возможностей данного типа устройств от изготовителя в виде текстовой строки.

Другие функции протокола SNMP:

```
unsigned char *snmp_execute (
    snmp_t *snmp,
    unsigned char *input,
    unsigned insz,
    unsigned char *output,
    unsigned outsz);
```

Обработка запроса SNMP.

```
bool_t snmp_trap_v1 (snmp_t *snmp,
    udp_socket_t *sock,
    unsigned char *local_ip,
    unsigned trap_type,
    asn_t *oid, asn_t *value);
```

Отправка trap-сообщения в формате SNMPv1.

```
bool_t snmp_trap_v2c (snmp_t *snmp,
    udp_socket_t *sock,
    const char *trap_type,
    asn_t *oid, asn_t *value);
```

Отправка trap-сообщения в формате SNMPv2c.

Пример реализации протокола SNMP в устройстве:

```
...
#include <snmp/snmp.h>

//
// Загружаем объявления стандартных переменных управления,
// в соответствии с SMIv1.
//
#include <snmp/snmp-var.h>
#include <snmp/snmp-system.h>
```

```
#include <snmp/snmp-netif.h>
#include <snmp/snmp-ip.h>
#include <snmp/snmp-snmp.h>
#include <snmp/snmp-icmp.h>
#include <snmp/snmp-udp.h>

//
// Объявляем функции обращения к стандартным переменным управления.
//
#include <snmp/snmp-vardecl.h>
SYSTEM_VARIABLE_LIST
IF_VARIABLE_LIST
IP_VARIABLE_LIST
ICMP_VARIABLE_LIST
UDP_VARIABLE_LIST
SNMP_VARIABLE_LIST

//
// Инициализируем массив описателей переменных управления.
//
static const snmp_var_t snmp_tab [] = {
#include <snmp/snmp-vardef.h>
      SYSTEM_VARIABLE_LIST
      IF_VARIABLE_LIST
      IP_VARIABLE_LIST
      ICMP_VARIABLE_LIST
      UDP_VARIABLE_LIST
      SNMP_VARIABLE_LIST
      // Здесь можно добавлять свои переменные.
};

#define ENTERPRISE_OID  20520          // Идентификатор производителя

snmp_t snmp;                           // Структура данных протокола SNMP

//
// Задача приёма и обработки запросов SNMP.
// Приоритет должен быть наже, чем у задачи IP.
//
void snmp_task (void *data)
{
    // Открываем сокет для приёма SNMP-запросов.
    udp_socket_t sock;
    udp_socket (&sock, &ip, 161);

    for (;;) {
        // Принимаем запрос пользователя.
        unsigned char user_addr [4];
        unsigned short user_port;
        buf_t *p = udp_recvfrom (&sock, user_addr, &user_port);

        // Создаём пакет для ответа.
        buf_t *r = buf_alloc (&pool, 1500, 50);
        if (! r) {
            debug_printf ("out of memory!\n");
            buf_free (p);
            continue;
        }
```

```
            // Обрабатываем запрос SNMP и формируем ответ.
            unsigned char *output = snmp_execute (&snmp, p->payload, p->len,
                                                  r->payload, r->len);
            buf_free (p);
            if (! output) {
                // Ответа не будет: неправильный запрос или
                // не прошла аутентификация.
                buf_free (r);
                continue;
            }

            // Убираем лишнее и отсылаем ответ.
            buf_add_header (r, - (output - r->payload));
            udp_sendto (&sock, r, user_addr, user_port);
        }
    }

    void uos_init (void)
    {
        ...
        // Запуск сетевого стека IP (описан раньше).
        ...
        // Инициализация протокола SNMP.
        snmp_init (&snmp, &pool, &ip, snmp_tab, sizeof(snmp_tab),
            ENTERPRISE_OID, SNMP_SERVICE_REPEATER,
            "Testing SNMP", "1.3.6.1.4.1.20520.1.1",
            "Test", "1.3.6.1.4.1.20520.6.1");
        ...
    }
```

20. Модуль tcl — встраиваемый язык скриптов

Модуль tcl предоставляет возможность встраивать в приложения интерпретатор языка Tcl. Это позволяет использовать в устройстве Tcl-скрипты для гибкой конфигурации, адаптации к нуждам пользователя и т.п.

Основные типы и функции встраиваемого интерпретатора Tcl:

`Tcl_Interp`	Основная структура данных интерпретатора Tcl.
`Tcl_CmdBuf`	Буфер формируемой командной строки.
`Tcl_CmdProc`	Тип функции-исполнителя команды языка Tcl. Например: `int echo_cmd (void *arg, Tcl_Interp *interp, int argc, unsigned char **argv)`.
`Tcl_Interp *Tcl_CreateInterp (` `  mem_pool_t *pool);`	Создание экземпляра интерпрератора. Приложение может иметь несколько независимых копий интерпретатора Tcl.
`Tcl_CmdBuf Tcl_CreateCmdBuf (` `  mem_pool_t *pool);`	Создание буфера для формирования командной строки.
`void Tcl_DeleteInterp (` `  Tcl_Interp *interp);`	Завершение работы интерпретатора, освобождение памяти.
`void Tcl_DeleteCmdBuf (` `  Tcl_CmdBuf buffer);`	Удаление буфера командной строки.

`void Tcl_CreateCommand (`
`  Tcl_Interp *interp,`
`  unsigned char *cmd_name,`
	Создание новой команды языка. Параметр cmd_name задаёт имя команды, proc – вызываемйю функцию пользователя, proc_arg – первый аргумент

```
    Tcl_CmdProc *proc,
    void *proc_arg,
    Tcl_CmdDeleteProc *delete_proc);
```

функции. Параметр delete_proc определяет необязательную функцию, которая будет вызвана при удалении данной команды из интерпретатора.

```
unsigned char *Tcl_AssembleCmd (
    Tcl_CmdBuf buffer,
    unsigned char *string);
```

Добавление фрагмента к буферу командной строки. Возвращает указатель на строку, готовую к выполнению. Если строка еще не завершена, возвращает 0.

```
int Tcl_Eval (Tcl_Interp *interp,
    unsigned char *cmd,
    int flags,
    unsigned char **term_ptr);
```

Выполнение командной строки языка Tcl в интерпретаторе. Возвращает **`TCL_OK`** в случает успеха. Текстовый результат находится в переменной **`interp->result`**. Параметры flags и term_ptr следует задавать нулевыми (используются для внутренних нужд).

Полный список функций интерпретатора можно посмотреть в файле sources/tcl/tcl.h.

Пример реализации интерпретатора Tcl в устройстве:

```
...
#include <tcl/tcl.h>
...

//
// Реализация команды языка Tcl:
//      echo arg ...
//
int
echo_cmd (void *arg, Tcl_Interp *interp, int argc, unsigned char **argv)
{
    stream_t *stream = arg;
    int i;

    for (i=1; ; i++) {
        if (! argv[i]) {
            if (i != argc)
echoError:      snprintf (interp->result, TCL_RESULT_SIZE,
                    "argument list not NULL-terminated", argv[0]);
            break;
        }
        if (i >= argc)
            goto echoError;
        if (i > 1)
            putchar (stream, ' ');
        puts (stream, (char*) argv[i]);
    }
    putchar (stream, '\n');
    return TCL_OK;
}

//
// Ввод строки из канала stream.
//
unsigned char *
getline (stream_t *stream, unsigned char *buf, int len)
{
    int c;
```

```
    unsigned char *s;

    s = buf;
        while (--len > 0) {
        c = getchar (stream);
        if (feof (stream))
            return 0;
        if (c == '\b') {
            if (s > buf) {
                --s;
                puts (stream, "\b \b");
            }
            continue;
        }
        if (c == '\r')
            c = '\n';
        putchar (stream, c);
        *s++ = c;
        if (c == '\n')
            break;
    }
    *s = '\0';
    return buf;
}

//
// Запуск интерпретатора Tcl на указанном канале stream.
//
void
tcl_main (stream_t *stream)
{
    Tcl_Interp *interp;
    Tcl_CmdBuf buffer;
    unsigned char line [200], *cmd;
    int result, got_partial, quit_flag;

    puts (stream, "\n\nEmbedded TCL\n");
    puts (stream, "~~~~~~~~~~~~\n");
    printf (stream, "Free memory: %ld bytes\n\n", mem_available (&pool));

    interp = Tcl_CreateInterp (&pool);
    Tcl_CreateCommand (interp, (unsigned char*) "echo",
        echo_cmd, stream, 0);

    buffer = Tcl_CreateCmdBuf (&pool);
    got_partial = 0;
    quit_flag = 0;
    while (! quit_flag) {
        if (! got_partial) {
            puts (stream, "% ");
        }
        if (! getline (stream, line, sizeof (line))) {
            if (! got_partial)
                break;

            line[0] = 0;
        }
        cmd = Tcl_AssembleCmd (buffer, line);
        if (! cmd) {
```

```
            got_partial = 1;
            continue;
        }

        got_partial = 0;
        result = Tcl_Eval (interp, cmd, 0, 0);

        if (result != TCL_OK) {
            puts (stream, "Error");

            if (result != TCL_ERROR)
                printf (stream, " %d", result);

            if (*interp->result != 0)
                printf (stream, ": %s", interp->result);

            putchar (stream, '\n');
            continue;
        }

        if (*interp->result != 0)
            printf (stream, "%s\n", interp->result);
    }

    Tcl_DeleteInterp (interp);
    Tcl_DeleteCmdBuf (buffer);
}

//
// Задача пользователя: интерактивный интерпретатор Tcl на порту UART.
//
void
main_tcl (void *data)
{
    for (;;)
        tcl_main ((stream_t*) &uart);
}
...
```

21. Модуль elvees — драйверы для процессоров НПЦ Элвис

21.1. Драйвер UARTX

Драйвер предназначен для обслуживания трёхканального асинхронного приёмопередатчика, подключенного к процессору через внешнее прерывание /IRQ2. Каждый канал асинхронного приёмопередатчика имеет архитектуру, аналогичную порту UART процессора МС-24. Базовые адреса:

- канал 0 – адрес 0xB0001000
- канал 1 – адрес 0xB0002000
- канал 2 – адрес 0xB0003000

При старте системы из функции uos_init() необходимо вызвать функцию uartx_init(). Она выполнит инициализацию аппаратных регистров приёмопередатчика, а также создаст задачу для обработки прерываний. Также производится самопроверка регистров и прерываний. При

наличии ошибок на консоль выдаются диагностические сообщения.

Функция инициализации:

```
void uartx_init (uartx_t u[3], int prio, unsigned int khz,
                 unsigned long baud);
```

Параметры:

- u – Поле данных для работы драйвера. Должна быть инициализирована нулевым значением. Представляет собой массив из трёх структур типа uartx_t, по одной структуре для каждого канала данных.
- prio – Приоритет для задачи обработки прерываний. Должен быть выше, чем у других задач, обращающихся к данному драйверу.
- khz – Базовая частота опорного генератора. Для проекта “Олимп” можно указывать частоту процессора KHZ.
- baud – Требуемая скорость данных, бит/сек.

Пример:

```
…
#include <elvees/uartx.h>

uartx_t uartx [3];

void uos_init (void)
{
    …
    uartx_init (uartx, 50, KHZ, 9600);
}
```

Для передачи и приёма данных следует использовать стандартные функции интерфейса stream_t:

- putchar (поток, символ) – посылка одного байта
- getchar (поток) – ожидание и приём одного байта
- peekchar (поток) – приём одного байта без ожидания
- fflush (поток) – ожидание выдачи всех буферизованных символов
- puts (поток, строка) – посылка строки
- gets (поток, буфер, размер) – приём строки
- printf (поток, формат, …) - форматный вывод
- vprintf (поток, формат, аргументы) – форматный вывод stdarg

Пример:

```
puts (&uartx[0], "Hello, World!\r\n");
printf (&uartx[1], "UART1 frame errors = %u\r\n", uartx[1].frame_errors);
```

21.2. Драйвер Ethernet для микроконтроллера NVCom-01

Драйвер служит для работа со встроенным контроллером Ethernet микропроцессора NVCom-01.

Функции драйвера:

```
void eth_init (eth_t *u,
    const char *name, int prio,
    mem_pool_t *pool, arp_t *arp,
    const unsigned char *macaddr);
```

Инициалтзация драйвера. Создаются две задачи с приоритетами prio и prio+1 для обработки прерываний по передаче п приёму. Параметр macaddr задаёт MAC-адрес адаптера, который

должен быть уникальным для каждого экземпляра устройства.

`void eth_start_negotiation (eth_t *u);`	Принудительный запуск сеанса “договаривания” с противоположной стороной о режимах скорости и дуплекса Ethernet.
`int eth_get_carrier (eth_t *u);`	Запрос наличия соединения. Возвращает ненулевое значение, если сетевой кабель подключен и функционирует.
`long eth_get_speed (eth_t *u, int *duplex);`	Запрос информации о соединении. Возвращает скорость в битах в секунду, а также режим дуплекса. Если соединение отсутствует, возвращает 0.
`void eth_set_loop (eth_t *u, int on);`	Управление внутренним шлейфом. Если шлейф включён, передаваемые пакеты возвращаются обратно в приемник контроллера. Используется для самотестирования.
`void eth_set_promisc (eth_t *u, int station, int group);`	Управление фильтрацией принимаемых пакетов. Если station==1, контроллер будет принимать все unicast-пакеты независимо от адреса назначения. Если group==1, контроллер будет принимать все multicast-пакеты.
`void eth_poll (eth_t *u);`	Опрос состояния прерываний. Если по какой-то причине возможна потеря прерываний, необхолимо обеспечить периодический вызов этой функции, например из низкоприоритетной фоновой задачи.
`void eth_debug (eth_t *u, stream_t *stream);`	Выдача отладочной информации о состоянии контроллера.

Пример инициализации драйвера Ethernet:

```
…
#include <elvees/eth.h>
…
eth_t eth;

void uos_init (void)
{
    …
    const unsigned char mac_addr [6] = { 0, 9, 0x94, 0xf1, 0xf2, 0xf3 };
    eth_init (&eth, "eth0", 70, &pool, arp, mac_addr);
    …
}
```

21.3. Драйвер LPORT для микроконтроллера MC-24

Драйвер выполняет обмен информацией через LPORT микропроцессора MC-24.

Функции драйвера:

`void lport_init (lport_t *l, mem_pool_t *pool, unsigned long port, unsigned long mode,`	Инициализация драйвера. Создаётся задача для обработки прерываний от L-

`unsigned long dma, unsigned long clk, unsigned long size, unsigned long prio);`	порта.
`void lport_set_port (lport_t *l, unsigned long port);`	Установка номера порта.
`void lport_set_mode (lport_t *l, unsigned long mode);`	Установка направления приём-передача.
`void lport_set_dma (lport_t *l, unsigned long dma);`	Установка режима DMA.
`void lport_set_clk (lport_t *l, unsigned long clk);`	Установка скорости.
`void lport_set_size (lport_t *l, unsigned long size);`	Установка размера пакета.
`void lport_reset (lport_t *l);`	Сброс порта в начальное состояние.
`void lport_send_data (lport_t *l, unsigned long *data, unsigned long size);`	Отправка данных.
`int lport_recv_data (lport_t *l, unsigned long *data, unsigned long size);`	Приём данных.
`void lport_kill (lport_t *l);`	Прекращение работы драйвера.

21.4. Обслуживание микросхемы MCB-01

Драйвер предназначен для работы с микросхемой MCB-01.

Чтение/запись памяти и регистров блока MBA осуществляется с помощью обычного обращения по адресу. Чтение/запись регистров SWIC, PMSC и адресного окна PCI должна осуществляться только с помощью функций mcb_read_reg() и mcb_write_reg(). Для разъяснений см. Руководство пользователя на микросхему 1892ХД1Я, п. 5.3.

`unsigned mcb_read_reg (unsigned addr);`	Чтение регистра блока SWIC, блока PMSC или адресного окна PCI.
`void mcb_write_reg (unsigned addr, unsigned value);`	Запись регистра блока SWIC, блока PMSC или адресного окна PCI.

Пример обращения к регистрам MCB-01:

```
…
#include <elvees/pci.h>
#include <elvees/mcb-01.h>
…

void user_proc()
{
    …
    mcb_write_reg (MCB_PCI_STATUS_COMMAND,
                   MCB_PCI_COMMAND_MEMORY | MCB_PCI_COMMAND_MASTER);
    …
    unsigned csr_master = mcb_read_reg (MCB_PCI_CSR_MASTER);
    …
}
```

Реализован набор функций для работы шины PCI в режиме «мастер»:

```
void pci_init (void);
```

Инициализация контроллера MCB-01 в режиме master.

```
int pci_cfg_read (unsigned dev, unsigned function, unsigned reg,
        unsigned *result);
```

Чтение конфигурационых регистров PCI-устройства. Возвращает 0 в случае фатальной ошибки. Результат помещается по адресу result. Параметры:

- dev - номер устройства на шине PCI (от 0 до 20)
- function - номер функции внутри устройства (от 0 до 7)
- reg - номер конфигурационного регистра (от 0 до 63)

```
int pci_cfg_write (unsigned dev, unsigned function, unsigned reg,
        unsigned value);
```

Запись конфигурационых регистров PCI-устройства. Возвращает 0 в случае фатальной ошибки. Параметры:

- dev - номер устройства на шине PCI (от 0 до 20)
- function - номер функции внутри устройства (от 0 до 7)
- reg - номер конфигурационного регистра (от 0 до 63)

```
int pci_io_read (unsigned addr, unsigned *result);
```

Чтение 32-битного слова из i/o-пространства PCI-устройства. Возвращает 0 в случае фатальной ошибки. Результат помещается по адресу result.

```
int pci_io_write (unsigned addr, unsigned value);
```

Запись 32-битного слова в i/o-пространство PCI-устройства. Возвращает 0 в случае фатальной ошибки.

```
int pci_mem_read (unsigned addr, unsigned *data, unsigned nwords);
```

Чтение массива 32-битных слов из памяти PCI-устройства. Возвращает 0 в случае фатальной ошибки.

```
int pci_mem_write (unsigned addr, unsigned *data, unsigned nwords);
```

Запись массива 32-битных слов в память PCI-устройства. Возвращает 0 в случае фатальной ошибки.