

NedoPC-ARMOS. Описание возможностей. Примеры.

26 сентября 2007 г.

Аннотация

Вся приведённая информация относится к NedoPC-ARMOS версии 0.11.0 и будет меняться в других версиях.

1 Введение.

NedoPC-ARMOS предназначена для работы с процессорами типа AT91SAM7xxx. Но может быть без особого труда перенесена и на другие типы процессоров с ядром ARM7thumb.

Поскольку NedoPC-ARMOS не достигла ещё первой версии (от роду ей всего месяц с небольшим) - то возможности её весьма скромные, но уже позволяют реализовать весьма интересные проекты.

NedoPC-ARMOS состоит из трёх основных компонентов - уровня абстракции оборудования, что сосредоточен в каталоге *hard*, системных модулей и библиотек, что расположены в каталоге *sysmods* и дополнительных модулей, что находятся в каталоге *modules*.

Рассмотрим кратко назначение содержимого остальных каталогов:

doc - тут всё ясно. там лежит документация, её читать нужно.

include - заголовочные файлы NedoPC-ARMOS и стандартной библиотек *libc* и *libm* (тех самых, которые содержат функции работы со строками и синусы-косинусы разные).

incmods - заголовочные файлы дополнительных модулей. При сборке Makefile дополнительных модулей сами копируют необходимые пользовательским программам *.h файлы в этот каталог.

lib и *obj* - содержат, полученные при сборке, библиотеки и объектные файлы NedoPC-ARMOS.

scripts - содержит основные скрипты, которые описывают как собирать систему, флаги компиляции и тому подобное.

Возможности NedoPC-ARMOS:

- Запускать несколько потоков команд, выполняющихся независимо.

- Наличие абстракции драйвера и файла (правда файлы пока в очень зачаточном виде, но команду `open("/dev/usart1", O_RDWR)` - выполняет правильно).
- Полное отделение аппаратно-зависимой части системы от аппаратно-независимой, что даёт возможность без особых телодвижений перенести систему на другой тип процессора.

Будем считать, что вы распаковали архив в каталог `/home/user/armos`. Сама NedoPC-ARMOS находится в каталоге `/home/user/armos`. Остальные каталоги содержат тестовые примеры. Конфигурация "по умолчанию" (файл `armos.dsc`) соответствует процессору AT91SAM7S256 с кварцем 18.432 МГц, к ногам PA0, PA1, PA2 - припаяны светодиоды.

2 Как собрать средства разработки.

Прежде, чем компилировать NedoPC-ARMOS, необходимо вооружиться кросс-средствами. Почему-то сборка кросс-компиляторов считается трудной. Хотя, на самом деле - это ничуть не сложнее, чем собрать любую другую программу. Как всегда в Linux это делается с помощью трёх всемогущих команд "configure", "make" и "make install". В принципе - вся трудность сводится к тому, что параметров у этих команд можно указывать до нескольких десятков и людей непривычных это отпугивает (а люди привычные - наслаждаются множеством возможностей, предоставляемых им, но практически никогда не используемых).

В нашем случае кросс-средства - это arm-elf-binutils и arm-elf-gcc. Всё ниже сказанное относится к работе в Linux. На компьютере должен быть установлен GNU C и make. Если их нет, то поинтересуйтесь как их установить в вашем дистрибутиве Linux.

Найдите и скачайте в интернете пакеты `binutils-2.16.tar.bz2` и `gcc-3.4.1.tar.bz2`. Это архивы с исходными текстами кросс-средств.

Устанавливать кросс-средства будем в каталог `/home/user/arm-cross-tools`.

Итак, начали:

Устанавливаем путь к кросс-средствам:

```
# export PATH=$PATH:/home/user/arm-cross-tools/bin
```

Если вы пользуетесь командным интерпретатором bash, то советую добавить эту строчку в конец файла `.bashrc`, который находится в вашем домашнем каталоге (для нашего примера это будет файл `/home/user/.bashrc`). Это позволит вам не набирать её каждый раз вручную.

Создаём каталоги для инсталляции.

```
# mkdir /home/user/arm-cross-tools
```

Создаём каталоги для сборки утилит *binutils*, которые включают в себя ассемблер, линкер, дизассемблер, а также ещё несколько утилит, предназначенных для просмотра, преобразования файлов из одного формата в другой и других полезных действий.

```
# mkdir /home/user/build
# mkdir /home/user/build/binutils
```

Скопируйте файл *binutils-2.16.tar.bz2* в каталог */home/user/build/binutils*. Далее переходим в этот каталог.

```
# cd /home/user/build/binutils
# tar -xvf binutils-2.16.tar.bz2
```

После распаковки в каталоге */home/user/build/binutils* появится подкаталог *binutils-2.16*. Переименуйте его в *src*:

```
# mv binutils-2.16 src
```

И удалите архив:

```
# rm -f binutils-2.16.tar.bz2
```

Далее, надо объяснить исходным текстам, как мы их хотим скомпилировать:

```
# src/configure --prefix=/home/user/arm-cross-tools --target=arm-elf
```

Ключ *--prefix=* говорит конфигуратору, куда необходимо установить собранные программы, а ключ *--target=* целевую платформу, в нашем случае - это *arm-elf*.

После завершения работы конфигулятора, если не возникло никаких ошибок (а они могут возникнуть, если в вашей системе не хватает чего-нибудь), можно компилировать утилиты:

```
# make
```

и, затем, установить их:

```
# make install
```

Теперь, чтобы удостовериться, что утилиты установлены и пути прописаны дайте команду:

```
# arm-elf-as --version
```

В ответ на которую должно выдаться сообщение, вроде такого:

```
GNU assembler 2.16
Copyright 2005 Free Software Foundation, Inc.
This program is free software; you may redistribute it under the terms of
the GNU General Public License.  This program has absolutely no warranty.
This assembler was configured for a target of 'arm-elf'.
```

Ну вот! Ассемблер есть! Осталось поставить *C*.

Продолжим. Всё почти так же как и в случае с *binutils*.

Создаём каталог для сборки C:

```
# mkdir /home/user/build/gcc.
```

Скопируйте файл *gcc-3.4.1.tar.bz2* в каталог */home/user/build/gcc*.

Далее переходим в этот каталог.

```
# cd /home/user/build/gcc
# tar -xvf gcc-3.4.1.tar.bz2
```

После распаковки в каталоге */home/user/gcc* появится подкаталог *gcc-3.4.1*. Переименуйте его в *src*:

```
# mv gcc-3.4.1 src.
```

И удалите архив:

```
# rm -f gcc-3.4.1.tar.bz2
```

Далее, надо объяснить исходным текстам, как мы их хотим скомпилировать:

```
# src/configure --prefix=/home/user/arm-cross-tools --target=arm-elf
--enable-languages=c
```

Ключ *-enable-languages=* указывает, поддержку каких языков коллекции компиляторов мы хотим обеспечить. В нашем случае - это только *C*.

После завершения работы конфигулятора, если не возникло никаких ошибок, можно компилировать:

```
# make LANGUAGES='c'
```

и, затем, установить язык *C*:

```
# make LANGUAGES='c' install
```

Теперь, чтобы удостовериться, что утилиты установлены и пути прописаны дайте команду:

```
# arm-elf-gcc --version
```

В ответ на которую должно выдаться сообщение о версии GNU C, например:

```
arm-elf-gcc (GCC) 3.4.1
Copyright (C) 2004 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Вот собственно и всё! Этого достаточно, чтобы писать программы, используя NedoPC-ARMOS.

Всё, что было описано в 99% случаев будет прекрасно работать и с другими (не более старыми!) версиями пакетов *binutils* и *gcc*. Так что, если не удалось скачать те версии, что указаны здесь - не расстраивайтесь, скачайте другие версии и проделайте с ними все описанные действия.

Если что-то идёт не так - то причин может быть (практически, теоретически можно выдумать хоть десяток) всего две - или у вас в системе не установлен какой-то компонент или не прописан какой-то путь.

3 Скрипты линкера или о страшных lds файлах.

Люди боятся скриптов линкера. Они вызывают в людях панический страх, сравнимый разве что с походом к зубному врачу 20 лет назад. Кто ходил, тот знает.

А между тем, скрипт линкера (который часто имеет расширение lds от linker-debugger-script) - это всего лишь текстовый файл, в котором описано как привязать к физическим адресам откомпилированную программу.

Для того, что бы понять зачем нужен скрипт линкера и как он работает, рассмотрим как происходит компиляция программ.

В общем случае сборка программы идёт в ТРИ этапа.

Этап 1. Препроцессор. На этом этапе происходит преобразование исходного кода в соответствии с директивами препроцессора. Директивы препроцессора, это то, что в исходном коде начинается с символа *#*:

```
#define VAR 123
#undef VAR1
#if VAR_A
#ifdef VAR_B
#ifdef VAR_C
#else
#endif
#endif
#endif
```

Все подобные конструкции обрабатываются препроцессором. Более на этом останавливаться не будем. Кому интересны подробности - пусть наберёт волшебную команду *info gcc*.

Этап 2. С выхода препроцессора обработанный файл поступает на вход компилятора. Компилятор формирует из исходного кода - объектный.

Что такое объектный код? Это код, не привязанный к физическим адресам и содержащий перекрёстные ссылки на процедуры и переменные, содержащиеся в других объектных файлах. Обычно файлы, скомпилированных в объектный код программ, имеют расширение **.o**.

Этап 3. Линкер. Последний этап сборки программы. На этом этапе из объектного кода (одного или нескольких файлов ***.o**) собирается исполняемый файл. То есть расставляются физические адреса, перекрёстные ссылки заменяются на фактические значения. И на выходе получается исполняемый файл формата **elf**. Не пугайтесь, вы не ограничены этим форматом! С помощью утилиты **arm-elf-objcopy**, входящей в состав **binutils** этот файл можно преобразовать в любой другой формат - **.bin**, **.hex**, и т.д.

Для того, чтобы линкер привязал программу к необходимым нам адресам памяти, и нужен скрипт линкера. Но прежде, чем заняться его написанием, рассмотрим несколько положений:

3.1 Секции программы.

Исторически сложилось так, что всё, что находится в программе условно делили на три части - код программы, данные и стек. Конечно, это не описывает всех возможных ситуаций, но в большинстве случаев - годится. На сегодняшний день, компилятор GCC имеет возможность задать сколько угодно секций, назвать их по своему и привязать к любым физическим адресам. Но три основные секции - код (секция **.text**), инициализированные переменные (секция **.data**) и неинициализированные переменные (секция **.bss**) присутствуют всегда. Сразу отмечу, что хотя секция **.bss** и называется секцией стека, но в неё входит не только стек но и все неинициализированные переменные. Кроме того, эта секция (по стандарту) должна быть заполнена нулём перед запуском программы. Физически переменные, расположенные в секции **.bss** ни в каком файле не сохраняются.

Когда компилятор создаёт переменную или компилирует процедуру, то он ВСЕГДА помечает (даже если пользователь не указал этого явно) - в какой секции она находится. Итак, если не задано иное:

- Процедуры располагаются в секции **.text**.
- Инициализированные статические (то есть глобальные или с модификатором **static**) переменные (те, что инициализируются ДО исполнения программы, например так **int a=5;** (a - глобальная переменная) или **static int a=5;**) располагаются в секции **.data**.
- Неинициализированные статические (то есть глобальные или с модификатором **static**) переменные (те, что не инициализируются ДО исполнения программы, и определяются например так **int a;** (a - глобальная переменная) или **static int a;**) располагаются в секции **.bss**.
- Статические константы (определённые, например, **как const int f=8;**) располагаются в секции **.rodata**. Секция **.rodata** является частью секции **.data**, если не указано иное в скрипте линкера.

3.2 Простейший скрипт линкера.

Итак, посмотрим на простейший скрипт линкера, описывающий распределение памяти в процессоре at91sam7s256 :

```
/* Области памяти процессора at91sam7s256. */
MEMORY
{
    flash    : ORIGIN = 0x100000, LENGTH = 256K      /* FLASH EPROM */
    ram      : ORIGIN = 0x200000, LENGTH = 64K        /* static RAM area */
}
/* Секции. */
SECTIONS
{
    /* Начало памяти */
    . = 0;
    /* Секция кода программы и констант */
    .text :
    {
        *(.text) /* Код программы */
        *(.rodata) /* Константы */
    } >flash
    /* Секция инициализированных данных. При компиляции располагается
по адресу _etext (сразу после секции кода). При запуске программы копируется
в память */
    .data :
    {
        *(.data) /* */
    } >ram
    /* Секция неинициализированных данных.
Перед выполнением программы заполняется нулями */
    .bss :
    {
        *(.bss)
    } >ram
}
```

Каждый раздел файла начинается с названия, определяющего его содержимое. Описание раздела заключается в фигурные скобки.

Сначала (в разделе MEMORY) мы описываем расположение и размеры областей памяти в формате:

<имя области памяти> : ORIGIN = <начальный адрес>, LENGTH = <длина>.

Имя области памяти - какое придумаете (желательно понятное вам). Начальный адрес - и так понятно. Длина - тоже понятно, но есть нюанс - длину можно задавать как в байтах, так и в килобайтах и в мегабайтах. Например:

LENGTH = 256 - длина 256 байт.
LENGTH = 32K - длина 32 Кбайт.
LENGTH = 2M - длина 2 Мбайт.

В процессоре at91sam7s256 две области памяти - флеш-ПЗУ и ОЗУ. Вот они то и определены в секции MEMORY. Если раздел MEMORY отсутствует в скрипте линкера, то линкер не сможет отследить переполнение памяти (например, если вы задали слишком много переменных, огромный массив или написали слишком длинную программу). Поэтому всегда создавайте этот раздел в скрипте - меньше головной боли наживёте.

За разделом MEMORY следует раздел SECTIONS, в котором собственно и описаны все секции программы и их привязка к физическим адресам.

Переменная . (точка) - это текущий адрес линковки. Он может ТОЛЬКО УВЕЛИЧИВАТЬСЯ! В начале раздела SECTIONS этот адрес установлен директивой . = 0;

Далее следует описание секций кода *.text*, *.data* и *.bss*.

Символ * означает “всё-что-угодно”. То есть дериктива **(.text)* означает “всё-что-угодно” с пометкой *.text*. В нашем случае, в область флеш-ПЗУ входит не только текст программы, но и статические константы **(.rodata*)*.

Все секции и подсекции располагаются ТОЧНО в том порядке, как они указаны в скрипте линкера!

В конце секции *.text* стоит *> flash*. Это означает “содержимое секции привязать к адресам области памяти, описанной под именем *flash*”. Поскольку мы ещё ничего не писали в область памяти *flash*, то начальный адрес секции *.text* совпадёт с начальным адресом области памяти *flash* (в нашем случае 0x100000).

Далее подобным же образом описаны секции *.data* и *.bss*. Обратите внимание, что обе эти секции привязаны к одной области памяти (*>ram*). С секцией дата - всё ясно. Её начало будет совпадать с началом области памяти *ram*. А вот начало секции *.bss* будет смещено (поскольку часть области памяти *ram* уже занята секцией *.data*). То есть:

*Начало Секции .bss = Начало Области Памяти .ram +
Размер Секции .data;*

3.3 Определение собственных секций.

Когда линкер определяет адреса для процедур, то он гарантирует, что все процедуры будут расположены в секции *.text*, но никак не определяет порядок их расположения внутри секции. Обычно нам не важно к каким физическим адресам какая процедура или переменная привязана, но в некоторых случаях такая строгая привязка необходима. Например, это важно при начальной загрузке процессора. В процессорах с ядром ARM таблица векторов прерываний должна быть расположена с адреса 0 и никак иначе. Именно с 0го адреса начинается выполнение программы. Самый простой способ добиться строгой привязки расположения процедуры или переменных по конкретным адресам - это определить собственную секцию. Сделать

это не просто, а очень просто, например так (это секция *.text* из предыдущего примера, с определённой в ней подсекцией *.init*):

```
/* Секция кода программы и констант */
.text :
{
    *(.init)          /* Вектора прерываний */
    *(.text) /* Код программы */
    *(.rodata) /* Константы */
} >flash
```

Всё! Теперь любая переменная или процедура, определённая в секции *.init* будет располагаться с начала флеш-памяти.

В языке ассемблер секция указывается так:

```
.section .init
/*-----*/
/* Таблица векторов прерываний. Расположена с адреса 0x100000. Но при включении
процессора в адреса 0x100000 и 0x000000 включено флеш-ПЗУ. */
/*-----*/
vectable:
/* Таблица векторов прерываний */
        ldr    pc, [pc, #24]
        ldr    pc, [pc, #24]
        ldr    pc, [pc, #24]
        ldr    pc, [pc, #24]
        ldr    pc, [pc, #24]
        nop
        ldr    pc, [pc, #24]
        ldr    pc, [pc, #24]
        /* Таблица адресов переходов */
        .word   reset_proc
        .word   0
        .word   0
        .word   0
        .word   0
        nop
        .word   irq_proc
        .word   irq_proc
```

Всё, что описано после директивы ассемблера *.section <имя секции>* будет линковаться в секцию с этим именем пока не встретится другая директива *.section*. На код после *vectable* внимания не обращаем - он зависит от типа процессора.

На языке C секция принудительно указывается так - после имени переменной или процедуры добавляется:

```
__attribute__ ((section (".init")));
```

Разумеется, вместо *.init* должно быть указано имя той секции, которая нужна.

Неинициализированная переменная в секции *.init*:

```
int v __attribute__ ((section (".init")));
```

Инициализированная переменная в секции *.init*:

```
extern int v __attribute__ ((section (".init")));  
int v=8;
```

Процедура в секции *.init*:

```
void procname() __attribute__ ((section (".init")));  
void procname()  
{  
    /*Тело процедуры*/  
}
```

4 Как собрать NedoPC-ARMOS (очень кратко).

Для сборки NedoPC-ARMOS вам требуются следующие программы:

arm-elf-binutils (у меня 2.16), arm-elf-gcc (у меня 3.4.1) и make (у меня 3.81).

Если у вас нет таких программ, то установите их. Обязательно укажите переменной среды *PATH* пути к этим программам.

Зайдите в каталог */home/user/armos/armos* и наберите команду *make > log*. Если всё сделано правильно, то компьютер немного подумает и вернётся в командный интерпретатор ничего не сказав. Если же были какие-то ошибки, то о них будет сообщено на экране.

Если всё указанные манипуляции прошли успешно, то в каталогах */home/user/armos/armos/lib* и */home/user/armos/armos/obj* должны появиться соответственно *.a и *.o файлы.

Это и есть NedoPC-ARMOS. Осталось только написать программу пользователя, скомпилировать её и загрузить в память микроконтроллера.

Зайдите, например, в каталог */home/user/armos/armos-printf-test/*, наберите make и получите на выходе файл *main.bin*.

Загрузите его в микроконтроллер (например с помощью программки *samba*) - и всё! Светодиоды замигают, со всех USART посыпятся сообщения (на скорости 115200).

5 Makefile программы пользователя.

Рассмотрим подробно как компилируется программа пользователя. Для этого заглянем в Makefile, что расположен в каталоге `/home/user/armos/armos-printf-test/` :

```
#-----
# Тут указан путь к корневому каталогу NedoPC-ARMOS
# (можно указать, например /home/user/armos/armos)
SYSDIR=../armos
#-----
# main - Имя приложения.
APP= main
#-----
# Включаем скрипт, который автоматически устанавливает все
# переменные среды - CC, OBJCOPY, пути к библиотекам и так
# далее. Это очень-очень важная строчка!
include $(SYSDIR)/scripts/common.mk
#-----
# Основная цель.
all:
    # Компиляция системы (не нужна, если система уже скомпилирована)
    (1)    make -C $(SYSDIR)
    # Сборка программы пользователя.
    $(CC) *.c $(LD) *.o -o$(APP) $(LIBS)
    # Преобразование в *.bin файл
    $(OBJCOPY) $(APP) -O binary $(APP).bin
    # Преобразование в *.hex файл (не нужно, если программатор понимает *.bin)
    $(OBJCOPY) -I binary -O ihex $(APP).bin $(APP).hex
    # Полный дизассемблер программы (не обязателен,
    # если не хотите полюбоваться на то, что сотворил компилятор)
    $(OBJDUMP) -D -z $(APP) > $(APP).dmp $(OBJDUMP) -x -z $(APP) > $(APP).map
#-----
clean:
    # Очистка системных каталогов. (не нужна, если система не была переконфигурирована)
    (2)    make -C $(SYSDIR) clean
    # Очистка пользовательской программы
    rm -rf *.o $(APP) *.hex *.bin *.dmp *.map
#-----
burn: samba -f cmd
#-----
```

В принципе всё понятно и из комментариев. Сборка происходит следующим образом - компилируются все файлы `*.c`, находящиеся в текущем каталоге (в данном случае - это один файл `main.c`), затем они линкуются с файлами, расположенными в каталоге `/home/user/armos/armos/obj` и библи-

лиотеками, расположенными в каталоге `/home/user/armos/armos/lib`. Получается файл формата *elf* (в данном случае - файл *main*), который преобразуется в бинарный “имидж” и *hex*-файл.

Если вы скомпилировали систему, как было сказано ранее, то строки, помеченные как (1) и (2) можно смело удалять - они вызывают компиляцию и очистку системы. Если вы удалите эти строки, то компиляция программы пользователя будет проходить гораздо быстрее,

но имейте ввиду, что если вы внесёте изменения в код самой NedoPC-ARMOS, то не забудьте её перекомпилировать, то есть зайти в каталог `/home/user/armos/armos/` и дать команду `make clean > log` и `make > log`.

6 Функции NedoPC-ARMOS для работы с задачами.

Рассмотрим теперь, как построена система “изнутри”, с точки зрения выполняющейся программы.

Каждая процедура, запущенная как отдельный поток, описывается структурой *stask* (см. файл `/home/user/armos/armos/include/task.h`). Максимально возможное количество потоков (задач) в системе определяется константой *MAXTASKS*. Задача имеет следующие ресурсы: стек (он описывается полями структуры *stask* - *sp*, *sp_bot* и *sp_size*) и файлы (они описываются полями структуры *stask* - *dfile* и *pfile*). Максимально возможное количество файлов, которое может открыть одна задача определяется константой *MAX_FILES*. Эта константа ВСЕГДА должна быть не меньше 3х, поскольку с каждым потоком всегда связано 3 файла - стандартный ввод, стандартный вывод и сообщения об ошибках.

Любая процедура, запущенная как поток имеет заголовок:

```
int <имя процедуры>(int argc, char** argv);
```

то есть полностью соответствует функции `main` в языке C.

Предположим, мы захотели помигать двумя светодиодами, каждым в своём потоке:

```
/* Не забываем включить заголовки */
#include <leds.h>
#include <task.h>
/* Процедура, мигающая светодиодом в отдельном потоке. */
int migalka_th1(int argc, char** argv)
{
    /* Бесконечный цикл */
    while(1)
    {
        leds_flash(1);    /* Мигаем */
    }
}
```

```

        task_sleep(1);    /* Спим секунду */
    }
}
/* Точка входа в программу пользователя */
int main(int argc, char** argv)
{
    /* Добавляем поток-задачу мигания первым светодиодом. */
    task_add(migalka_th1, 0, 0);
    /* Бесконечный цикл */
    while(1)
    {
        leds_flash(0);    /* Мигаем */
        task_sleep(1);    /* Спим секунду */
    }
}
/* */

```

Всё! Компилируйте, загружайте в контроллер - и два светодиода будут весело мигать, говоря о том, что многозадачность - не шутка.

Рассмотрим, что делают функции ОС в нашей программе:

```
int task_add(void *proc, int argc, char** argv);
```

Ставит задачу на выполнение. Передаёт ей в качестве параметров *argc* и *argv*. Возвращает идентификатор задачи (как принято его называть - *pid*), который всегда больше 0. Если же не удалось запустить задачу (например нет места в памяти) - то возвращается -1.

В противоположность функции *task_add()* имеется функция

```
int task_del(int pid);
```

которая удаляет задачу по её *pid*. Возвращает 0 при успешном выполнении или -1 при ошибке (например нет такой задачи).

Функции:

```

void task_sleep(int n);    /* n - время в секундах */
void task_msleep(int n);  /* n - время в миллисекундах */
void task_usleep(int n);  /* n - время в микросекундах */

```

Переводят задачу в состояние “сна”, когда на определённое время процесс перестаёт выполняться, занимать такты процессора. После истечения этого времени задача “просыпается” и продолжает выполняться. Учтите, что эти функции не гарантируют ТОЧНОГО по времени пробуждения. Если в качестве параметра *n* был задан 0, то задача засыпает “вечным сном”. То есть не проснётся до тех пор, пока её не разбудят функцией:

```
int task_wakeup(int pid);
```

которая принудительно пробуждает процесс с заданным *pid*. При успешном завершении функция *task_wakeup()* возвращает 0 и -1 при ошибке.

Есть ещё одна полезная функция управления задачами:

```
void task_next();
```

которая позволяет отдать оставшиеся кванты времени другим задачам. Например, если задача ждёт какое-то событие, то после проверки разумно принудительно перейти к выполнению другой задачи, а не болтаться в цикле ожидания. Для этих целей и предназначена эта функция.

7 Функции NedoPC-ARMOS для работы с файлами¹

Как и в любой операционной системе, имеются стандартные функции работы с файлами: *open()*, *close()*, *read()*, *write()*.

Рассмотрим их по отдельности:

```
int open(char *pathname, int flags);
```

открывает файл *pathname* с правами, установленными во *flags*. Если файл открыт успешно, то возвращается его *дескриптор* - неотрицательное число. Если произошла ошибка (нет такого файла или нет места в таблице открытых файлов) - то возвращается -1. Например, чтобы получить полный доступ (на чтение и запись) к USART2 следует написать строчку:

```
int fd=open("/dev/usart2", O_RDWR);
```

после этого можно пользоваться процедурами чтения и записи.

Параметр *flags* указывает в каком режиме следует открывать файл:

O_RDONLY - только чтение.

O_WRONLY - только запись.

O_RDWR - чтение и запись, то же самое, что и (*O_RDONLY* / *O_WRONLY*).

O_NONBLOCK - неблокирующий режим. По умолчанию, при операции чтения *read()*, происходит блокирование процесса, пока не появятся данные в файле. При установке флага *O_NONBLOCK* управление будет возвращаться процессу независимо, имеются ли данные для считывания или нет.

O_SYNC - синхронный режим. По умолчанию, при операции записи *write()*, данные сохраняются в буфере и управление возвращается процессу. При установке флага *O_SYNC* управление процессу не будет возвращено до тех пор, пока все данные не будут переданы по назначению.

Функция

¹На данный момент (версия 0.10.6.):

В качестве файлов видны только устройства, условно расположенные в каталоге "/dev". - "/dev/usart0", "/dev/usart1", "/dev/usart2".

```
int fcntl(int fd, int flag, ...);
```

позволяет изменить режим доступа к уже открытому файлу, дескриптор которого **fd**. Параметр **flag** задаёт действие, остальные параметры зависят от типа этого действия.

int fcntl(fd, F_GETFL); - возвращает режим доступа к файлу fd.

int fcntl(fd, F_SETFL, flags); - устанавливает режим доступа к файлу fd в соответствии с параметром flags. Флаги - те же, что и при вызове open(). Например, если нужно уже открытый файл перевести в неблокируемый режим, то это будет выглядеть так:

```
fcntl(fd, F_SETFL, fcntl(fd, F_GETFL) | O_NONBLOCK);
```

При ошибке функция fcntl() возвращает -1.

Функция

```
long write(int fd, void *buf, long count);
```

записывает в файл, дескриптор которого **fd**, **count** байт, которые расположены в памяти с адреса **buf**. При успешном завершении возвращает количество записанных байт. Их может быть меньше, чем **count**, или даже 0 (не записан ни один байт). Это не считается ошибкой. Если же произошла ошибка, то возвращается -1.

Функция

```
long read(int fd, void *buf, long count);
```

считывает из файла, дескриптор которого **fd**, **count** байт, и располагает их в памяти с адреса **buf**. При успешном завершении возвращает количество считанных байт. Их может быть меньше, чем **count**, или даже 0 (не считан ни один байт). Это не считается ошибкой. Если же произошла ошибка, то возвращается -1.

Функция

```
int close(int fd);
```

закрывает файл, дескриптор которого **fd**, освобождает все связанные с ним ресурсы. Если файл успешно закрыт, возвращается 0, если же произошла ошибка, то возвращается -1.

Для управления файлами-устройствами имеется функция:

```
int ioctl(int fd, int flags, ...);
```

которая позволяет задавать специфические данные, зависящие от типа устройства. Например, для задания скорости USART2 9600бит/сек, открытого ранее функцией open() надо написать следующую строку:

```
ioctl(fd, IOCTL_SETSPEED, 9600);
```

- 8 Организация драйверов NedoPC-ARMOS.
- 9 Начальная загрузка.
- 10 Макросы системной инициализации.