Оглавление

[ЛК 3 (Сентябрь) 2](#_Toc98659422)

[ЛК4 8](#_Toc98659423)

[ЛК5 (Октябрь) 24](#_Toc98659424)

[ЛК6 32](#_Toc98659425)

[ЛК7 38](#_Toc98659426)

[**Лк 8. Сервоприводы** 44](#_Toc98659427)

[**Лк 9. Устройство управления шаговым двигателем.** 46](#_Toc98659428)

[ESP8266 49](#_Toc98659429)

[Гироскопы МЭМС (Микроэлектромеханические системы) 52](#_Toc98659430)

[Виды датчиков 53](#_Toc98659431)

[Операционные системы реального времени (RTOS) 56](#_Toc98659432)

[Февраль, ЛК1. Понятие операционной системы 56](#_Toc98659433)

[ОС как расширенная виртуальная машина 56](#_Toc98659434)

[ОС как менеджер ресурсов 57](#_Toc98659435)

[Особенности ОС реального времени 60](#_Toc98659436)

[Области применения систем реального времени 64](#_Toc98659437)

[Аппаратурная среда систем реального времени 65](#_Toc98659438)

[Февраль, ЛК2. Стандарты ОСРВ 66](#_Toc98659439)

[Архитектура ОС 71](#_Toc98659440)

[Архитектура ОС реального времени 76](#_Toc98659441)

[Февраль, ЛК3. Операционные системы реального времени 77](#_Toc98659442)

[ОС реального времени OS-9 77](#_Toc98659443)

[ОС реального времени VxWorks 79](#_Toc98659444)

[ОС реального времени Wind River Linux 80](#_Toc98659445)

[ОС реального времени OSEK/VDX 80](#_Toc98659446)

[ОС реального времени LynxOS 82](#_Toc98659447)

[ОС реального времени Windows CE 83](#_Toc98659448)

[RTX (Real Time Extension) 84](#_Toc98659449)

[ОС реального времени RTEMS 84](#_Toc98659450)

[ОС реального времени FreeRTOS 86](#_Toc98659451)

[ОС реального времени QNX 87](#_Toc98659452)

[Февраль, ЛК4. Микроядро QNX Neutrino 89](#_Toc98659453)

[Потоки и процессы 93](#_Toc98659454)

[Март, ЛК1. Планирование потоков 101](#_Toc98659455)

[Службы синхронизации 107](#_Toc98659456)

# ЛК 3 (Сентябрь)

***Интерфейсы микроконтроллеров***

*Физический (аппаратный) интерфейс — способ взаимодействия физических устройств.*

При попытке связать микроконтроллер с другими устройствами требуется знать определенный набор правил, методов и характеристик оборудования.

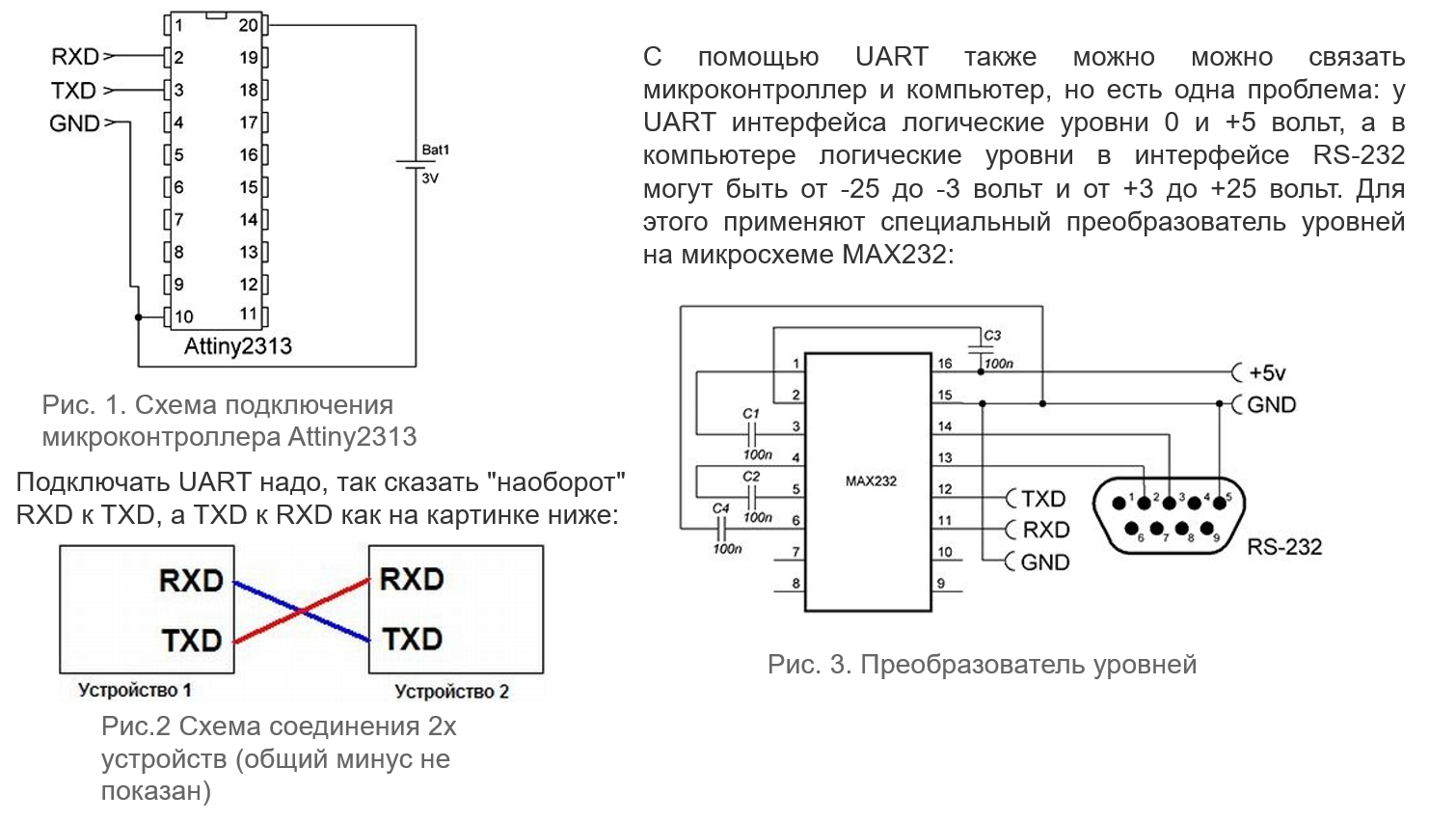
**Последовательный интерфейс UART/USART**

Универсальный асинхронный или универсальный синхронно/асинхронный приемопередатчик (Universal Synchronous/Asynchronous Receiver and Transmitter — **UART** или **USART**) — удобный и простой последовательный интерфейс для организации информационного канала обмена микроконтроллера с внешним миром. Способен работать в дуплексном режиме (одновременная передача и прием данных). Он поддерживает протокол стандарта RS-232, что обеспечивает возможность организации связи с персональным компьютером.

Изначально использовался в компьютерах для большинства периферийных устройств, таких как плоттер, удаленный принтер, мышь, внешний модем и т. д. До настоящего времени для последовательной связи IBM PC-совместимых компьютеров используются адаптеры с интерфейсом **RS-232С** (новое название EIA-232D). В современном IBM PC-совместимом компьютере может использоваться до четырех последовательных портов, имеющих логические имена соответственно COM1, COM2, COM3 и COM4. Основой последовательного адаптера является микросхема UART (Universal Asynchronous Receiver/Transmitter) — универсальный асинхронный приемопередатчик. Обычно используется микросхема UART 16550A. Она имеет 16-символьный буфер на прием и на передачу и, кроме того, может использовать несколько каналов прямого доступа в память DMA .

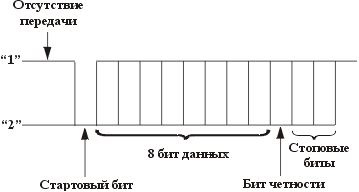
При передаче микросхема UART преобразует параллельный код в последовательный и передает его побитно в линию, обрамляя исходную последовательность битами старта, останова и контроля. При приеме данных UART преобразует последовательный код в параллельный (разумеется, опуская служебные символы). Непременным условием правильной передачи (приема) является одинаковая скорость работы приемного и передающего UART, что обеспечивается стабильной частотой кварцевого резонатора. Основным преимуществом последовательной передачи является возможность пересылки данных на большие расстояния, как правило, не менее 30 метров. В IBM PC-совместимых персональных компьютерах из 25 сигналов, предусмотренных стандартом RS-232, используются в соответствии с EIA только 9; таким образом, в данном интерфейсе как правило применяются 9-контактные разъемы типа DB-Shell.

В современных компьютерах UART и СОМ порт уже не применяются напрямую, но они получили вторую жизнь для связи с различными нестандартными внешними устройствами в числе которых вошли и устройства на микроконтроллерах. Аппаратная часть при этом стала значительно проще для связи микроконтроллеров друг с другом подключение UART осуществляется по трём линиям: RXD – приём, TXD – передача и GND – общий (минус).



Все сигналы UART передаются специально выбранными уровнями, обеспечивающими высокую помехоустойчивость связи. Отметим, что данные передаются в инверсном коде (логической единице соответствует низкий уровень, логическому нулю — высокий уровень.   
Формат передаваемых данных показан на рисунке 4. Собственно данные (5, 6, 7 или 8 бит) сопровождаются стартовым битом, битом четности и одним или двумя стоповыми битами.

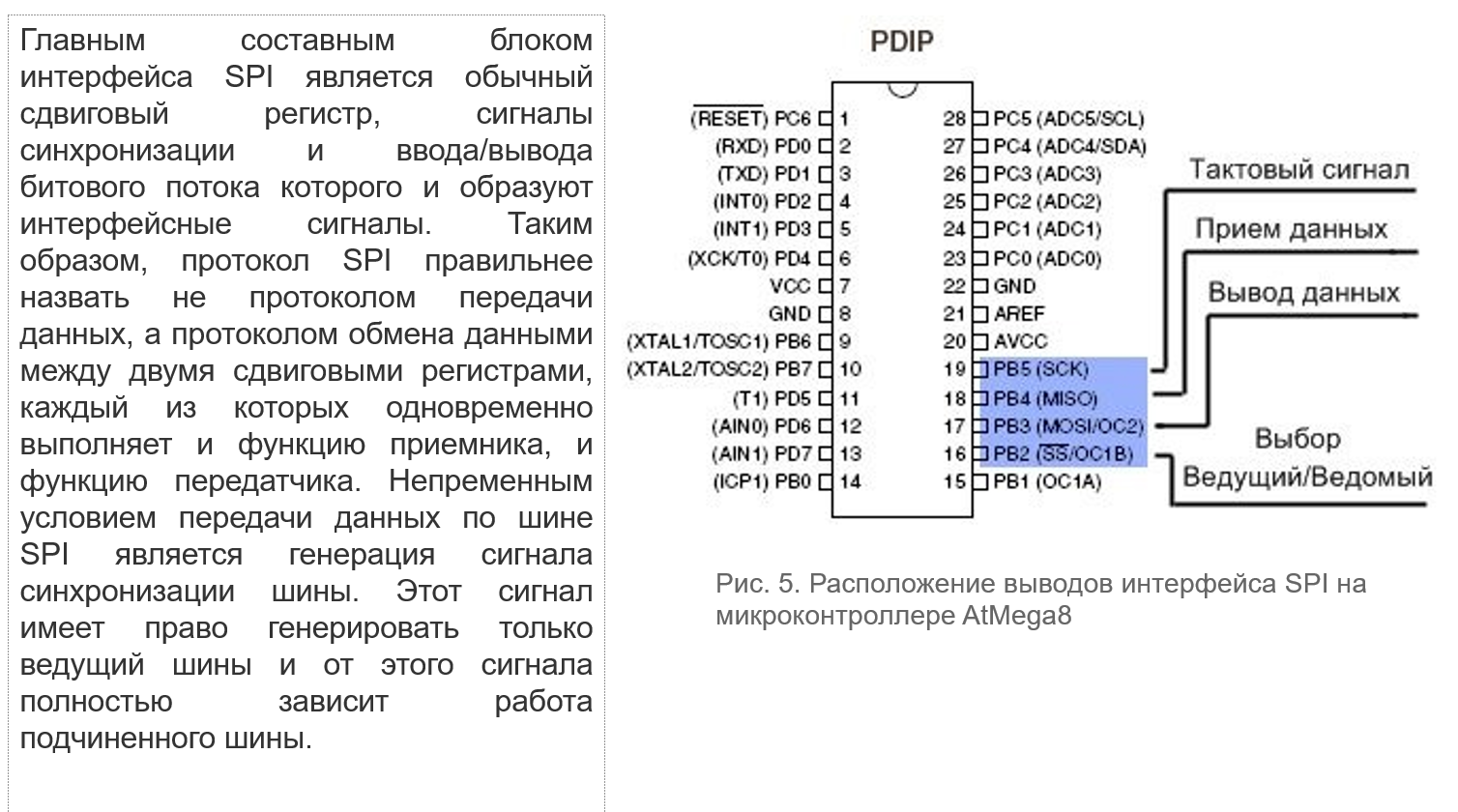
Получив стартовый бит, приемник выбирает из линии биты данных через определенные интервалы времени. Очень важно, чтобы тактовые частоты приемника и передатчика были одинаковыми, допустимое расхождение — не более 10%). Скорость передачи по RS-232C может выбираться из ряда: 110, 150, 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200 бит/с.



**Последовательный периферийный интерфейс SPI**

Последовательный периферийный трехпроводный интерфейс SPI (Serial Peripheral Interface) предназначен для организации обмена данными между двумя устройствами. С его помощью может осуществляться обмен данными между микроконтроллером и различными устройствами, такими, как цифровые потенциометры, ЦАП/АЦП, FLASH-ПЗУ и др. С помощью этого интерфейса удобно производить обмен данными между несколькими микроконтроллерами AVR. Кроме того, через интерфейс SPI может осуществляться программирование микроконтроллера.

Изначально он был придуман компанией Motorola, а в настоящее время используется в продукции многих производителей. Его наименование является аббревиатурой от 'Serial Peripheral Bus', что отражает его предназначение — шина для подключения внешних устройств. Шина SPI организована по принципу 'ведущий-подчиненный'. В качестве ведущего шины обычно выступает микроконтроллер, но им также может быть программируемая логика, DSP-контроллер или специализированная ИС. Подключенные к ведущему шины внешние устройства образуют подчиненных шины. В их роли выступают различного рода микросхемы, в т.ч. запоминающие устройства (EEPROM, Flash-память, SRAM), часы реального времени (RTC), АЦП/ЦАП, цифровые потенциометры, специализированные контроллеры и др.

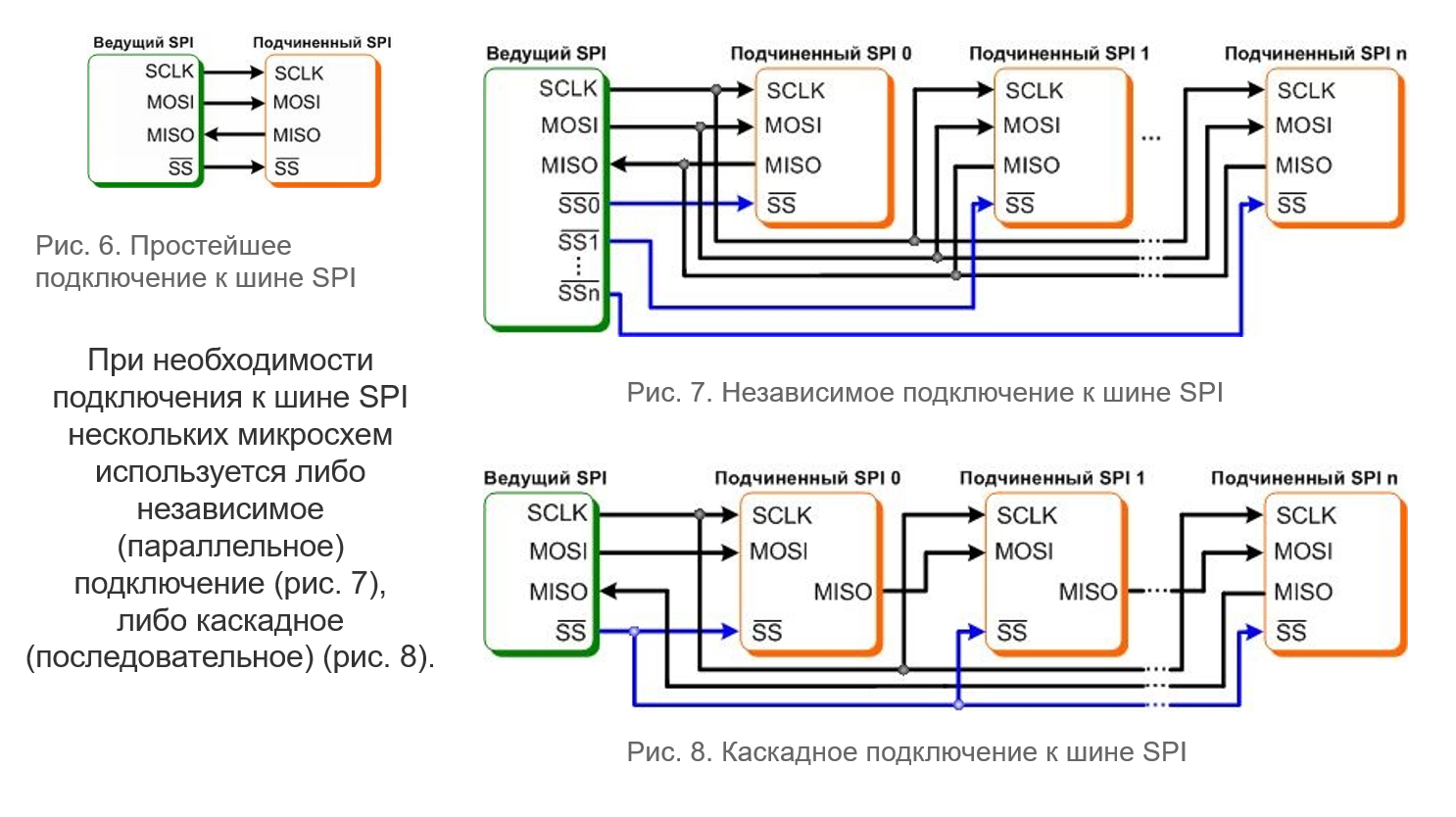


***Электрическое подключение***

Существует три типа подключения к шине SPI, в каждом из которых участвуют четыре сигнала.

Самое простое подключение, в котором участвуют только две микросхемы, показано на рисунке 6. Здесь, ведущий шины передает данные по линии MOSI синхронно со сгенерированным им же сигналом SCLK, а подчиненный захватывает переданные биты данных по определенным фронтам принятого сигнала синхронизации. Одновременно с этим подчиненный отправляет свою посылку данных. Представленную схему можно упростить исключением линии MISO, если используемая подчиненная ИС не предусматривает ответную передачу данных или в ней нет потребности. Одностороннюю передачу данных можно встретить у таких микросхем как ЦАП, цифровые потенциометры, программируемые усилители и драйверы. Таким образом, рассматриваемый вариант подключения подчиненной ИС требует 3 или 4 линии связи.

Чтобы подчиненная ИС принимала и передавала данные, помимо наличия сигнала синхронизации, необходимо также, чтобы линия SS была переведена в низкое состояние. В противном случае, подчиненная ИС будет неактивна. Когда используется только одна внешняя ИС, может возникнуть соблазн исключения и линии SS за счет жесткой установки низкого уровня на входе выбора подчиненной микросхемы. Такое решение крайне нежелательно и может привести к сбоям или вообще невозможности передачи данных, т.к. вход выбора микросхемы служит для перевода ИС в её исходное состояние и иногда инициирует вывод первого бита данных.



Независимое подключение более распространенное, т.к. достигается при использовании любых SPI-совместимых микросхем. Здесь, все сигналы, кроме выбора микросхем, соединены параллельно, а ведущий шины, переводом того или иного сигнала SS в низкое состояние, задает, с какой подчиненной ИС он будет обмениваться данными. Главным недостатком такого подключения является необходимость в дополнительных линиях для адресации подчиненных микросхем (общее число линий связи равно 3+n, где n-количество подчиненных микросхем). Каскадное включение избавлено от этого недостатка, т.к. здесь из нескольких микросхем образуется один большой сдвиговый регистр.

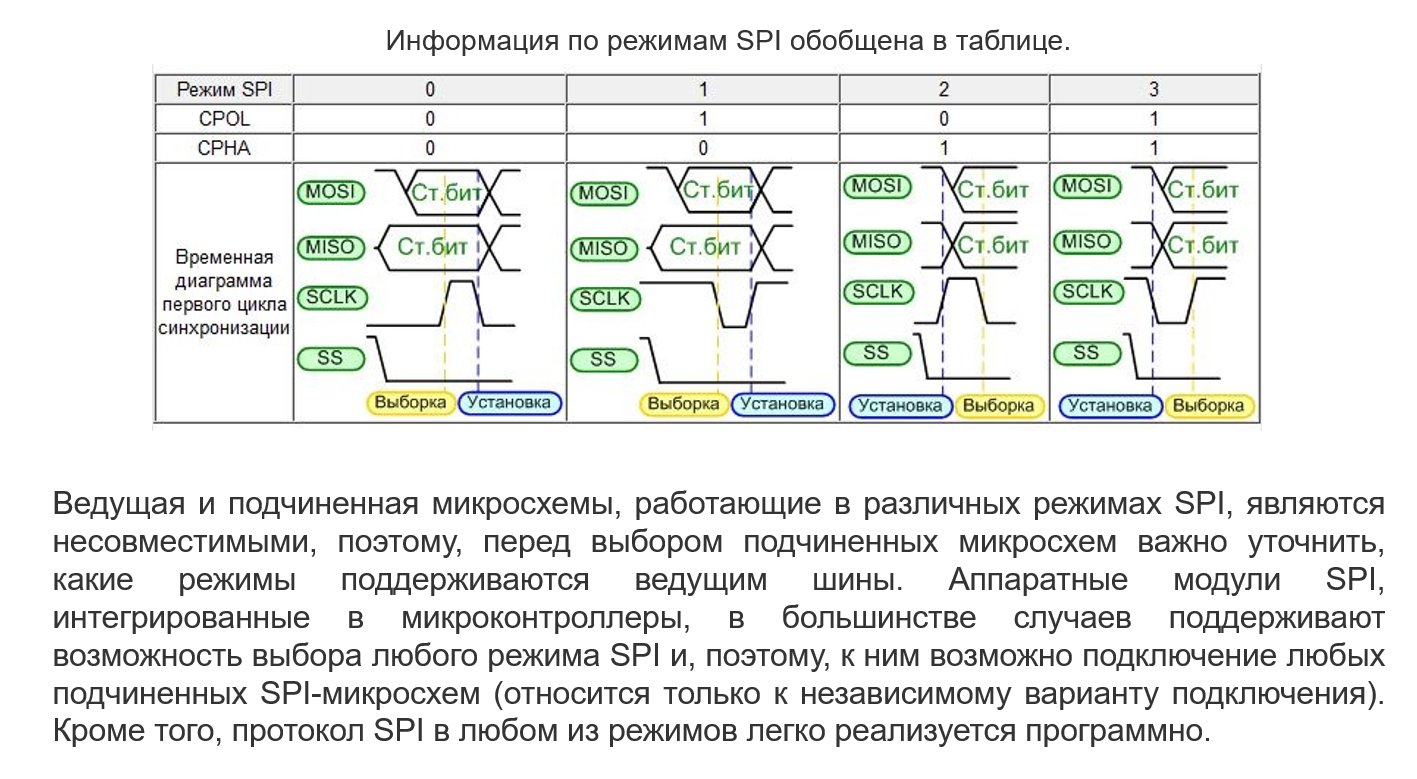
Для этого выход передачи данных одной ИС соединяется со входом приема данных другой, как показано на рисунке 8. Входы выбора микросхем здесь соединены параллельно и, таким образом, общее число линий связи сохранено равным 4. Однако использование каскадного подключения возможно только в том случае, если его поддержка указана в документации на используемые микросхемы. Чтобы выяснить это, важно знать, что такое подключение по-английски называется 'daisy-chaining'.

***Протокол передачи***

Протокол передачи по интерфейсу SPI предельно прост и, по сути, идентичен логике работы сдвигового регистра, которая заключается в выполнении операции сдвига и, соответственно, побитного ввода и вывода данных по определенным фронтам сигнала синхронизации. Установка данных при передаче и выборка при приеме всегда выполняются по противоположным фронтам синхронизации. Это необходимо для гарантирования выборки данных после надежного их установления. Если к этому учесть, что в качестве первого фронта в цикле передачи может выступать нарастающий или падающий фронт, то всего возможно четыре варианта логики работы интерфейса SPI. Эти варианты получили название режимов SPI и описываются двумя параметрами:

**CPOL** — исходный уровень сигнала синхронизации (если CPOL=0, то линия синхронизации до начала цикла передачи и после его окончания имеет низкий уровень (т.е. первый фронт нарастающий, а последний — падающий), иначе, если CPOL=1, — высокий (т.е. первый фронт падающий, а последний — нарастающий));

**CPHA** — фаза синхронизации; от этого параметра зависит, в какой последовательности выполняется установка и выборка данных (если CPHA=0, то по переднему фронту в цикле синхронизации будет выполняться выборка данных, а затем, по заднему фронту, — установка данных; если же CPHA=1, то установка данных будет выполняться по переднему фронту в цикле синхронизации, а выборка — по заднему).



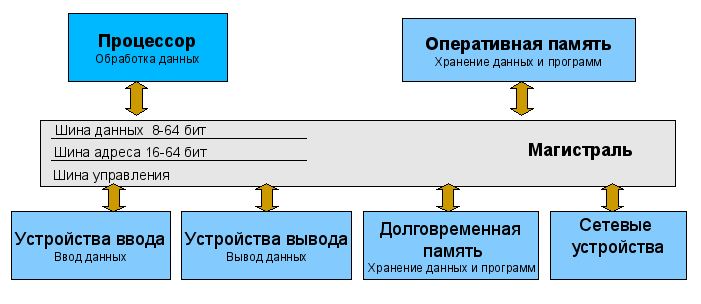
# ЛК4

**Микроконтроллеры**

**Классификация и структура микроконтроллеров**

**Основная особенность** современных микропроцессорных систем (МПС) состоит в завершении **переход**а **от систем, выполненных на основе нескольких больших интегральных схем (ИС)**, **к однокристальным системам**, которые объединяют в одном кристалле все основные элементы ЭВМ:

* + **центральный процессор** (ЦП),
  + **постоянное запоминающее устройство** (ПЗУ),
  + **оперативное запоминающее устройство** (ОЗУ),
  + **порты ввода/выводы** (УВВ),
  + устройства формирования временных интервалов - **таймеры** и т.д.



**Микроконтроллер** – это полноценный **компьютер на одной микросхеме**. Предназначен для управления различными электронными устройствами и осуществления взаимодействия между ними в соответствии с заложенной в микроконтроллер программой. В отличие от микропроцессоров, используемых в персональных компьютерах, микроконтроллеры содержат встроенные дополнительные устройства. Эти устройства выполняют свои задачи под управлением микропроцессорного ядра микроконтроллера.

**Основным классификационным признаком микроконтроллеров (МК)** является разрядность данных, обрабатываемых *арифметико-логическим устройством* (АЛУ).

По этому признаку они делятся на 4-, 8-, 16-, 32- и 64-разрядные.

Все типы **МК** можно условно разделить на три основных класса:

* + **8-разрядные *МК* для встраиваемых приложений;**
  + **16- и 32-разрядные управляющие *МК*;**
  + **цифровые сигнальные процессоры (DSP) для обработки данных.**
* **Наиболее распространенным** представителем семейства ***МК*** являются **8-разрядные приборы**. потому что основная область их применения находится **в системах управления реальными объектами**, где применяются, в основном, **алгоритмы с преобладанием логических операций**, скорость обработки которых практически не зависит от разрядности процессора.

**Основные фирмы производители**

1. **Intel** – i8048, **i8051** (CISC архитектура)
2. **Atmel** – серия **AT89** (CISC архитектура),

серия **AVR** AT90xx, Atmega, ATxmega (RISC архитектура)

1. **Microchip** – серия **PIC** (PIC12xx, PIC16xx и т.д.) (RISC архитектура)
2. **Motorola** – серия **М68** (М68НС05, НС08, НС11) (CISC архитектура)
3. **Texas Instruments** – серия **TMS320** – DSP (RISC архитектура)
4. **STMicroelectronics** – серия STM8, STM32 (RISC архитектура)

**Тактовая частота**, определяет, сколько вычислений может быть выполнено за единицу времени. В основном **производительность микроконтроллера и потребляемая им мощность увеличиваются с повышением тактовой частоты.**

**Производительность микроконтроллера** измеряют в **MIPS** (Million Instruсtions per Second – миллион инструкций в секунду)

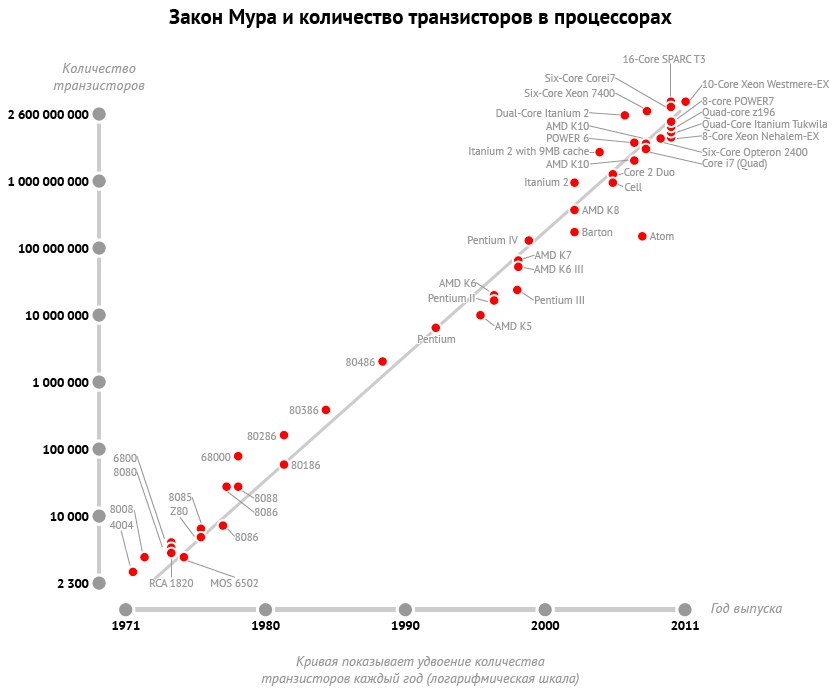
**Первый микроконтроллер**

Первый микроконтроллер появился на свет в 1976 году, через 5 лет после создания первого микропроцессора Intel 4004.

Это была микросхема фирмы **Intel**, получившая имя **i8048**.

Помимо *центрального процессора*, на кристалле находились 1 кбайт *памяти программ*, 64 байта *памяти данных*, два 8-битных *таймера*, тактовый генератор и 27 *портов ввода/вывода*.

Микроконтроллеры семейства 8048 использовались в игровых консольных приставках Magnavox Odyssey, в клавиатурах первых IBM PC и в ряде других устройств.



**Закон Мура.** В 1965 году один из основателей Intel *Гордон Мур* отметил, что новые модели микросхем разрабатывались спустя более-менее одинаковые периоды времени — 18-24 месяца — после появления их предшественников, при этом количество используемых в них транзисторов при этом возрастала каждый раз примерно вдвое.

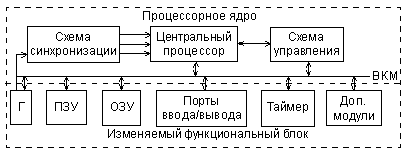
Наблюдение Мура впоследствии блестяще подтвердилось, а обнаруженная им закономерность наблюдается и в наши дни.

За 40 лет, истекшие с момента появления микропроцессора **i4004** в 1971 году и вплоть до выпуска процессора Xeon Westmere-EX®, количество транзисторов выросло более чем в 1,3 миллиона раз – с 2 300 до 2,6 миллиардов.

**Отличительные признаки микроконтроллеров**

1. **модульная организация**, при которой на базе одного ***процессорного ядра*** (центрального процессора) проектируется ряд (линейка) ***МК***, различающихся объемом и типом ***памяти программ***, объемом ***памяти данных***, набором периферийных модулей, частотой синхронизации;
2. **использование закрытой архитектуры** ***МК***, которая характеризуется отсутствием линий магистралей адреса и данных на выводах корпуса ***МК***. Таким образом, ***МК*** представляет собой **законченную систему обработки данных**, наращивание возможностей которой с использованием параллельных магистралей адреса и данных не предполагается;
3. **использование типовых функциональных периферийных модулей** (таймеры, процессоры событий, контроллеры последовательных интерфейсов, аналого-цифровые преобразователи и др.), имеющих незначительные отличия в алгоритмах работы в ***МК*** различных производителей;
4. **расширение числа режимов работы периферийных модулей**, которые задаются в процессе инициализации регистров специальных функций ***МК***.

При модульном принципе построения все ***МК*** одного семейства содержат ***процессорное ядро***, одинаковое для всех ***МК*** данного семейства, и изменяемый функциональный блок, который отличает ***МК*** разных моделей.



**INTEL**

Наиболее массовый микроконтроллер **Intel 8051**, выпущенный в 1980 году, стал классическим образцом устройств данного класса. Этот 8-битный чип положил начало целому семейству микроконтроллеров, которые господствовали на рынке вплоть до недавнего времени.

Большинство фирм производителей микроконтроллеров и сегодня выпускают устройства, основанные на этой архитектуре – **х51**. Среди них Philips, Atmel, Dallas, OKI, Siemens и т.д. (можно перечислить более полутора десятков имен).

Но **семейство х51 (система команд MCS-51) постепенно сдает свои позиции** более молодым и совершенным микроконтроллерам.

**Microchip**

Первые значительные перемены произошли с появлением **PIC**-контроллеров фирмы Microchip. Эти чипы предлагались по рекордно низким ценам, что позволило им в короткий срок захватить значительную часть рынка микроконтроллеров. К тому же кристаллы от Microchip оказались не уступающими, а нередко и превосходящими микроконтроллеры **х51** по производительности и не требовали дорогостоящего программатора.

**Atmel**

Настоящая революция в мире микроконтроллеров произошла в 1996 году, когда корпорация Atmel представила свое семейство чипов на новом прогрессивном ядре **AVR**.

Более продуманная архитектура AVR, быстродействие, превосходящее контроллеры Microchip, привлекательная ценовая политика и доступные средства разработки способствовали оттоку симпатий многих разработчиков от недавних претендентов на звание контроллера номер 1.

Микроконтроллеры AVR имеют **более развитую систему команд**, насчитывающую до 133 инструкций, высокую **производительность**, приближающуюся к 1 MIPS/МГц, **Flash ПЗУ программ с возможностью внутрисхемного перепрограммирования**. Многие чипы имеют функцию самопрограммирования.

**AVR-архитектура оптимизирована под язык высокого уровня Си.**

Кроме того, все кристаллы семейства совместимы "снизу вверх".

У Atmel много бесплатно распространяемых программных продуктов. Фирма Atmel уделяет этому вопросу большое внимание. Чрезвычайно удачная и совершенно бесплатная среда разработки **Atmel Studio**, работающая под Windows. Немаловажным является и то, что для программирования AVR можно обойтись вовсе без аппаратного программатора.

AVR становится еще одним индустриальным стандартом среди 8-разрядных микроконтроллеров общего назначения.

**МИКРОКОНТРОЛЛЕРЫ ATMEL AVR: ИСТОРИЯ СОЗДАНИЯ**

Идея разработки нового прогрессивного RISC-ядра зародилась в норвежском городе Тронхейм в головах двух студентов Norwegian University of Science and Technology (NTNU) **Альфа Богена** и **Вегарда Воллена**. Они создали такую архитектуру микроконтроллера, которая стала одной из самых удачных на мировом рынке микроконтроллеров.

В 1995 году Боген и Воллен предложили американской корпорации Atmel, известной на тот момент своим "ноу-хау" изготовления чипов с Flash-памятью, выпускать новый 8-битный RISC-микроконтроллер и снабдить его Flash-памятью программ на кристалле. Идея настолько понравилась руководству Atmel Corp., что было принято решение незамедлительно инвестировать в данный проект.

В конце 1996 года был выпущен опытный кристалл **AT90S1200**, а во второй половине 1997-го корпорация Atmel приступила к серийному производству нового семейства микроконтроллеров, к их рекламной и технической поддержке.

Новое ядро было запатентовано и получило название AVR, которое по прошествии уже нескольких лет стало трактоваться самыми различными способами. Кто-то утверждает, что это не иначе как **A**dvanced **V**irtual **R**ISC, другие полагают, что не обошлось здесь без **A**lf Egil Bogen **V**egard Wollan **R**ISC.

Держателями патента при этом являются: Wollan Vegard (NO); Bogen Alf-Egil (NO); Myklebust Gaute (NO); Bryant John, D. (US).

Система команд и внутреннее устройство чипов AVR разрабатывались совместно с фирмой **IAR Systems** – производителем компиляторов языков программирования С/С++, что обеспечило уникальные характеристики этих микроконтроллеров. В результате для AVR стало возможным получать **высокую плотность кода при использовании языков высокого уровня**, практически не теряя в производительности по сравнению с программами, написанными на низкоуровневом языке Ассемблера.

**Использование прогрессивной технологии конвейеризации** у AVR сокращало цикл "**выборка – исполнение**" команды.

Например, у микроконтроллеров семейства **x51** короткая команда выполняется за **12 тактов** генератора. В **PIC**-контроллерах фирмы Microchip, где уже реализован конвейер, короткая команда выполняется за **4 периода** тактовой частоты. В микроконтроллерах **AVR** короткая команда в общем потоке выполнялась всего за **один период** тактирующего сигнала. Такое построение кристалла обеспечило существенное повышение производительности, которая в пределе может достигать значения 1MIPS на 1МГц. **Это** во многих случаях при заданной производительности **позволяло снизить тактовую частоту**, а значит, **и потребляемую мощность устройства**.

**AVR-микроконтроллеры** предоставляли **более широкие возможности по оптимизации производительности/энергопотребления**, что было особенно важно при разработке приложений с батарейным питанием.

В 2003 году количество выпущенных Atmel Corp. микросхем с ядром AVR превысило **500 миллионов штук**!

**МОДЕЛИ МИКРОКОНТРОЛЛЕРОВ AVR**

В настоящее время в серийном производстве у Atmel находятся семейства AVR ATtiny, ATmega и ATxmega. Также все еще можно встретить в продаже чипы семейства Classic типа AT90.

**Classic AVR** – это классические (первые) модели AVR, что следует и из их названия. В настоящее время они представлены специализированными микроконтроллерами, например для работы с USB, CAN, для формирования многоканального ШИМ сигнала.

**Tiny AVR** – недорогие микроконтроллеры с малым количеством выходных контактов, чаще в восьмивыводном исполнении. Используются в системах, где не требуются сложные вычислительные алгоритмы.

**Mega AVR** – мощные микроконтроллеры, включающие большой набор периферии. Это семейство имеет самое большое разнообразие моделей для выбора.

**AVR XMEGA** – новое семейство компании Atmel переносит 8/16-битные микроконтроллеры на новый уровень системных характеристик. В этой связи, микроконтроллеры AVR XMEGA могут выступать в качестве эталонных 8/16-битных микроконтроллеров.

Независимо от выбора семейства микроконтроллеров AVR, следует помнить, что **система команд всех семейств совместима**, поэтому возможен простой перенос программы со слабого на более мощный микроконтроллер.

**Память**

**В микроконтроллерах AVR реализована архитектура**, в соответствии с которой разделены не только адресные пространства *памяти программ* и *памяти данных*, но и шины доступа к ним. Каждая из областей памяти данных расположена в своем адресном пространстве.

**Память программ**

Память программ в микроконтроллере AVR является энергонезависимой реализована по *flash* технологии и предназначена для хранения последовательности команд, управляющих функционированием микроконтроллера, и имеет 16-ти битную организацию.

**Все AVR имеют Flash-память программ**, которая может быть различного размера от 1 до 256 КБайт.

Программа заносится во Flash-память AVR как с помощью специального устройства − программатора, так и с помощью последовательных интерфейсов − SPI- или JTAG. Практически все микроконтроллеры AVR обладают возможностью внутрисхемного программирования (**функция ISP in system programming**) через коммуникационный интерфейс SPI.

**Все микроконтроллеры семейства Mega** имеют возможность **самопрограммирования**, т. е. самостоятельного изменения содержимого своей памяти программ.

**Память данных**

Память данных разделена на три части:

1. **регистровая память** (набор внутренних регистров микроконтроллера),
2. **оперативная память** (ОЗУ – оперативное запоминающее устройство с произвольной выборкой или RAM – Random Access Memory),
3. **энергонезависимая память** (ЭСППЗУ – электрически стираемое перепрограммируемое постоянное запоминающее устройство или EEPROM − Electrically Erasable Programmable Read-Only Memory).

**Регистровая память**

**Регистровая память** включает **32 регистра общего назначения** (РОН или GPR), объединенных в файл, и служебные **регистры ввода/вывода** (РВВ). И те и другие расположены в адресном пространстве ОЗУ, но не являются его частью.

В области **регистров ввода/вывода** расположены различные **служебные регистры** (регистры управления микроконтроллером, регистры состояния и т. п.), а также **регистры управления периферийными устройствами**, входящими в состав микроконтроллера.

**Управление микроконтроллером заключается в управлении этими регистрами**

**Энергонезависимая память данных**

**Для долговременного хранения** различной **информации**, которая может изменяться в процессе функционирования микроконтроллерной системы, используется **EEPROM-память**. **Все AVR имеют блок энергонезависимой электрически перезаписываемой памяти данных EEPROM** от 64 байт до 4 кбайт. Этот тип памяти, доступный программе микроконтроллера непосредственно в ходе ее выполнения, удобен для хранения промежуточных данных, различных констант, коэффициентов, серийных номеров, ключей и т.п. EEPROM может быть загружена извне как через SPI интерфейс, так и с помощью обычного программатора. **Число циклов стирание/запись – не менее 100 тыс.**

**Оперативная память**

**Внутренняя оперативная статическая память** (**Static RAM** − **SRAM**) имеет байтовый формат организации и используется для оперативного хранения данных.

Размер оперативной памяти может варьироваться у различных чипов от 64 байт до 4 кбайт. Число циклов чтения и записи в RAM не ограничено, но при отключении питающего напряжения вся информация теряется.

Для некоторых микроконтроллеров возможна организация подключения внешнего статического ОЗУ объемом до 64 кбайт

**Периферия микроконтроллера**

Периферия микроконтроллеров AVR включает:

* + **цифровые порты ввода/вывода** (от 3 до 48 линий ввода и вывода),
  + поддержку **внешних прерываний (IRQ)**,
  + **таймеры-счетчики (T/C)**,
  + **широтно-импульсные модуляторы (PWM)**
  + **сторожевой таймер (WD)**,
  + **аналоговые компараторы (AC)**,
  + **10-разрядный 8-канальный АЦП (ADC)**,
  + Последовательные интерфейсы **UART**, **JTAG** и **SPI**,
  + **устройство сброса** по понижению питания (**RESET**),

**Цифровые порты ввода/вывода**

***Порт ввода-вывода*** – логическое объединение сигнальных линий, через которое принимаются и передаются данные.

В зависимости от реализуемых функций различают следующие типы параллельных портов:

1. **однонаправленные порты**, предназначенные **только для ввода** или **только для вывода** информации;
2. **двунаправленные порты**, **направление передачи** которых (ввод или вывод) **определяется в процессе инициализации МК;**
3. **порты с альтернативной функцией** (мультиплексированные порты). Отдельные линии этих портов используются совместно со встроенными периферийными устройствами МК, такими как таймеры, АЦП, контроллеры последовательных интерфейсов;
4. **порты с программно управляемой схемотехникой** входного/выходного буфера.

Порты ввода/вывода можно назвать Периферией Независимой от Ядра, так как входы портов являются асинхронными, состояние портов сохраняется в спящем состоянии ядра.

**Входные элементы управления**

В качестве входных элементов управления могут использоваться

* + тумблеры;
  + джамперы;
  + множественные переключатели;
  + кнопки;
  + другие элементы схемы.

**Выходные элементы управления**

В качестве выходных элементов управления могут использоваться

* + элементы индикации (единичные светодиоды или светодиодные сборки);
  + Электромагнитные реле;
  + другие элементы схемы.

**Порт в МК выполняет роль** *устройства согласования* между самим **МК и объектом управления**, которые в общем случае **работают асинхронно**.

**Три типа алгоритмов обмена** информацией между МК и внешним устройством через параллельные порты ввода/вывода:

1. **режим простого программного ввода/вывода;**
2. **режим ввода/вывода со стробированием (синхронный);**
3. **режим ввода/вывода с полным набором сигналов подтверждения обмена (асинхронный).**

**Порты ввода/вывода** AVR имеют от 3 до 53 независимых линий "вход/выход".

Каждая линия порта может быть запрограммирована на вход или на выход.

**Мощные выходные драйверы обеспечивают высокую токовую нагрузочную способность 20 мА на линию порта** (**втекающий ток**) при максимальном значении **40 мА**, что позволяет, например, непосредственно подключать к микроконтроллеру светодиоды и биполярные транзисторы. **Общая токовая нагрузка на все линии одного порта не должна превышать 80 мА** (все значения приведены для напряжения питания 5 В).

**Архитектурная особенность** построения портов ввода/вывода у AVR заключается в том, что **для каждого физического вывода (пина, ножки) существует 3 бита контроля/управления**, а не 2, как у других распространенных 8-разрядных микроконтроллеров. Это повышает скорость работы микроконтроллера при работе с внешними устройствами, особенно в условиях внешних электрических помех.

**Прерывания (INTERRUPTS)**

**Система прерываний** – **одна из важнейших частей микроконтроллера**.

Все микроконтроллеры AVR имеют многоуровневую систему прерываний.

Прерывание прекращает нормальный ход программы для выполнения приоритетной задачи, определяемой внутренним или внешним *событием*.

Для обработки каждого такого события разрабатывается отдельная программа, которую называют *подпрограммой обработки запроса на прерывание* (для краткости − подпрограммой прерывания), и размещается в памяти программ.

При возникновении события, вызывающего прерывание, микроконтроллер сохраняет содержимое счетчика команд, прерывает выполнение центральным процессором текущей программы и переходит к выполнению подпрограммы обработки прерывания.

После выполнения подпрограммы прерывания осуществляется восстановление предварительно сохраненного счетчика команд и процессор возвращается к выполнению прерванной программы.

Для каждого события может быть установлен приоритет. Понятие приоритет означает, что выполняемая подпрограмма прерывания может быть прервана другим событием только при условии, что оно имеет более высокий приоритет, чем текущее. В противном случае центральный процессор перейдет к обработке нового события только после окончания обработки предыдущего.

**Таймеры/счетчики**

Микроконтроллеры AVR имеют в своем составе от 1 до 4 таймеров/счетчиков (TIMER/COUNTERS) с разрядностью 8 или 16 бит, которые могут работать и как таймеры от внутреннего источника тактовой частоты, и как счетчики внешних событий.

Их можно использовать:

1. **для точного формирования временных интервалов,**
2. **подсчета внешних импульсов на выводах микроконтроллера,**
3. **формирования последовательности импульсов,**
4. **формировать широтно-импульсную модуляцию ШИМ (PWM) с программируемыми частотой и скважностью.**

Таймеры/счетчики способны вырабатывать запросы на прерывания, переключая процессор на их обслуживание по событиям и освобождая его от необходимости периодического опроса состояния таймеров.

Поскольку основное применение микроконтроллеры находят в системах управления реального времени, **таймеры/счетчики являются одним из наиболее важных элементов таких систем.**

**Сторожевой таймер**

**Сторожевой таймер** (**WDT − WatchDog Timer**) предназначен для предотвращения катастрофических последствий от случайных сбоев программы. Он имеет свой собственный **RC-генератор**, работающий на частоте **1 МГц** от встроенного генератора.

**Работа с аналоговыми сигналами**

**Аналоговый компаратор**

**Аналоговый компаратор** (*Analog Comparator*) сравнивает напряжения на двух выводах (пинах) микроконтроллера. Результатом сравнения будет логическое значение, которое может быть прочитано из программы.

Выход аналогового компаратора можно включить на *прерывание* *от аналогового компаратора*. Пользователь может установить срабатывание прерывания по нарастающему или спадающему фронту или по переключению.

Присутствует у всех современных AVR, кроме Mega8515

**Аналого-цифровой преобразователь**

**Аналого-цифровой преобразователь** (АЦП, ADС – analog to digital cjnverter) служит для получения числового значения напряжения, поданного на его вход. Этот результат сохраняется в регистре данных АЦП. Какой из выводов (пинов) микроконтроллера будет являться входом АЦП (их может быть до 8 у AVR), определяется числом, занесенным в соответствующий регистр.

**Последовательные периферийные интерфейсы**

**Универсальный последовательный приемопередатчик (UART или USART)**

**Универсальный асинхронный или универсальный синхронно/ асинхронный приемопередатчик** (Universal Synchronous/Asynchronous Receiver and Transmitter – **UART** или **USART**) – удобный и простой последовательный интерфейс для организации информационного канала обмена микроконтроллера с внешним миром. Способен работать в дуплексном режиме (одновременная передача и прием данных). Он **поддерживает интерфейс стандарта RS-232**, что обеспечивает возможность организации связи с персональным компьютером. (Для стыковки МК и компьютера обязательно понадобится схема сопряжения уровней сигналов. Для этого существуют специальные микросхемы, например MAX232).

**Последовательный периферийный интерфейс SPI**

**Последовательный периферийный трехпроводный интерфейс SPI** (Serial Peripheral Interface) предназначен для организации обмена данными между двумя устройствами. С его помощью может осуществляться обмен данными между микроконтроллером и различными устройствами, такими, как цифровые потенциометры, ЦАП/АЦП, FLASH-ПЗУ и др. С помощью этого интерфейса удобно производить обмен данными между несколькими микроконтроллерами AVR.

Кроме того, **через интерфейс SPI может осуществляться программирование микроконтроллера**.

**Двухпроводной последовательный интерфейс TWI**

**Двухпроводной последовательный интерфейс TWI** (Two-wire Serial Interface) **является полным аналогом базовой версии интерфейса** **I2C** (двухпроводная двунаправленная синхронно-асинхронная шина) фирмы **Philips**. Этот интерфейс позволяет объединить вместе до **128** различных устройств с помощью двунаправленной шины, состоящей из двух линий - линии тактового сигнала (**SCL**) и линии данных (**SDA**).

**Интерфейс JTAG**

**Интерфейс JTAG** был разработан группой ведущих специалистов по проблемам тестирования электронных компонентов (**Joint Test Action Group**) и был зарегистрирован в качестве промышленного стандарта IEEE Std 1149.1-1990.

**Четырехпроводной интерфейс JTAG используется для тестирования печатных плат, внутрисхемной отладки, программирования микроконтроллеров.**

Многие микроконтроллеры семейства Mega имеют совместимый с IEEE Std 1149.1 интерфейс **JTAG** или **debugWIRE** для встроенной отладки. Кроме того, все микроконтроллеры Mega с флэш-памятью емкостью 16 кбайт и более могут программироваться через интерфейс JTAG.

**Тактовый генератор**

Тактовый генератор вырабатывает импульсы для синхронизации работы всех узлов микроконтроллера. Внутренний тактовый генератор AVR может запускаться от нескольких источников опорной частоты (**внешний генератор**, **внешний кварцевый резонатор**, **внутренняя или внешняя RC-цепочка**). Минимальная допустимая частота ничем не ограничена (вплоть до пошагового режима). Максимальная рабочая частота определяется конкретным типом микроконтроллера и указывается Atmel в его характеристиках, хотя практически любой AVR-микроконтроллер с заявленной рабочей частотой, например, в 10 МГц при комнатной температуре легко может быть "разогнан" до 12 МГц и выше.

**Система реального времени**

**Система реального времени – (RTC** – real time clock) реализована во всех микроконтроллерах Mega и в двух кристаллах "classic" - AT90(L)S8535. Таймер/счетчик RTC имеет отдельный предделитель, который может быть программным способом подключен или к источнику основной тактовой частоты, или к дополнительному асинхронному источнику опорной частоты (кварцевый резонатор или внешний синхросигнал). Для этой цели зарезервированы два вывода микросхемы. Внутренний осциллятор оптимизирован для работы с внешним "часовым" кварцевым резонатором **32,768 кГц**.

**ПРОГРАММИРОВАНИЕ МИКРОКОНТРОЛЛЕРОВ AVR**

**Языки**

Для микроконтроллеров AVR существуют различные языки программирования, но, пожалуй, наиболее подходящими являются ассемблер и Си, поскольку в этих языках в наилучшей степени реализованы все необходимые возможности по управлению аппаратными средствами микроконтроллеров.

**Ассемблер** – это низкоуровневый язык программирования, использующий непосредственный набор инструкций микроконтроллера. Создание программы на этом языке требует хорошего знания системы команд программируемого чипа и достаточного времени на разработку программы. Ассемблер проигрывает Си в скорости и удобстве разработки программ, но имеет заметные преимущества в размере конечного исполняемого кода, а соответственно, и скорости его выполнения.

**Си** – позволяет создавать программы с гораздо большим комфортом, предоставляя разработчику все преимущества языка высокого уровня.

Архитектура и система команд AVR создавалась при непосредственном участии разработчиков компилятора языка Си и в ней учтены особенности этого языка. **Компиляция исходных текстов, написанных на Си, осуществляется быстро и дает компактный, эффективный код.**

**Основные преимущества Си перед ассемблером**:

1. **высокая скорость разработки программ;**
2. **универсальность, не требующая досконального изучения архитектуры микроконтроллера;**
3. **лучшая документируемость и читаемость алгоритма;**
4. **наличие библиотек функций;**
5. **поддержка вычислений с плавающей точкой.**
6. В языке Си гармонично сочетаются возможности программирования низкого уровня со свойствами языка высокого уровня. Возможность низкоуровневого программирования позволяет легко оперировать непосредственно аппаратными средствами, а свойства языка высокого уровня позволяют создавать легко читаемый и модифицируемый программный код. Кроме того, практически все компиляторы Си имеют возможность использовать ассемблерные вставки для написания критичных по времени выполнения и занимаемым ресурсам участков программы.
7. Одним словом, **Си – наиболее удобный язык** как для начинающих знакомиться с микроконтроллерами AVR, так и для серьезных разработчиков

# ЛК5 (Октябрь)

**Двухпроводной последовательный интерфейс TWI / I2C**

Двухпроводной последовательный интерфейс **TWI** (Two-wire Serial Interface) является полным аналогом базовой версии интерфейса **I2C** (двухпроводная двунаправленная шина) фирмы Philips. Этот интерфейс позволяет объединить вместе до 128 различных устройств с помощью двунаправленной шины, состоящей из линии тактового сигнала (**SCL**) и линии данных (**SDA**).

Двухпроводной интерфейс (**TWI**) — двунаправленная двухпроводная последовательная шина передачи данных, совместимая со стандартными шинами **I2C** и **SMBus**.

Устройство, подключенное к шине, должно быть либо ведущим, либо подчиненным. Ведущее устройство инициирует передачу данных путем передачи адреса подчиненного устройства и типа передачи: чтение или запись. Если к шине подключено несколько ведущих устройств и некоторая их часть одновременно инициировала передачу, применяется механизм арбитража, который учитывает приоритет этих устройств.

Модуль **TWI** микроконтроллеров может работать и в роли ведущего, и в роли подчиненного устройства.

Ведущая и подчиненная работа полностью отделены друг от друга и предусматривают отдельное управление включением/отключением. Для этих функций также предусмотрены отдельные регистры управления и статуса, а также векторы прерываний. Потеря арбитража, ошибки, коллизии и удержание линии синхронизации обнаруживаются на аппаратном уровне и индицируются отдельными флагами статуса для ведущего и подчиненного режимов.

Двухпроводной последовательный интерфейс **TWI** (Two-wire Serial Interface) является полным аналогом базовой версии интерфейса **I2C** (двухпроводная двунаправленная шина) фирмы Philips. Этот интерфейс позволяет объединить вместе до 128 различных устройств с помощью двунаправленной шины, состоящей из линии тактового сигнала (**SCL**) и линии данных (**SDA**).

Двухпроводной интерфейс (**TWI**) — двунаправленная двухпроводная последовательная шина передачи данных, совместимая со стандартными шинами **I2C** и **SMBus**.

Устройство, подключенное к шине, должно быть либо ведущим, либо подчиненным. Ведущее устройство инициирует передачу данных путем передачи адреса подчиненного устройства и типа передачи: чтение или запись. Если к шине подключено несколько ведущих устройств и некоторая их часть одновременно инициировала передачу, применяется механизм арбитража, который учитывает приоритет этих устройств.

Модуль **TWI** микроконтроллеров может работать и в роли ведущего, и в роли подчиненного устройства.

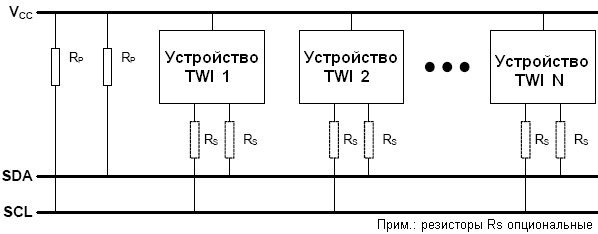
Ведущая и подчиненная работа полностью отделены друг от друга и предусматривают отдельное управление включением/отключением. Для этих функций также предусмотрены отдельные регистры управления и статуса, а также векторы прерываний. Потеря арбитража, ошибки, коллизии и удержание линии синхронизации обнаруживаются на аппаратном уровне и индицируются отдельными флагами статуса для ведущего и подчиненного режимов.

***Принцип действия шины TWI***

Двухпроводной интерфейс (**TWI**) подключается к простой двухпроводной и двунаправленной шине, которая состоит из двух линий: линия синхронизации (**SCL**) и линия последовательной передачи данных (**SDA**). Источниками сигналов для обеих линий являются схемы с открытым коллектором в выходном каскаде (используется принцип "монтажного И"). Для создания высоких логических уровней на линиях шины, в т.ч. когда к шине не подключено ни одно устройство, необходимы подтягивающие к плюсу питания резисторы (Rp) — единственно необходимые внешние компоненты. Альтернативно, вместо подтягивающих резисторов могут использоваться источники тока.

Шина **TWI** является простым и эффективным средством соединения по последовательной шине нескольких устройств.

Устройство, подключенное к шине, может быть либо ведущим (**master**), либо подчиненным (**slave**). Чтобы передача по шине стала возможной, к ней должно быть подключено как минимум одно ведущее устройство.



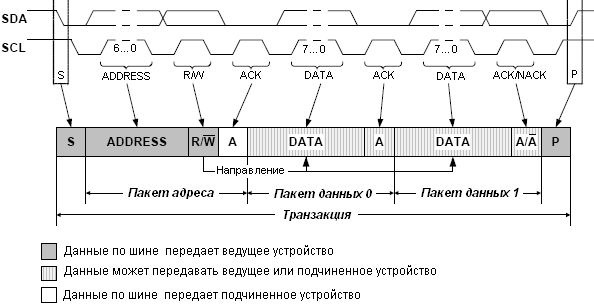
Каждое подключенное к шине подчиненное устройство должно иметь собственный неповторяющийся адрес, передачей которого ведущее устройство инициирует транзакцию. Могут использоваться 7- или 10-битные адреса. К одной шине допускается подключение нескольких ведущих устройств. Такая шина называется мультимастерной. Для определения, какое из одновременно осуществляющих доступ к шине ведущих устройств имеет приоритет, используется механизм арбитража.

Устройство может содержать, как логику ведущего устройства, так и логику подчиненного устройства. Кроме того, оно может эмулировать работу нескольких подчиненных устройств, путем реагирования на несколько адресов.

Ведущее устройство сигнализирует о начале транзакции генерацией на шине условия **START** (S). Затем, передается пакет адреса с адресом подчиненного устройства (**ADDRESS**) и указание желаемого направления передачи данных, т.е. чтение или запись (**R/W**). По завершении передачи всех пакетов данных (**DATA**), ведущее устройство генерирует на шине условие **STOP** (P) и транзакция завершается. После приема каждого байта получатель данных генерирует бит подтверждения (A или ACK) или неподтверждения (/A или NACK).

***Структура базовой транзакции шины TWI:***

Сигнал синхронизации транзакции генерирует ведущее устройство, однако, в целях снижения быстродействия шины, любое подключенное к шине устройство может увеличить длительность импульса низкого уровня.



**CAN BUS**

**CAN** (Control Area Network) — последовательная магистраль, обеспечивающая увязку в сеть "интеллектуальных" устройств ввода/вывода, датчиков и исполнительных устройств некоторого механизма или даже предприятия. Характеризуется протоколом, обеспечивающим возможность нахождения на магистрали нескольких ведущих устройств, обеспечивающим передачу данных в реальном масштабе времени и коррекцию ошибок, высокой помехоустойчивостью. Система **CAN** обеспечена большим количеством микросхем, обеспечивающих работу подключенных к магистрали устройств, разработку которых начинала фирма **BOSH** для использования в автомобилях, и в настоящее время широко используемых в автоматизации промышленности.

Предназначен для организации высоконадежных недорогих каналов связи в распределенных системах управления. Интерфейс широко применяется в промышленности, энергетике и на транспорте. Позволяет строить как дешевые мультиплексные каналы, так и высокоскоростные сети.

Скорость передачи задается программно и может быть до 1 Мбит/с. Пользователь выбирает скорость, исходя из расстояний, числа абонентов и емкости линий передачи.  
Максимальное число абонентов, подключенных к данному интерфейсу фактически определяется нагрузочной способностью примененных приемопередатчиков.

Протокол **CAN** использует оригинальную систему адресации сообщений. Каждое сообщение снабжается идентификатором, который определяет назначение передаваемых данных, но не адрес приемника. Любой приемник может реагировать как на один идентификатор, так и на несколько. На один идентификатор могут реагировать несколько приемников.  
Протокол **CAN** обладает развитой системой обнаружения и сигнализации ошибок.

Для этих целей используется поразрядный контроль, прямое заполнение битового потока, проверка пакета сообщения **CRC**-полиномом, контроль формы пакета сообщений, подтверждение правильного приема пакета данных. Хемминговый интервал d=6. Общая вероятность необнаруженной ошибки 4.7x10-11.

Система арбитража протокола **CAN** исключает потерю информации и времени при "столкновениях" на шине.

Интерфейс с применением протокола **CAN** легко адаптируется к физической среде передачи информации. Это может быть дифференциальный сигнал, оптоволокно, просто открытый коллектор и т.п.

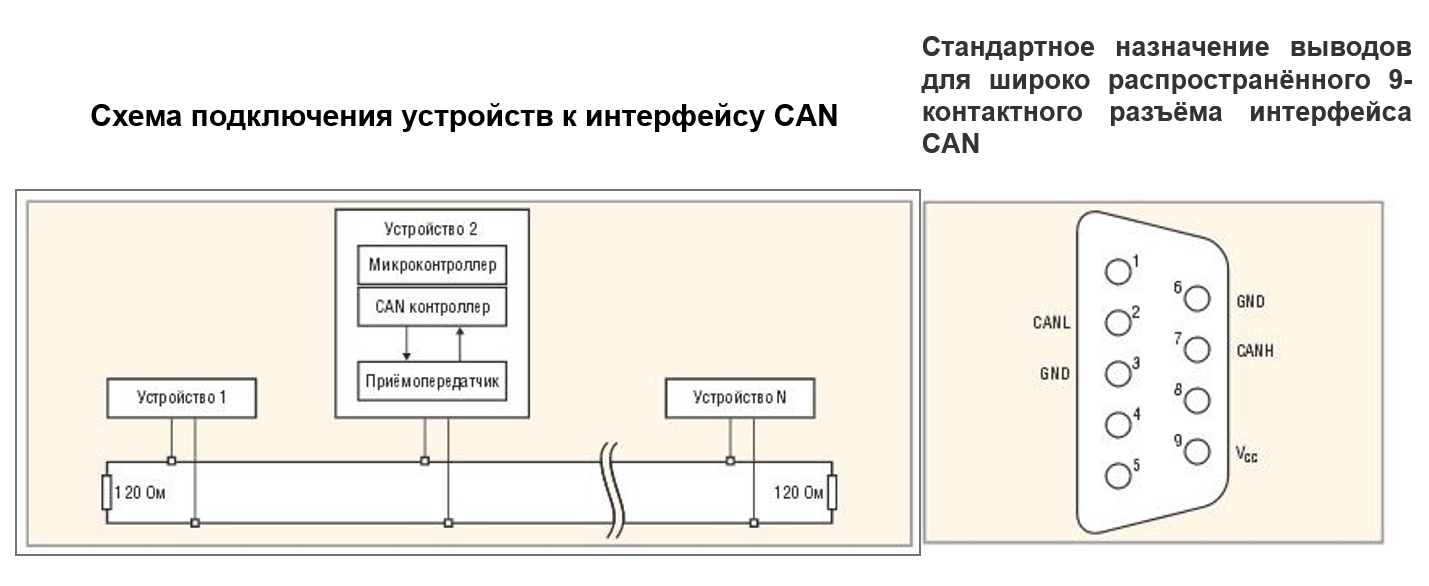
Несложно делается гальваническая развязка. Элементная база, поддерживающая **CAN**, широко выпускается в индустриальном исполнении.

**Описание интерфейса CAN**

Интерфейс CAN предназначен для организации последовательных, высоконадёжных и недорогих каналов связи в распределённых системах управления. Он позволяет организовывать как мультиплексные каналы, так и высокоскоростные сети. Данный интерфейс имеет протокол, поддерживающий возможность нахождения на магистрали нескольких ведущих устройств и обеспечивает передачу данных в реальном масштабе времени.

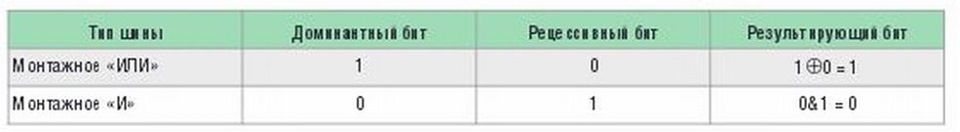
Интерфейс обладает высокой помехоустойчивостью благодаря коррекции ошибок. Передача данных осуществляется кадрами, которые принимаются всеми устройствами сети. Кадр состоит из идентификатора длиной 11 бит для стандартного формата или 29 бит для расширенного формата и блока данных, содержащего от 0 до 8 байт. Идентификатор описывает содержимое пакета данных и служит для определения приоритета при попытке одновременной передачи несколькими устройствами.

Скорость передачи данных выбирается исходя из расстояния, числа абонентов в сети и ёмкости линии связи. Она задаётся программно и может составлять от десятка Кбод до единиц Мбод. Стандарт интерфейса CAN определяет передачу данных независимо от физического уровня, т.е. канал связи может быть каким угодно, например, радиоканалом или оптоволокном. Однако на практике под CAN-интерфейсом обычно подразумевается сеть с физическим каналом связи в виде дифференциальной пары проводов, определённым в стандарте ISO 11898.



Интерфейс **CAN** относится к типу сетей CR (Collision Resolution, разрешение коллизий), в отличие от сетей типа CD (Collision Detect, обнаружение коллизий), например **Ethernet**. Тип сетей CR обеспечивает приоритетный доступ к передаче сообщения, что необходимо для промышленных устройств.

Приоритетный доступ к передаче сообщений в стандарте ISO-11898 реализуют так называемые рецессивные и доминантные биты. В зависимости от типа шины, объединяющей устройства **CAN**, эти биты могут принимать значение лог. 0 либо лог. 1. В таблице приведены два примера организации шин, поясняющие образование управляющих битов.

Спецификация интерфейса **CAN** избегает описания двоичных значений сигналов, как лог. 0 либо лог. 1, с целью абстрагирования от среды передачи. Именно поэтому вместо них применяются термины «рецессивный» и «доминантный» биты. При этом подразумевается, что при передаче одним устройством сети рецессивного бита, а другим доминантного, принят будет доминантный бит. Например, при реализации физического уровня с помощью радиоканала отсутствие сигнала означает рецессивный бит, а наличие сигнала — доминантный.

Стандарт сети требует от физического уровня передачи, чтобы доминантный бит мог подавить рецессивный, но не наоборот. Например, в оптическом волокне доминантному биту должен соответствовать свет, а рецессивному — его отсутствие. В электрическом проводе рецессивное состояние соответствует высокому потенциалу на линии связи, а доминантное — низкому потенциалу, т.е. когда устройство подключает линию связи к нулевому потенциалу. Если линия находится в рецессивном состоянии, перевести её в доминантное состояние может любое устройство сети, например, включив свет в оптоволокне.

***Кадры и их форматы***

Кроме рецессивного и доминантного битов, стандарт интерфейса CAN оперирует таким понятием, как кадр, который используется в протоколе обмена информацией между устройствами. Каждый кадр представляет собой единый набор нескольких служебных битов и байтов. Существуют следующие виды кадров:

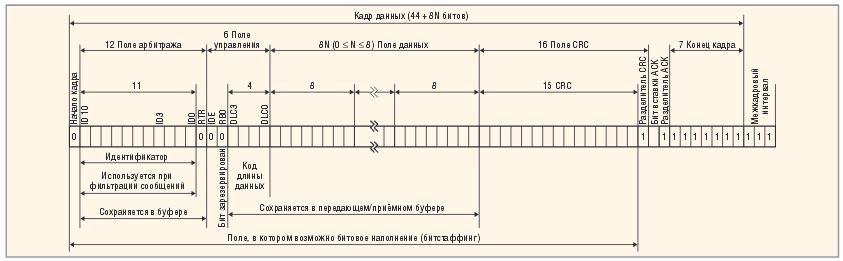
• **кадр данных** (data frame) служит для передачи данных;  
• **кадр запроса передачи** (remote frame) служит для запроса на передачу кадра данных с тем же идентификатором;  
• **кадр перегрузки** (overload frame) обеспечивает промежуток между кадрами данных или кадрами запросов;  
• **кадр ошибки** (error frame) передаётся узлом, обнаружившим ошибку.  
В таблицах ниже представлены базовый формат и расширенный формат кадра данных.



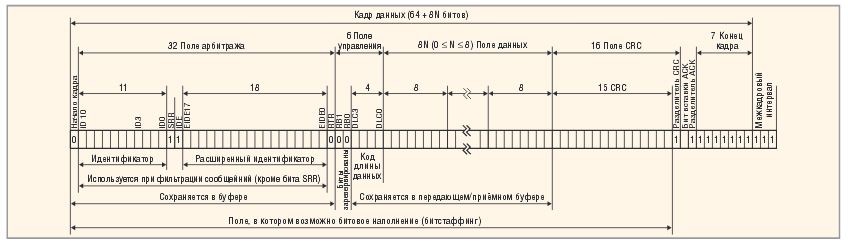


Графическое представление данных кадров, с пояснениями и порядком следования бит, приведено на рисунках ниже. Межкадровый интервал служит для разделения кадров между собой и состоит как минимум из трёх рецессивных бит. Он необходим передающему устройству для подготовки очередного кадра.

Стандартный кадр данных



 Расширенный кадр данных



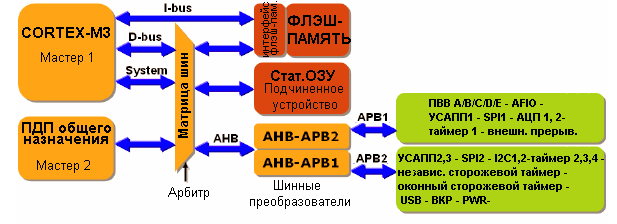
# ЛК6

**ВСТРОЕННЫЕ СИСТЕМЫ УПРАВЛЕНИЯ. МИКРОКОНТРОЛЛЕРНАЯ   
ПЛАТФОРМА STM32**

**Экономическая целесообразность использования микроконтроллеров STM32**

* Современный уровень развития технологии обеспечил возможность создания микроконтроллеров с 32- битной архитектурой, цена которых сопоставима с 8-битными микроконтроллерами
* **32-х битное ядро** для реализации требует **несколько десятков тысяч транзисторов**, в то время как **256Кб flash-памяти** — это   
  **2 млн транзисторов**. Соответственно, основное место площади современного контроллера занимает память, порты ввода/вывода и периферийные модули
* Уменьшение площади кристалла даже в 2 раза не уменьшает стоимость вдвое, так как львиную долю стоимости микроконтроллера составляет механическая обработка
* Компания STMicroelectronics разрабатывала 32 -битные микроконтроллеры на основе ядер ARM7 и ARM9
* Новое поколение 32- битных микроконтроллеров компании STMicroelectronics использует ядро CORTEX M3 и обеспечивает фантастическое соотношение цена/качество. При больших партиях стоимость микроконтроллера доходит до 1 евро

**Архитектура микроконтроллеров STM32**



* Микроконтроллеры семейства STM32 выполнены на основе ядра Cortex-M3, которое подключено к flash-памяти по шине инструкций   
  I-bus
* Шина данных D-bus и системная шина System Cortex подключены к матрице высокоскоростных шин AHB
* Внутреннее статическое ОЗУ подключено напрямую к матрице шин AHB, с которой также связан блок прямого доступа к памяти (ПДП).

**Обзор платформы STM32**

Микроконтроллеры STM32 изначально выпускались в 14 различных вариантах, разделенные на две группы:

* Performance Line, в которую вошли микроконтроллеры с тактовой частотой ЦПУ до 72 МГц;
* Access Line (тактовая частота до 36 МГц)

**Энергопотребление**

* Микроконтроллеры работают от 2В-ого источника питания на тактовой частоте 72МГц и потребляют с учетом нахождения в активном состоянии всех встроенных ресурсов, всего лишь 36 мА
* Если же использовать поддерживаемые ядром Cortex экономичные режимы работы, то потребляемый ток можно снизить до 2 мкА в режиме STANDBY
* Для быстроты возобновления активной работы микроконтроллера используется внутренний RC-генератор на частоту 8 МГц. Его активность сохраняется на время запуска внешнего генератора. Благодаря быстроте перехода в экономичный режим работы и выхода из них результирующая средняя потребляемая мощность еще больше снижается

**АРХИТЕКТУРА**

**Контроллер вложенных векторных прерываний (NVIC)**

NVIC – неотъемлемая часть процессора Cortex-M3. Модуль NVIC в процессоре Cortex-M3 осуществляет обработку прерываний аппаратными средствами.

NVIC отвечает за генерацию прерываний на различные события:

* внешние — изменение логического уровня на входе ножки, пробуждение из режима сна;
* внутренние — завершение приёма/отправки данных, переполнение счётчика таймера и т.п.

NVIC имеет несколько типов конфигурации:

* Стандартная конфигурация: поддержка одного немаскируемого прерывания (NMI), 32 физических прерывания общего назначения с восемью уровнями приоритета;
* Улучшенная конфигурация: поддержка от 1 до 240 физических прерываний с количеством уровней приоритета до 256

**Шинная матрица (Bus matrix)**

Шинная матрица — это развитие идеи простого контроллера шины: здесь шины соединены так, что устройства могут взаимодействовать напрямую, не через ядро.

Архитектурой Cortex-M3 предусмотрены 4 шины, подключенных к матрице:

* ICode, для выборки инструкций и векторов прерываний — для пользовательского кода. 32-битная шина AHB-Lite типа;
* DCode, для выборки/записи данных и отладочного доступа — для пользовательского кода. 32-битная шина AHB-Lite типа;
* System, для выборки инструкций и векторов прерываний, а также выборки/записи данных и отладочного доступа в системном пространстве — для внутренних компонентов МК. 32-битная шина AHB-типа;
* PPB (Private Peripheral Bus), для выборки/записи данных и отладочного доступа — для периферии. 32-битная шина APB-типа

**Ядро Cortex-M3**

* Выполняет инструкции, производит вычисления в своём арифметико-логическом устройстве (АЛУ)
* Его Гарвардская архитектура позволяет одновременно загружать инструкции и осуществлять доступ к памяти — благодаря этому, а также трёхступенчатому конвейеру, большинство инструкций выполняются за 1 такт
* Ядро Cortex-M3 поддерживает набор инструкций Thumb-2, который содержит как 32-битные, так и 16-битные инструкции для сокращения объёма кода за счёт менее дальнобойных переходов; имеет 13 регистров общего назначения, снижая потребность в частом доступе к памяти

**СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ**

**Библиотека поддержки ядра**

* Ядро ARM Cortex-M3 выходит за рамки обычного понятия ядра микроконтроллера и представляет собой мини-микроконтроллер с периферией - встроенные системный таймер, контроллер прерываний и т.д.
* Стандартная библиотека поддержки ядра – CMSIS (разработана компанией ARM)
* CMSIS предоставляет собой файлы определения констант и определения символьных имен, библиотеку функций доступа к регистрам и периферийным модулям ядра и интерфейса пользовательского ПО для операционных систем реального времени (RTOS)

**Структура CMSIS**

CMSIS состоит из трех файлов:

* core\_m3.h - вспомогательные функции доступа к регистрам ядра;
* startup\_stm32f10x\_xx.s - набор файлов для каждой линейки семейства STM32, обеспечивающие инициализацию стека и таблицу векторов прерываний;
* system\_stm32f10x.h - файл начальной инициализации тактовой частоты микроконтроллера

**Стандартная библиотека STM32**

Стандартная библиотека для работы с периферийными модулями написана в соответствии со стандартом ANSI C и может использоваться с любым стандартизованным компилятором

Библиотека состоит из двух взаимодополняющих составляющих:

* заголовочных файлов и файлов реализации всей периферии микроконтроллеровSTM32 - STM32F10x\_StdPeriph\_Driver;
* заголовочных файлов и файлов реализации ядра ARM Cortex-M3

Вся функциональность периферийных модулей описана в заголовочных файлах и файлах реализации. Например, для портов ввода-вывода это два файла: stm32f10x\_gpio.h и stm32f10x\_gpio.c

**Модификация библиотек**

Библиотека содержит три файла, доступные для модификации пользователем:

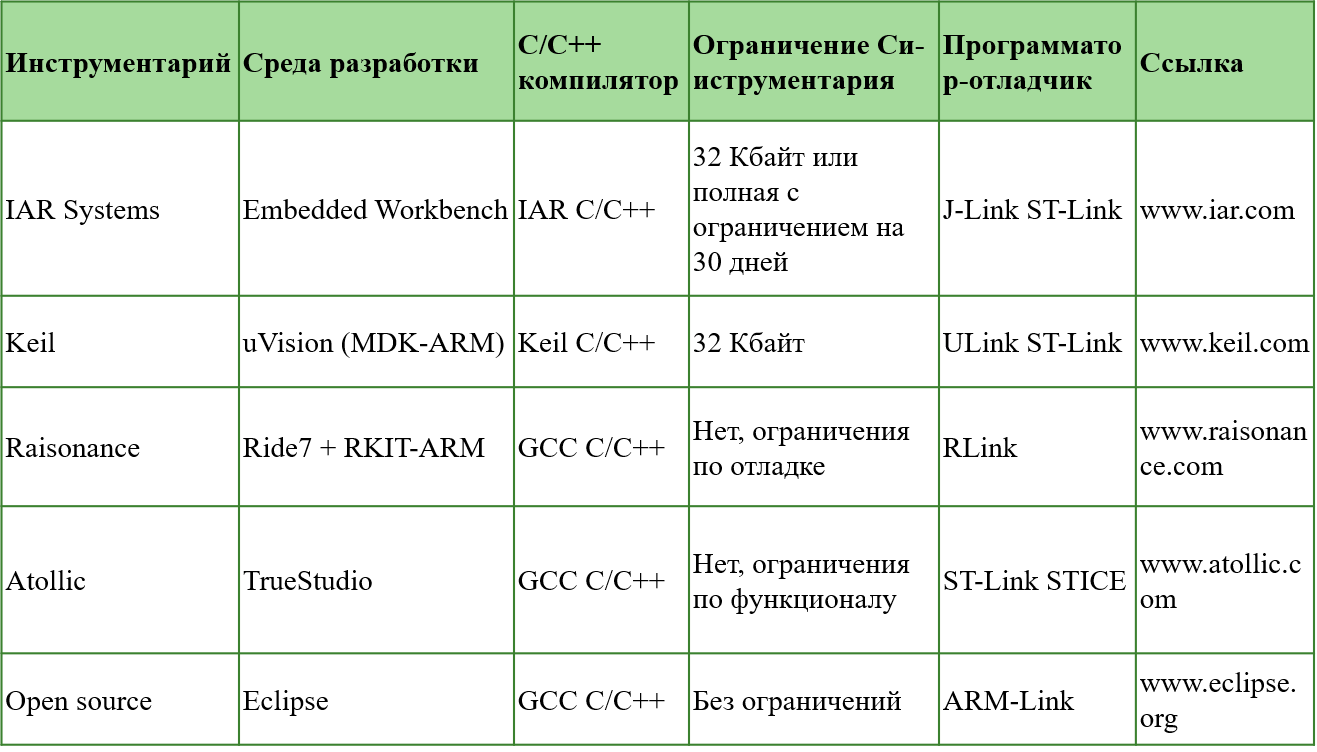
* файл конфигурации библиотеки stm32f10x\_conf.h
* файлы обработчиков прерываний stm32f10x\_it.h и stm32f10x\_it.c.

***Для использования определенных модулей периферии в проект необходимо добавить файлы реализации и сконфигурировать файл stm32f10x\_conf.h***

Под конфигурацией файла stm32f10x\_conf.h подразумевается удаление символов «комментарий» в строчках с названием периферийного модуля, предполагаемого для использования в конкретном проекте.

**ПРОГРАММНО-ИНСТРУМЕНТАЛЬНАЯ СРЕДА**

**Выбор программного инструментария под ARM-архитектуру**



**ОСВОЕНИЕ ПЛАТФОРМЫ**



**Поддержка разработчиков**

Стандартные библиотеки можно загрузить с официального сайта STMicroelectronics. Структура пакета библиотек следующая:

* Libraries:
* CMSIS (библиотека ядра ARM Cortex-M3);
* STM32F10x\_StdPeriph\_Lib\_V3.4.0\Libraries\STM32F10x\_StdPeriph\_Drive (библиотека периферии STM32)
* Project (типовые применения):
* STM32F10x\_StdPeriph\_Lib\_V3.4.0\Project\STM32F10x\_StdPeriph\_Examples
* STM32F10x\_StdPeriph\_Lib\_V3.4.0\Project\STM32F10x\_StdPeriph\_Template
* Utilities (драйвера для отладочных плат STMicroelectronics):
* STM32F10x\_StdPeriph\_Lib\_V3.4.0\stm32f10x\_stdperiph\_lib\_um.chm (файл справки)

# ЛК7

**Микроконтроллер ESP32**

Микроконтроллер ESP32 – это одна из самых доступных и мощных платформ для создания умных ардуино-проектов с поддержкой WiFi. Придя на смену ESP8266, этот чип дал новые возможности для разработчиков, хотя по-прежнему остались старые проблемы с поддержкой и документацией.

**Описание микроконтроллера ESP32**

Фирма Espressif выпустила мощный недорогой микроконтроллер ESP32 летом 2016 года. Устройство представляет собой систему на кристалле, построенную по технологии TSMC 40 нм, с Wi-Fi и Bluetooth контроллерами. Оно оснащено двухъядерным 32-битным процессором, который работает на частотах 80, 160 или 240 МГц. Также в систему интегрированы антенные коммутаторы, радиочастотные компоненты, фильтры, усилители, модули управления питанием. Подключается ESP32 к компьютеру через обычный USB провод.

**Характеристики чипа**

Технические характеристики ESP32:

* Двух- или одноядерный 32-битный процессор Tensilica Xtensa LX6;
* Тактовая частота – 160 или 240 МГц;
* 520 Кб SRAM;
* Максимальный ток потребления 260 мА, в спящем режиме – 10 мА;
* Стандарты беспроводной связи – Wi-Fi: 802.11 b / g / N, Bluetooth: v4.2 BR/EDR and BLE;
* Наличие датчиков температуры, Холла, тач-сенсоров;
* Инфракрасное дистанционное управление;
* Можно подключать двигатели и светодиоды через ШИМ разъем;
* Стандарт IEEE 802.11 с поддержкой WFA, WPA/WPA2 и WAPI;
* Возможность безопасной загрузки;
* Шифрование флэш диска.

Также в модуле традиционно присутствует встроенное управление энергопитанием. Для этого используются линейный регулятор, индивидуальное питание для RTC (ядро низкого энергопотребления), пробуждение по таймеру или сенсорному датчику.

Программирование модуля ESP32 может производиться на самых разных платформах, и средах, вот небольшой список наиболее популярных вариантов:

* Arduino IDE;
* Espressif IoT Development Framework;
* Espruino;
* PlatformIO;
* Pymakr IDE.

Большинство проектов реализуется на Arduino IDE и Espruino.

**Использование ESP 32**

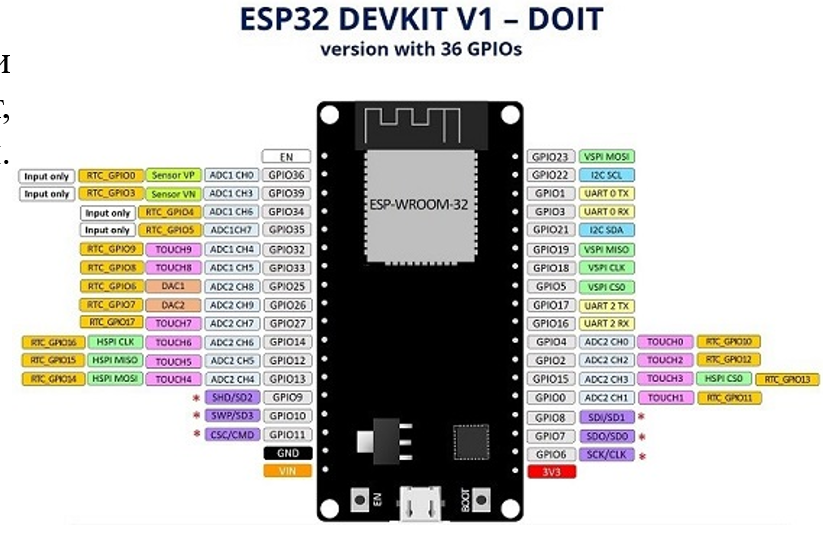
Хоть плата ESP32 появилась недавно, она уже активно используется в коммерческих проектах, связанных с мобильными приложениями, электроникой и задачами IoT. Например, на ее базе построен светодиодный браслет IoT группы Alibaba. Он представляет собой живой беспроводной экран, в котором каждый браслет работает как пиксель. Также на основе микроконтроллера реализована биометрическая система отслеживания посещаемости и проекты, связанные с анализом климатических условий. Имея плату ESP32 и датчики температуры, влажности и давления, можно самостоятельно собрать метеостанцию. ESP32 используется в музыкальных плеерах, помощниках с голосовым управлением, аудиогарнитуре.

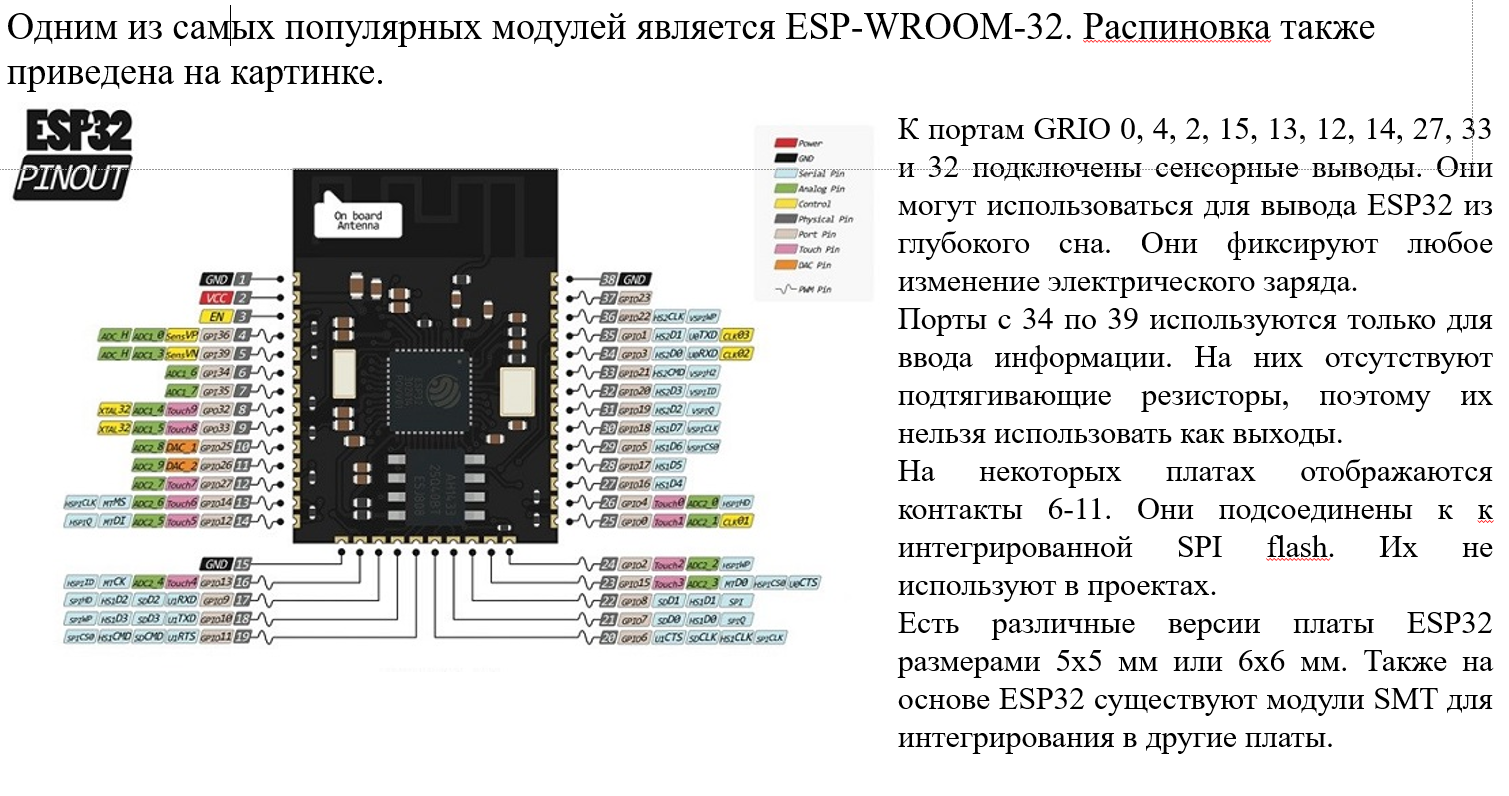
**Распиновка ESP32**

Микроконтроллер оснащен 48 контактами плюс 1 большой тепловой контакт, которые обладают разными функциями. Выводы микросхемы:

* 18 каналов 12-разрядного АЦП;
* 10 GRIO портов;
* 3 SPI;
* 3 UART;
* 2 I2C;
* 16 ШИМ выводов;
* 2 8-битных ЦАП вывода;
* 2

Расположение выводов зависит от производителя. Например, есть плата ESP32 DEVKIT V1 DOIT, у которой 36 контактов. Распиновка представлена на рисунке:





**Отличия esp32 от esp8266**

Платформы ESP8266 и ESP32 произведены одной компанией Espressif. Микроконтроллер ESP32 отличается от своего предшественника улучшенными характеристиками,  увеличенным функционалом и большим объемом памяти. Стоимость нового прибора дороже примерно в 2 раза.

Важным преимуществом ESP32 является более быстрый Wi-Fi и Bluetooth. В ESP32 установлен более мощный процессор, позволяющий реализовывать сложные проекты. Эта платформа подходит для приложений, в которых требуется интернет или новые интерфейсы. Для более дешевых разработок используется ESP8266.

Объем памяти у нового устройства ESP32 увеличен – 512 Кб против 160 Кб ESP8266. Также ESP32 отличается большим количеством выводов GRIO. К нескольким контактам на ESP32 прикреплены емкостные сенсорные датчики и датчик температуры. На обоих устройствах контакты GRIO можно использовать по-разному.  ESP32 имеет 18 12-битных АЦП каналов. У его предшественника есть всего 1 10-битный вывод АЦП.

Мощность процессора значительно влияет на скорость работы. Модуль ESP32 показывает рекордную производительность по сравнению с предшественником ESP8266. Загрузка страницы с длинным скетчем и множеством графики занимает секунды.

Из недостатков ESP32 можно выделить отсутствие библиотек для поддержки сенсоров и малое количество драйверов. Это связано с тем, что плата появилась в продаже недавно. Но учитывая все преимущества микроконтроллера и его перспективы, эта проблема будет решена уже в ближайшее время.

**Настройка Arduino IDE для работы с ESP32**

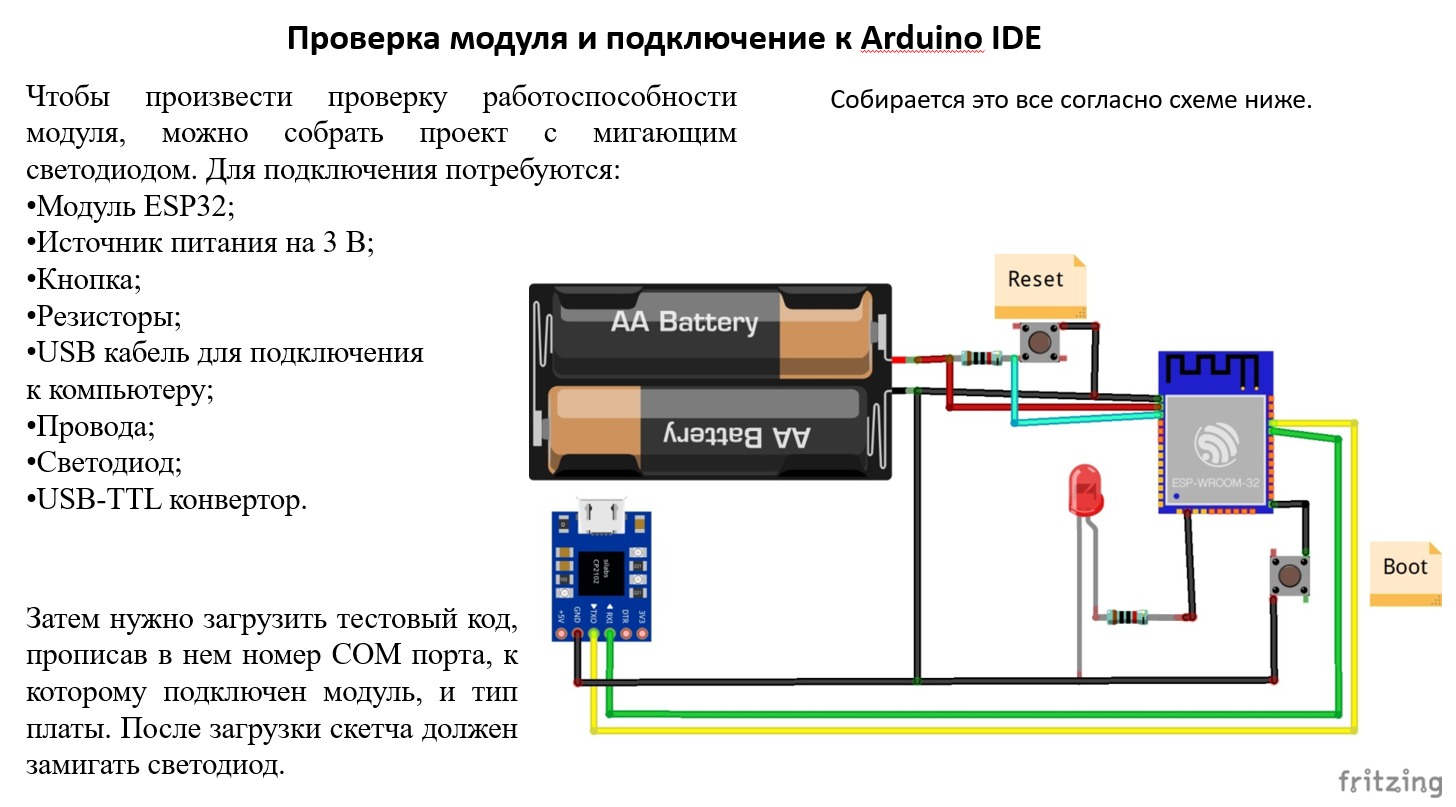
Изначально изделия от компании Espressif поставляются с прошивкой, позволяющей работать с помощью AT команд. Это не всегда удобно, поэтому лучше программировать плату в привычной среде разработки – Arduino IDE.

Чтобы начать создавать проекты на ESP32, сначала нужно иметь его поддержку в среде разработки Arduino IDE. Поддержка в настоящий момент находится на начальном, но работоспособном уровне. Проблемы могут возникнуть с драйверами устройств, но из-за популярности модуля вскоре будет поддерживаться вся периферия.

В первую очередь нужно скачать Arduino IDE и официальную инструкцию с сайта <https://github.com/espressif/arduino-esp32>. Нужно загрузить файлы дистрибутива и поместить в папку C:\Users\User\Documents\Arduino\hardware\espressif\esp32.

Так как плата ESP32 новая и поддержка в среде разработки Ардуино появилась недавно, то могут возникнуть сложности с установкой. Это связано с тем, что драйвера только разрабатываются и изменен порядок расположения системных директорий. Все скетчи должны быть расположены в папке C:\Users\User\Documents\Arduino. Файлы дистрибутива должны быть размещены внутри этой папки, как советует производитель. Если этого не учесть, то поддержка в Ардуино ESP32 будет отсутствовать.

После установки можно открыть Arduino IDE. Затем нужно перейти в настройки и в менеджере плат выбрать нужную. Теперь можно прошивать модуль.



**Настройка официальной среды разработки ESP-IDF (Espressif IoT Development Framework)**

Среду ESP-IDF можно установить на Windows. Для этого нужно:

* Скачать с официального сайта среду разработки;
* Распаковать архив, поместить его на диск C;
* Затем появится каталог msys32, в который нужно перейти и запустить программу mingw32.exe;
* Появится окно терминала, где нужно создать каталог с именем esp;
* Нужно перейти в созданный каталог;
* Подключить необходимые библиотеки с помощью команды git clone –recursive <https://github.com/espressif/esp-idf.git>;
* Затем устанавливаются пакеты Python командой python -m pip install –user -r $IDF\_PATH/requirements.txt.

После этого можно начинать работу. В каталоге \esp-idf\examples\ есть различные примеры скетчей для работы с модулем.

# **Лк 8. Сервоприводы**

Сервопривод – это такой вид привода, который может точно управлять параметрами движения. Другими словами, это двигатель, который может повернуть свой вал на определенный угол или поддерживать непрерывное вращение с точным периодом.

В проектах ардуино робототехники серво часто используется для простейших механических действий:

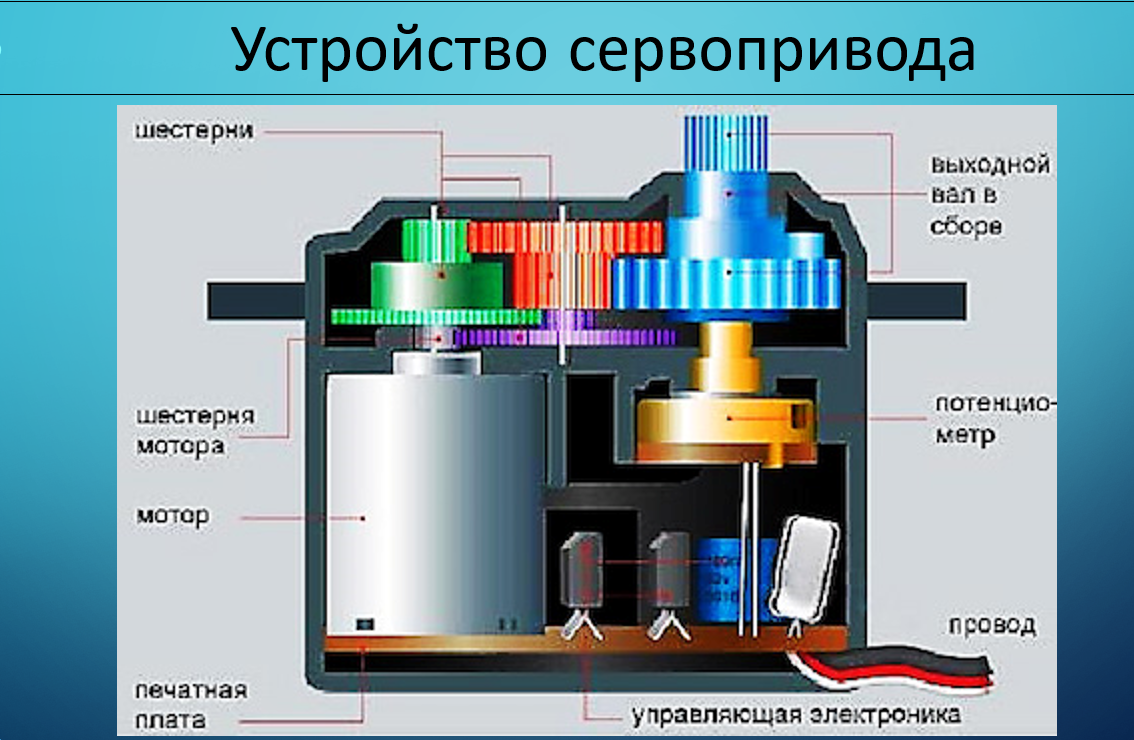
Повернуть дальномер или другие датчики на определенный угол, чтобы измерить расстояние в узком секторе обзора робота.

Сделать небольшой шаг ногой, движение конечностью или головой.

Для создания роботов-манипуляторов.

Для реализации механизма рулевого управления.

Открыть или закрыть дверку, заслонку или другой предмет.  
Основной задачей таких устройств является реализация в области сервомеханизмов. Также сервоприводы нередко  используются в таких сферах как обработка материалов, производство транспортного оборудования, обработка древесины, изготовление металлических листов, производство стройматериалов и другие.



Вся схема управления серво находится внутри корпуса, управляющие сигналы и питание подаются, как правило, идут по трем проводам: земля, напряжение питания и управляющий сигнал.

Выделяют два основных вида серводвигателей – с **непрерывным** вращением и с **фиксированным** углом  (чаще всего, 180 или 270 градусов). Отличие серво ограниченного вращения заключается в механических элементах конструкции, которые могут блокировать движение вала вне заданных параметрами углов. Достигнув угла 180, вал окажет воздействие на ограничитель, а тот отдаст команду на выключение мотора. У серводвигателей непрерывного вращения таких ограничителей нет.

У большинства сервоприводов связующим звеном между валом и внешними элементами является шестеренка, поэтому очень важно, из какого материала она сделана. Наиболее доступных вариантов два: металлические или пластмассовые шестерни. В более дорогих моделях можно найти элементы из карбона и даже титана.

Широкое использование сервоприводов связано с тем, что они обладают стабильной работой, высокой устойчивостью к помехам, малыми габаритами и широким диапазоном контроля скорости. Важными особенностями сервоприводов являются способность увеличивать мощность и обеспечение обратной информационной связи. И этого следует, что при прямом направлении контур является передатчиком энергии, а при обратном – передатчиком информации, которая используется для улучшения точности управления

Включая или выключая обычный электрический двигатель, мы можем сформировать вращательное движение и заставить двигаться колеса или другие предметы, прикрепленные к валу. Движение это будет непрерывным, но для того, чтобы понять, на какой угол повернулся вал или сколько оборотов он сделал, потребуется устанавливать дополнительные внешние элементы: энкодеры. Сервопривод уже содержит все необходимое для получения информации о текущих параметрах вращения и может самостоятельно выключаться, когда вал повернется на необходимый угол.

Сервопривод обладает тремя контактами, которые окрашены в разные цвета. Коричневый провод ведет к земле, красный – к питанию +5В, провод оранжевого или желтого цвета – сигнальный. К Ардуино устройство подключается через макетную указанным на рисунке образом. Оранжевый провод (сигнальный) подключается к  цифровому пину, черный и красный – к земле и питанию соответственно.

Сервоприводы играют очень важную роль для многих проектов Ардуино, от робототехнических до систем умного дома. Все, что связано с движением,  традиционно требует особых знаний и создать полноценный правильно работающий привод – непростая задача. Но с помощью серводвигателей можно во многих случаях упростить задачу, поэтому серво постоянно используется

# **Лк 9. Устройство управления шаговым двигателем.**

Шаговый электродвигатель — это синхронный бесщёточный электродвигатель с несколькими обмотками, в котором ток, подаваемый в одну из обмоток статора, вызывает фиксацию ротора. Последовательная активация обмоток двигателя вызывает дискретные угловые перемещения (шаги) ротора.

**Описание:** Конструктивно шаговые электродвигатели состоят из статора, на котором расположены обмотки возбуждения, и ротора, выполненного из магнитомягкого или из магнитотвёрдого материала. Шаговые двигатели с магнитным ротором позволяют получать больший крутящий момент и обеспечивают фиксацию ротора при обесточенных обмотках.

Таким образом по конструкции ротора выделяют следующие разновидности шагового двигателя:

* с постоянными магнитами (ротор из магнитотвёрдого материала);
* реактивный (ротор из магнитомягкого материала);
* гибридный.

Гибридные двигатели сочетают в себе лучшие черты двигателей с переменным магнитным сопротивлением и двигателей с постоянными магнитами.

**Использование:** В машиностроении наибольшее распространение получили высокомоментные двухфазные гибридные шаговые электродвигатели с угловым перемещением 1,8°/шаг (200 шагов/оборот) или 0,9°/шаг (400 шаг/об). Точность выставления шага определяется качеством механической обработки ротора и статора электродвигателя.

Шаговые электродвигатели применяются в приводах машин и механизмов, работающих в старт-стопном режиме, или в приводах непрерывного движения, где управляющее воздействие задаётся последовательностью электрических импульсов, например, в станках с ЧПУ. В отличие от сервоприводов, шаговые приводы позволяют получать точное позиционирование без использования обратной связи от датчиков углового положения.

Шаговые двигатели с постоянными магнитами могут использоваться в качестве датчиков угла поворота благодаря возникновению ЭДС на обмотках при вращении ротора.

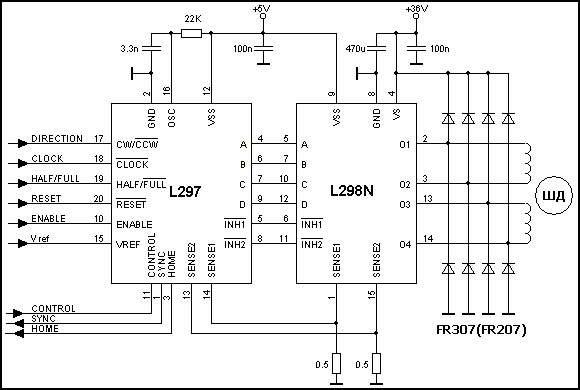
**Преимущества:** Главное преимущество шаговых приводов — точность. При подаче потенциалов на обмотки шаговый двигатель повернётся строго на определённый угол.

К приятным моментам можно отнести стоимость шаговых приводов, в среднем в 1,5-2 раза дешевле сервоприводов. Шаговый привод, как недорогая альтернатива сервоприводу, наилучшим образом подходит для автоматизации отдельных узлов и систем, где не требуется высокая динамика.

**Недостатки:** Возможность «проскальзывания» ротора — наиболее известная проблема этих двигателей. Это может произойти при превышении нагрузки на валу, при неверной настройке управляющей программы (например, ускорение старта или торможения не адекватно перемещаемой массе), при приближении скорости вращения к резонансной. Наличие датчика позволяет обнаружить проблему, но автоматически скомпенсировать её без остановки производственной программы возможно только в очень редких случаях. Чтобы избежать проскальзывания ротора, как один из способов, можно увеличить мощность двигателя.

****

Типовая схема управления шаговым двигателем с помощью комплекта микросхем L297 и L298N:



Технические характеристики контроллера шаговых двигателей на L297 и L298:



# ESP8266

Микросхема ESP8266 – один из самых популярных инструментов для организации беспроводной связи в проектах умного дома. С помощью беспроводного контроллера можно организовывать связь по интерфейсу WiFi, обеспечивая проектам Arduino выход в интернет и возможность дистанционного управления и сбора данных. На основе ESP8266 созданы такие популярные платы как WeMos и NodeMcu, а также огромное количество самодельных проектов.

**Описание ESP8266**

ESP8266 – микроконтроллер с интерфейсом WiFi, который имеет возможность исполнять программы из флеш-памяти.

Контроллер недорогой, обладает небольшим количеством внешних элементов и имеет следующие технические параметры:

* Поддерживает Wi-Fi протоколы 802.11 b/g/n с WEP, WPA, WPA2;
* Обладает 14 портами ввода и вывода, SPI, I2C, UART, 10-бит АЦП;
* Поддерживает внешнюю память до 16 МБ;
* Необходимое питание от 2,2 до 3,6 В, потребляемый ток до 300 мА в зависимости от выбранного режима.

Важной особенностью является отсутствие пользовательской энергонезависимой памяти на кристалле. Программа выполняется от внешней SPI ПЗУ при помощи динамической загрузки необходимых элементов программы. Доступ к внутренней периферии можно получить не из документации, а из API набора библиотек. Производителем указывается приблизительное количество ОЗУ – 50 кБ.

**Особенности платы ESP8266**

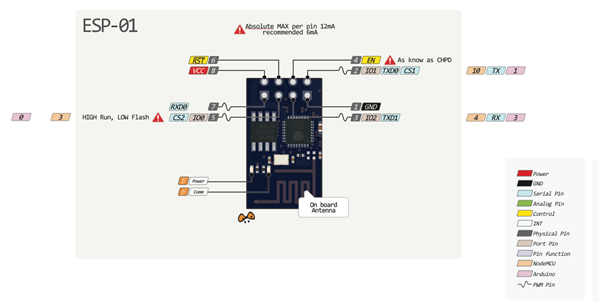
* Удобное подключение к компьютеру – через USB кабель, питание от него же;
* Наличие встроенного преобразователя напряжения 3,3В;
* Наличие 4 Мб флеш-памяти;
* Встроенные кнопки для перезагрузки и перепрошивки;
* Все порты выведены на плату на две гребенки с шагом 2,5 мм.

**Сферы применения модуля ESP8266**

* Автоматизация;
* Различные системы для умного дома: Беспроводное управление, беспроводные розетки, управление температурой, дополнение к сигнализационным системам;
* Мобильная электроника;
* ID метки;
* Детские игрушки;
* Mesh-сети.

**Распиновка esp8266**

Исполнение программы требуется задавать состоянием портов GPIO0, GPIO2 и GPIO15, когда заканчивается подача питания. Можно выделить 2 важных режима – когда код исполняется из UART (GPIO0 = 0, GPIO2 = 1 и GPIO15 = 0) для перепрошивки флеш-карты и когда исполняется из внешней ПЗУ (GPIO0 = 1, GPIO2 = 1 и GPIO15 = 0) в штатном режиме.

Распиновка для ESP01 изображена на картинке.  
  
**Описание контактов:**

* 1 – земля, 8 – питание. По документации напряжение подается до 3,6 В – это важно учесть при работе с Ардуино, на которую обычно подают 5 В.
* 6 – RST, нужна для перезагрузки микроконтроллера при подаче на него низкого логического уровня.
* 4 – CP\_PD, также используется для перевода устройства в энергосберегающий режим.
* 7 и 0 – RXD0 и TXD0, это аппаратный UART, необходимый для перепрошивки модуля.
* 2 – TXD0, к этому контакту подключается светодиод, который загорается при низком логическом уровне на GPIO1 и при передаче данных по UART.
* 5 – GPIO0, порт ввода и вывода, также позволяет перевести устройство в режим программирования (при подключении порта к низкому логическому уровню и подачи напряжения).
* 3 – GPIO2, порт ввода и вывода.

**Основные отличия Ардуино от ESP8266**

* ESP8266 имеет больший объем флеш-памяти, при этом у ESP8266 отсутствует энергонезависимая память;
* Процессор ESP8266 быстрее, чем у Ардуино;
* Наличие Wi-Fi у ESP8266;
* ESP8266 потребляеn больше тока, чем для Ардуино;

**Программирование ESP8266 в Arduino IDE**

Программный комплект разработчика esp8266 включает в себя:

* Компилятор из пакета GNU Compiler Collection.
* Библиотеки, стеки протоколов WiFi, TCP/IP.
* Средство загрузки информации в программу контроллера.
* Операционная IDE.

Изначально модули ESP8266 поставляются с прошивкой от фирмы-изготовителя. С ее помощью можно управлять модулем с внешнего микроконтроллера, реализовывать работу с Wi-Fi как с модемом. Также существует множество других готовых прошивок. Некоторые из них позволяют настраивать работу модуля при помощи WEB-интерфейса.

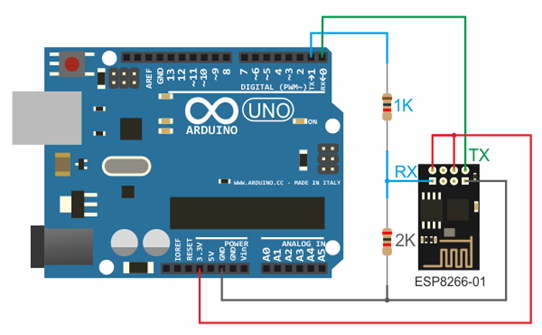
Можно программировать из среды Arduino IDE. При ее помощи можно легко писать скетчи и загружать их в ESP8266, прошивать ESP8266, при этом не требуется сама плата Ардуино. Arduino IDE поддерживает все виды модулей ESP8266.

**В настоящий момент для ESP8266 можно реализовать следующие функции:**

* Основные функции языка Wiring. Управлять портами GPIO можно точно так же, как и пинами на плате Ардуино: pinMode, digitalRead, digitalWrite, analogWrite. Команда analogRead(А0) позволяет считать значения АЦП. При помощи команды analogWrite (pin, value) можно подключить ШИМ на нужном выходе GPIO. При value=0 ШИМ отключается, максимальное значение достигает константы, равной 1023.С помощью функций attachInterrupt, detachInterrupt можно выполнять прерывание на любом порте GPIO, кроме 16.
* Тайминг и delay. Используя команды millis и micros можно вернуть мс и мкс, которые прошли с момента старта. Delay позволяет приостановить исполнение программы на нужное время. Также функция delay(…) позволяет поддерживать нормальную работу Wi-Fi, если в скетче присутствуют большие элементы, которые выполняются более 50 мс. Yield() – аналог функции delay(0).
* Serial и Serial1 (UART0 и UART1). Работа Serial на ESP8266 аналогична работе на ардуино. Запись и чтение данных блокируют исполнение кода, если FIFO на 128 байт и программный буфер на 256 байт заполнены. Объект Serial пользуется аппаратным UART0, для него можно задать пины GPIO15 (TX) и GPIO13 (RX) вместо GPIO1(TX) и GPIO3(RX). Для этого после функции Serial.begin(); нужно вызвать Serial.swap();. Аналогично Serial1 использует UART1, который работает на передачу. Необходимый пин для этого GPIO2.
* Макрос PROGMEM. Его работа аналогична работе в Ардуино. Позволяет перемещать данные read only и строковые постоянные во flash-память. При этом в ESP8266 не сохраняются одинаковые константы, что приводит к дополнительной трате флеш-памяти.
* I2C. Перед началом работы с шиной I2C выбираются шины с помощью функции Wire.pins(int sda, int scl).
* SPI, OneWire – поддерживаются полностью.

**Использование esp8266 для связи Ардуино по WiFi**

Перед подключением к Ардуино важно помнить, что у ESP8266 напряжение питания не может быть выше 3,6, в то время как на пате Ардуино напряжение равно 5 В. Соединять 2 микроконтроллера нужно с помощью резистивных делителей. Перед подключением модуля нужно ознакомиться с распиновкой выбранного ESP8266.  Схема подключения для ESP8266-01 представлена на рисунке.



3,3 В с Ардуино – на Vcc&CH\_PD на модуле ESP8266, GND с Ардуино – к GND с ESP8266, 0 – TX, 1 – RX.

Для поддержки стабильной работы ESP8266 необходим источник постоянного напряжения на 3,3 В и максимальный ток 250 мА. Если питание происходит от конвертера USB-TTL, могут происходить неполадки и сбои в работе.

Работа с библиотекой Wi-Fi для ESP8266 схожа с библиотекой для обыкновенного шилда. **Имеется несколько особенностей**:

* mode(m) – для выбора одного из трех режимов: клиент, точка доступа или оба режима единовременно.
* softAP(ssid) – нужен для создания открытой точки доступа.
* softAP(ssid, password) – создает точку доступа с паролем, который должен состоять не менее чем из 8 знаков.
* WiFi.macAddress(mac) и WiFi.softAPmacAddress(mac)– определяет МАС адрес.
* WiFi.localIP() и WiFi.softAPIP() – определение IP адреса.
* printDiag(Serial); – позволят узнать данные о диагностике.
* WiFiUDP – поддержка передачи и приема multicast пакета в режиме клиента.

**Работа выполняется по следующему алгоритму:**

* Подключение USB-TTL к USB и к ESP.
* Запуск Arduino IDE.
* Выбрать в меню инструменты нужный порт, плату, частоту и размер flash-памяти.
* Файл — Примеры — ESP8266WiFi — WiFiWebServer.
* Записать в скетче SSID и пароль сети Wi-Fi.
* Начать компиляцию и загрузку кода.
* Дождаться окончания процесса прошивки, отсоединить GPIO0 от земли.
* Поставить скорость 115200.
* Произойдет подключение, будет записан адрес IP.
* Открыть браузер, ввести в адресной строке номер IP/gpio/1
* Посмотреть монитор порта, если к выходу GPIO2 подключен светодиод, он должен загореться.

**NodeMCU на базе esp8266**

NodeMCU – это платформа, основанная на базе модуля esp8266. Используется для управления схемой на расстоянии при помощи интернета через Wi-Fi. Плата малогабаритная, компактная, стоит дешево, на лицевой стороне имеется разъем для USB. Рядом кнопки для отладки и перезагрузки микроконтроллера. Также установлен чип ESP8266. Напряжение питания – от 5 до 12 В, желательно подавать более 10 В.

Большим преимуществом платы является ее малое энергопотребление. Нередко их используют в схемах с автономным питанием. На плате расположены всего 11 портов общего назначения, из них некоторые имеют специальные функции:

* D1 и D2 – для интерфейса I2C/ TWI;
* D5-D8- для интерфейса SPI;
* D9, D10 – для UART;
* D1-D10 – могут работать как ШИМ.

Платформа имеет современное API для аппаратного ввода и вывода. Это позволяет сократить количество действий во время работы с оборудованием и при его настройке. С помощью прошивки NodeMCU можно задействовать весь рабочий потенциал для быстрой разработки устройства.

**WeMos на базе esp8266**

WeMos  – еще один вид платформы, основанный на базе микроконтроллера esp8266. Соответственно, имеется Wi-Fi модуль, поддерживается Arduino IDE, имеется разъем для внешней антенны. Плата имеет 11 цифровых входов/выходов, которые (кроме D0) поддерживают interrupt/pwm/I2C/one-wire. Максимальное напряжение питания достигает 3,3 В. Также на платформе присутствует USB разъем.  Аналоговый вход 1 с максимальным напряжением 3,2В.

Для работы с модулем нужно установить драйвер CH340 и настроить Ардуино IDE под ESP8266. Для этого нужно в меню настройки в строке «дополнительная ссылка для менеджера плат» добавить адрес http://arduino.esp8266.com/stable/package\_esp8266com\_index.json.

После этого требуется найти пакет esp8266 by ESP8266 и установить его.  Затем нужно выбрать в меню инструменты микроконтроллер Wemos D1 R2 и записать нужный скетч.

**Выводы по ESP8266**

С помощью плат на основе микросхемы ESP8266 вы можете добавить в свои проекты возможности “большого интернета”, сделав их гораздо более интеллектуальными. Дистанционное управление, сбор и анализ данных на сервере, обработка голоса и работа с изображением – все это становится доступным, когда мы подключаем наш проект по WiFi к интернету. В следующих статьях мы подробно рассмотрим то, как можно программировать устройства на базе esp8266, а также уделим внимание таким популярным платам как WeMos и NodeMcu.

# Гироскопы МЭМС (Микроэлектромеханические системы)

Гироскоп — это устройство, способное реагировать на изменение углов ориентации тела, на котором оно установлено, относительно инерциальной системы отсчета.

**Одноосевой МЭМС-датчик угловой скорости (гироскоп) с вибрирующим кремниевым кольцом**

Данный кремниевый цифровой гироскоп разработан с учетом требований к низкой стоимости изделия и экономичному энергопотреблению для систем навигации и наведения нового поколения. Он способен измерять угловую скорость до ± 1,0 рад/с и имеет два режима вывода: аналоговый сигнал напряжения, линейно-пропорциональный угловой скорости, и цифровой по протоколу SPI®.

Как правило, подобные гироскопы выпускаются в герметичных керамических LCC корпусах, которые можно устанавливать на системные платы. **Датчик состоит из пяти основных компонентов:**

* кремниевый кольцевой МЭМС-сенсор (MEMS-ring),
* основание из кремния (Pedestal),
* интегральная микросхема гироскопа (ASIC),
* корпус (Package Base),
* крышка (Lid).

**Гироскоп L3GD20**

* 3-осевой цифровой гироскоп, цифровой SPI/I2C интерфейс
* Высокое разрешение, 16 бит разрешение
* Измерение вращения по 3 шкалам: ±250°/с, ±500°/си ±2000°/с
* Power Down (5 мкА) и Sleep (2 мА) режимы
* Interruption и Data Ready выходные линии
* Высокая производительность:
* Иммунитет к аудио и механическому шуму
* Высокое разрешение/высокая температурная стабильность
* Высокая шоковая устойчивость: 10 000 g в течении о.1 мс
* Дополнительные параметры:
* 4 Output Data Rates(ODR): 95, 190, 380, 760 Гц
* 8-бит выход температурного датчика, FIFO – буфер
* Конфигурируемые фильтры низких и высокихчастот
* Функция самотестирования
* Совместимость по выводам и программная совместимость с L3G4200D

**Гироскоп L3G3250A**

* 3-Осевой Аналоговый Гироскоп
* Иммунитет к аналоговому шуму и вибрациям
* 2 шкалы измерения: ±625°/с и ±2500°/с
* Power down и Sleep режимы
* Функция самотестирования
* Заводская калибровка
* Высокая чувствительность: 2 мВ/°/с при 625°/с
* Встроенный фильтр нижних частот
* Высокая температурная стабильность (0.08°/с/°C)
* Высокое шоковое состояние: 10000g в течении 0.1 мс
* Температурный диапазон от -40 до 85°C
* Напряжение питания: 2.4 - 3.6В
* Потребление: 6.3 мA в Normal, 2 мA в Sleep и 5 мкA в Power Down режимах
* Корпус: 3.5 x 3 x 1 LGA

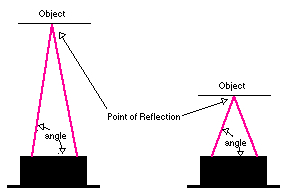
**A3G4250D: 3-осевой гироскоп для автомобильного рынка**

* A3G4250D-Automotive – 3 axes – Gyroscope –
* 4x4 LGA 16L– ±245dps full scale – Digital Output
* Совместим по выводам с L3G4200D
* Низкое потребление
* Низкий шум и высокая стабильность по температуре
* Области применения: трекинг-системы, автотрекеры, навигация, определение угла наклона
* AEC-Q100

# Виды датчиков

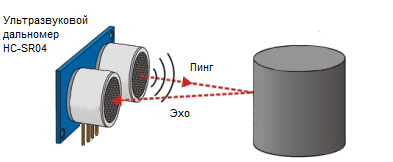
Для определения расстояния до какого-либо объекта используются различные датчики дальности. Одними из них являются инфракрасный датчик и ультразвуковой, которые широко используются на платформе Arduino.

**Принцип работы (инфракрасный)**



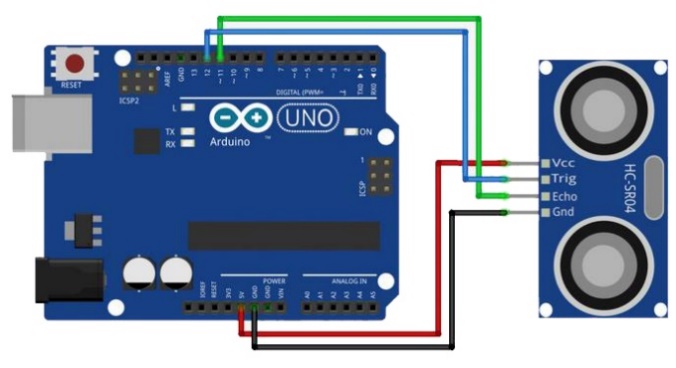
Датчик фиксирует отражение от объекта луча светодиода на удалении не более 0,8 м. “Угол зрения” можно представить в виде конуса с диаметром в средней части около 0,16 м. Излучение отражается под углом и возвращается на воспринимающий элемент датчика. Чем дальше объект, тем острее угол.

**Принцип работы (ультразвуковой)**



Ультразвуковой датчик HC-SR04, или, как его еще называют, сонар, подает звуковые импульсы (пинг) на частоте 40 кГц и ловит отраженный сигнал (эхо). По времени передачи импульса и его возврату определяется дальность.

**Схема подключения (ультразвуковой)**



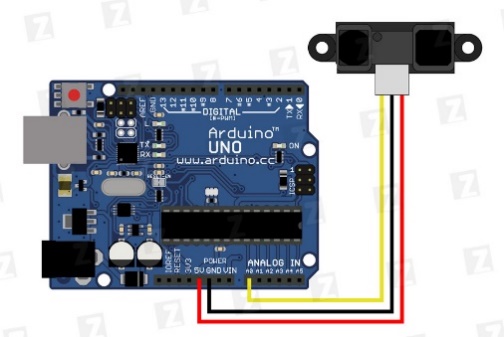
Vcc - питание 5В.

Trig - цифровой вход. Для запуска измерения необходимо подать на этот вход логическую единицу на 10 мкс. Следующее измерение рекомендуется выполнять не ранее чем через 50 мс.

Echo - цифровой выход. После завершения измерения, на этот выход будет подана логическая единица на время, пропорциональное расстоянию до объекта.

GND - земля.

**Схема подключения (инфракрасный)**

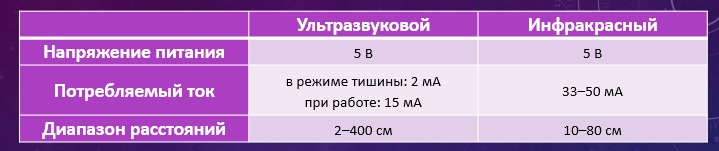


GND на любой из GND пинов­­­ ардуино.

OUT на любой из аналоговых входов.

VCC на + 5 вольт.

**Сравнение**

****

Оба датчика могут использоваться не только в качестве измерителей расстояния. Например, закрепив и подключив к платформе, они могут использоваться как «глаза» робота, которые не позволят ему слепо передвигаться, врезаясь во все подряд. Напротив, он сможет объезжать препятствия, служить сигнализацией, строить карту помещения. Так, с помощью датчиков можно сделать робота, который найдет выход из лабиринта. Каждый из датчиков имеет свои плюсы и минусы.

# Операционные системы реального времени (RTOS)

## Февраль, ЛК1. Понятие операционной системы

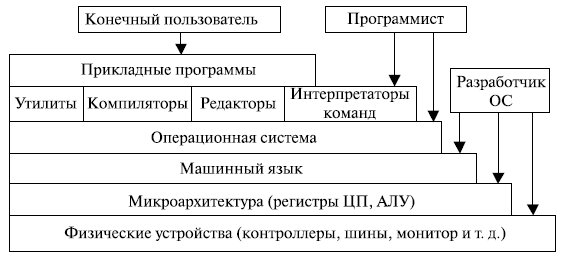
Операционная система была создана для того, чтобы автоматизировать работу оператора и скрыть от пользователя сложности общения с аппаратурой, предоставив ему более удобную систему команд.

Таким образом, операционная система — это программный интерфейс между пользователем и аппаратурой вычислительной машины.

С точки зрения пользователя ОС выполняет функцию расширенной машины или виртуальной машины, в которой легче программировать и легче работать, чем непосредственно с аппаратным обеспечением, составляющим реальный компьютер. Операционная система предоставляет простой, абстрагированный интерфейс, скрывая множество кропотливой работы с прерываниями, счетчиками времени, организацией памяти и другими компонентами низкого уровня. Это взгляд сверху вниз.

Взгляд снизу вверх дает представление об операционной системе как о менеджере, присутствующем в компьютере для управления всеми компонентами этой сложнейшей системы.

Так, работа операционной системы заключается в обеспечении организованного и контролируемого распределения процессоров, памяти, дисков, принтеров, устройств ввода-вывода, датчиков времени и т.п. между различными программами, конкурирующими за право их использовать.



Расположение операционной системы в иерархической структуре программного и аппаратного обеспечения компьютера.

Уточним первоначальное определение.

Операционная система – это набор программ, контролирующих работу прикладных программ и системных приложений и исполняющих роль интерфейса между пользователями, программистами, прикладными программами, системными приложениями и аппаратным обеспечением компьютера.

## ОС как расширенная виртуальная машина

Вместо реальной аппаратуры компьютера ОС представляет пользователю расширенную виртуальную машину, с которой удобнее работать и которую легче программировать. Вот список основных сервисов, предоставляемых типичными операционными системами.

1. **Разработка программ**: ОС представляет программисту разнообразные инструменты разработки приложений: редакторы, отладчики и т.п. Ему не обязательно знать, как функционируют различные электронные и электромеханические узлы и устройства компьютера. Часто пользователь не знает даже системы команд процессора, поскольку он может обойтись мощными высокоуровневыми функциями, которые представляет ОС.
2. **Исполнение программ**. Для запуска программы нужно выполнить ряд действий: загрузить в основную память программу и данные, инициализировать устройства ввода-вывода и файлы, подготовить другие ресурсы. ОС выполняет всю эту рутинную работу вместо пользователя.
3. **Доступ к устройствам ввода-вывода**. Для управления каждым устройством используется свой набор команд. ОС предоставляет пользователю единообразный интерфейс, который скрывает все эти детали и обеспечивает программисту доступ к устройствам ввода-вывода с помощью простых команд чтения и записи. Если бы программист работал непосредственно с аппаратурой компьютера, то для организации, например, чтения блока данных с диска ему пришлось бы использовать более десятка команд с указанием множества параметров. После завершения обмена программист должен был бы предусмотреть еще более сложный анализ результата выполненной операции.
4. **Контролируемый доступ к файлам**. При работе с файлами управление со стороны ОС предполагает не только глубокий учет природы устройства ввода-вывода, но и знание структур данных, записанных в файлах. Многопользовательские ОС, кроме того, обеспечивают механизм защиты при обращении к файлам.
5. **Системный доступ**. ОС управляет доступом к совместно используемой или общедоступной вычислительной системе в целом, а также к отдельным системным ресурсам. Она обеспечивает защиту ресурсов и данных от несанкционированного использования и разрешает конфликтные ситуации.
6. **Обнаружение ошибок и их обработка**. При работе компьютерной системы могут происходить разнообразные сбои за счет внутренних и внешних ошибок в аппаратном обеспечении, различного рода программных ошибок (переполнение, попытка обращения к ячейке памяти, доступ к которой запрещен и др.). В каждом случае ОС выполняет действия, минимизирующие влияние ошибки на работу приложения (от простого сообщения об ошибке до аварийной остановки программы).
7. **Учет использования ресурсов**. Хорошая ОС имеет средства учета использования различных ресурсов и отображения параметров производительности вычислительной системы. Эта информация важна для настройки (оптимизации) вычислительной системы с целью повышения ее производительности

Таким образом, назначение ОС состоит в предоставлении пользователю (программисту) некоторой расширенной виртуальной машины, которую легче программировать и с которой легче работать, чем непосредственно с аппаратурой.

## ОС как менеджер ресурсов

ОС не только представляет пользователям и программистам удобный интерфейс к аппаратным средствам компьютера, но и является своеобразным диспетчером ресурсов компьютера. К числу основных ресурсов современных вычислительных систем относятся процессоры, основная память, таймеры, наборы данных, диски, накопители на магнитных лентах (МЛ), внешние накопители памяти (CD/DVD/Blu-Ray/USB), принтеры, сетевые устройства и др.

Управление ресурсами может происходить:

* во времени (например, выделение процессорного времени);
* в пространстве (например, запись файла на жёсткий диск).

Управление ресурсами включает решение ряда общих, не зависящих от типа ресурса задач:

* планирование ресурса – определение, какому процессу, когда и в каком качестве (если ресурс может выделяться частями) следует выделить данный ресурс;
* удовлетворение запросов на ресурсы – выделение ресурса процессам;
* отслеживание состояния и учет использования ресурса – поддержание оперативной информации о занятости ресурса и распределенной его доли;
* разрешение конфликтов между процессами, претендующими на один и тот же ресурс.

В отличие от функций расширенной виртуальной машины большинство функций управления ресурсами выполняются операционной системой автоматически и прикладному программисту недоступны.

ОС облегчает процессы эксплуатации аппаратных и программных средств вычислительной системы. Ряд операционных систем имеет в своем составе наборы служебных программ, обеспечивающие резервное копирование, архивацию данных, проверку, очистку и дефрагментацию дисковых устройств и др.

Кроме того, современные ОС имеют достаточно большой набор средств и способов диагностики и восстановления работоспособности системы. Сюда относятся:

* диагностические программы для выявления ошибок в конфигурации ОС;
* средства восстановления последней работоспособной конфигурации;
* средства восстановления поврежденных и пропавших системных файлов и др.

Наиболее важными подсистемами управления ресурсами являются подсистемы управления процессами, памятью, файлами и внешними устройствами, а подсистемами, общими для всех ресурсов, являются подсистемы пользовательского интерфейса, защиты данных и администрирования.

**Управление процессами**. Подсистема управления процессами непосредственно влияет на функционирование вычислительной системы. Для каждой выполняемой программы ОС организует один или более процессов. Каждый такой процесс представляется в ОС информационной структурой (таблицей, дескриптором, контекстом процессора), содержащей данные о потребностях процесса в ресурсах, а также о фактически выделенных ему ресурсах (область оперативной памяти, количество процессорного времени, файлы, устройства ввода-вывода и др.). Кроме того, в этой информационной структуре хранятся данные, характеризующие историю пребывания процесса в системе: текущее состояние (активное или заблокированное), приоритет, состояние регистров, программного счетчика и др.

**Управление памятью**. Подсистема управления памятью производит распределение физической памяти между всеми существующими в системе процессами, загрузку и удаление программных кодов и данных процессов в отведенные им области памяти, настройку адресно-зависимых частей кодов процесса на физические адреса выделенной области, а также защиту областей памяти каждого процесса. Стратегия управления памятью складывается из стратегий выборки, размещения и замещения блока программы или данных в основной памяти.

Одним из наиболее популярных способов управления памятью в современных ОС является виртуальная память.

Важная функция управления памятью – защита памяти. Нарушения защиты памяти связаны с обращениями процессов к участкам памяти, выделенной другим процессам прикладных программ или программ самой ОС. Средства защиты памяти должны пресекать такие попытки доступа.

**Управление файлами**. Функции управления файлами сосредоточены в файловой системе ОС. Операционная система виртуализирует отдельный набор данных, хранящихся на внешнем накопителе, в виде файла – простой неструктурированной последовательности байтов, имеющих символьное имя. Для удобства работы с данными файлы группируются в каталоги, которые, в свою очередь, образуют группы – каталоги более высокого уровня. Файловая система преобразует символьные имена файлов, с которыми работает пользователь или программист, в физические адреса данных на дисках, организует совместный доступ к файлам, защищает их от несанкционированного доступа.

**Управление внешними устройствами**. Функции управления внешними устройствами возлагаются на подсистему управления внешними устройствами, называемую также подсистемой ввода-вывода. Она является интерфейсом между ядром компьютера и всеми подключенными к нему устройствами. Спектр этих устройств очень обширен (принтеры, сканеры, мониторы, модемы, манипуляторы, сетевые адаптеры, АЦП разного рода и др.), сотни моделей этих устройств отличаются набором и последовательностью команд, используемых для обмена информацией с процессором и другими деталями.

Программа, управляющая конкретной моделью внешнего устройства и учитывающая все его особенности, называется драйвером. Созданием драйверов занимаются как разработчики ОС, так и компании, выпускающие внешние устройства. ОС должна поддерживать четко определенный интерфейс между драйверами и остальными частями ОС.

**Защита данных и администрирование**. Безопасность данных вычислительной системы обеспечивается средствами отказоустойчивости ОС, направленными на защиту от сбоев и отказов аппаратуры и ошибок программного обеспечения, а также средствами защиты от несанкционированного доступа. Для каждого пользователя системы обязательна процедура логического входа, в процессе которой ОС убеждается, что в систему входит пользователь, разрешенный административной службой. Администратор вычислительной системы определяет и ограничивает возможности пользователей в выполнении тех или иных действий.

Важным средством защиты являются функции аудита ОС, заключающегося в фиксации всех событий, от которых зависит безопасность системы.

Поддержка отказоустойчивости вычислительной системы реализуется на основе резервирования (дисковые RAID-массивы, резервное копирование и пр.)

**Интерфейс прикладного программирования.** Прикладные программисты используют в своих приложениях обращения к операционной системе, когда для выполнения тех или иных действий им требуется особый статус, которым обладает только ОС. Возможности операционной системы доступны программисту в виде набора функций, который называется интерфейсом прикладного программирования (Application Programming Interface, API). Приложения обращаются к функциям API с помощью системных вызовов. Способ, которым приложение получает услуги операционной системы, очень похож на вызов подпрограмм.

Способ реализации системных вызовов зависит от структурной организации ОС, особенностей аппаратной платформы и языка программирования.

**Пользовательский интерфейс**. ОС обеспечивает удобный интерфейс не только для прикладных программ, но и для пользователя (программиста, администратора).

Современные ОС поддерживают развитые функции пользовательского интерфейса для интерактивной работы за терминалами двух типов: алфавитно-цифрового и графического.

При работе за алфавитно-цифровым терминалом пользователь имеет в своем распоряжении систему команд, развитость которой отражает функциональные возможности данной ОС. Обычно командный язык ОС позволяет запускать и останавливать приложения, выполнять различные операции с каталогами и файлами, получать информацию о состоянии ОС, администрировать систему. Команды могут вводиться не только в интерактивном режиме с терминала, но и считываться из так называемого командного файла, содержащего некоторую последовательность команд.

Программный модуль ОС, ответственный за чтение отдельных команд или же последовательности команд из командного файла, называют командным интерпретатором.

## Особенности ОС реального времени

Существует несколько определений систем реального времени (СРВ) (real time operating systems (RTOS)), большинство из которых противоречит друг другу.

Например, системой реального времени называется система, в которой успешность работы любой программы зависит не только от ее логической правильности, но и от времени, за которое она получила результат. Если временные ограничения не удовлетворены, то фиксируется сбой в работе систем.

Примером задачи, где требуется СРВ, является управление роботом, берущим деталь с ленты конвейера.

Различают сильное (hard) и слабое (soft) требование реального времени. Если запаздывание программы приводит к полному нарушению работы управляемой системы, то говорят о сильном реальном времени (жесткие СРВ). Если же запаздывание ведет только к потере производительности, то говорят о слабом реальном времени (мягкие СРВ).

В СРВ необходимо введение некоторого директивного срока (в англоязычной литературе – deadline), до истечения которого задача должна обязательно (для систем мягкого реального времени – желательно) выполниться.

**Признаки систем жесткого реального времени:**

* недопустимость никаких задержек, ни при каких условиях;
* бесполезность результатов при опоздании;
* катастрофа при задержке реакции;
* цена опоздания бесконечно велика.

Пример системы жесткого реального времени — бортовая система управления самолетом.

**Признаки систем мягкого реального времени:**

* за опоздание результатов приходится платить;
* снижение производительности системы, вызванное запаздыванием реакции на происходящие события.

Пример — автомат розничной торговли или подсистема сетевого интерфейса.

Не существует операционных систем жесткого или мягкого реального времени. Понятия системы реального времени и операционной системы реального времени (ОСРВ) часто смешиваются.

**Система реального времени** – это конкретная система, связанная с реальным объектом. Она включает в себя необходимые аппаратные средства, операционную систему и прикладное программное обеспечение.

**Операционная система реального времени** – это только инструмент, помогающий построить конкретную систему реального времени. Поэтому бессмысленно говорить об операционных системах жесткого или мягкого реального времени.

Для создания системы жесткого реального времени необходимо сочетание подходящих аппаратных средств, адекватной операционной системы и правильного проектирования прикладного программного обеспечения.

Если, например, принято решение построить систему реального времени, обслуживающую TCP/IP-соединение через Ethernet, то система никогда не будет системой жесткого реального времени, поскольку сам Ethernet непредсказуем. В данном случае, основное ограничение на создание СРВ оказывает метод случайного доступа CSMA/CD.

Если, с другой стороны, вы создаете приложение над такой ОС, как "Windows 3.11", то ваша система никогда не будет системой жесткого реального времени, поскольку непредсказуемо поведение операционной системы.

Стандарт POSIX 1003.1 определяет СРВ следующим образом: «Реальное время в операционных системах — это способность операционной системы обеспечить требуемый уровень сервиса в заданный промежуток времени».

Иногда системами реального времени называют системы постоянной готовности (on-line системы), или «интерактивные системы с достаточным временем реакции». Обычно это делают фирмы-производители по маркетинговым соображениям. Если интерактивную программу называют работающей в реальном времени, то это означает, что она успевает обрабатывать запросы от человека, для которого задержка в сотни миллисекунд незаметна.

Часто понятие «система реального времени» отождествляют с понятием «быстрая система». Это не всегда правильно.

Согласно определению, СРВ должна «обеспечить требуемый уровень сервиса в заданный промежуток времени». Этот промежуток времени обычно задается периодичностью и скоростью процессов, которыми управляет система.

Типичные времена реакции на внешние события в процессах, управляемых СРВ:

* радиолокация – несколько миллисекунд;
* складской учет – несколько секунд;
* торговые операции – несколько минут;
* управление производством – несколько минут.

Интервал между поступлениями сообщений в ЭВМ может быть случайным и определяться внешними факторами, такими, как нажатие клавиши оператором, или он может быть циклическим и управляться от часов или от сканирующего механизма в ЭВМ.

**Определение**. Система реального времени реагирует в предсказуемое время на непредсказуемое появление внешних событий.

Это определение предъявляет к системе вполне определенные базовые требования.

Мартин Тиммерман сформулировал следующие необходимые требования для ОСРВ:

* ОС должна быть многозадачной и допускающей вытеснение (preemptable),
* ОС должна обладать понятием приоритета для потоков,
* ОС должна поддерживать предсказуемые механизмы синхронизации,
* ОС должна обеспечивать механизм наследования приоритетов,
* поведение ОС должно быть известным и предсказуемым (задержки обработки прерываний, задержки переключения задач, задержки драйверов и т.д.); это значит, что во всех сценариях рабочей нагрузки системы должно быть определено максимальное время отклика.

**Требование 1**: ОСРВ должна быть многонитиевой или многозадачной и поддерживать диспетчеризацию с вытеснением.

Поведение ОСРВ должно быть предсказуемым. Это не означает, что ОСРВ должна быть быстрой, но означает, что максимальный промежуток времени для выполнения любой операции должен быть известен заранее и должен быть согласован с требованиями приложения.

Планировщик должен иметь возможность вытеснять любую нить (задачу) и передавать управление той нити (задаче), которая больше всего в этом нуждается.

**Требование 2:** Должно существовать понятие приоритета нити (задачи).

При построении конкретной системы реального времени разработчик должен выстроить приоритеты задач таким образом, чтобы каждая из них успела с реакцией к своему критическому сроку, то есть он должен трансформировать базовое требование реального времени "успеть с реакцией к нужному моменту" в комбинацию приоритетов и в сценарий их динамического изменения.

Очевидно, что при этой трансформации возможны ошибки, приводящие к неправильной работе системы.

Как бы то ни было, во всех современных ОСРВ приходится использовать механизм приоритетов как один из инструментов предсказуемости поведения системы. На сегодня не имеется другого решения.

**Требование 3:** ОС должна поддерживать предсказуемые механизмы синхронизации нитей (задач). Все нити (задачи) разделяют данные (ресурсы) и должны обмениваться между собой информацией, поэтому необходимы механизмы межзадачного (межнитиевого) взаимодействия.

Синхронизационные механизмы подразделяют на следующие основные категории:

* универсальные для низкого уровня — можно использовать различными способами (семафоры);
* простые для низкого уровня — каждый из которых приспособлен к решению только одной задачи (мьютексы и условные переменные);
* универсальные для высокого уровня, выраженные через простые — к этой группе относится концепция монитора, которая может быть выражена через мьютексы и условные переменные;
* простые для высокого уровня — приспособленные к решению конкретной синхронизационной задачи (блокировка чтения-записи и барьеры).

Синхронизационные механизмы обеспечивают разделение **критических ресурсов**.

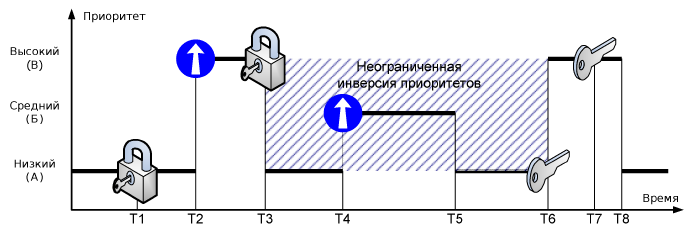
Те места в программах, в которых происходит обращение к критическим ресурсам, называются **критическими секциями**. Решение этой проблемы заключается в организации такого доступа к критическому ресурсу, когда только одному процессу разрешается входить в критическую секцию.

Ресурсы, которые не допускают одновременного использования несколькими процессами, называются **критическими**. Если нескольким вычислительным ресурсам необходимо пользоваться критическим ресурсом в режиме разделения, им следует синхронизировать свои действия таким образом, чтобы ресурс всегда находился в распоряжении не более чем одного из процессов.

**Требование 4:** Должен существовать механизм наследования приоритетов (система должна быть защищена от инверсии приоритетов).

Инверсия приоритета может произойти, когда многие потоки с различным приоритетом имеют общий доступ к одним и тем же блокировкам и ресурсам, и поток с низким приоритетом практически до бесконечности задерживает работу потока с более высоким приоритетом.

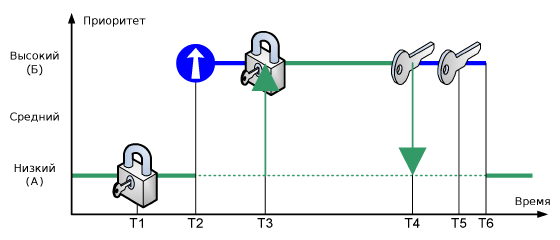
Если ресурс заблокирован задачей (А), а он требуется задаче (В), то высокоприоритетная задача блокируется. Но допустим, что задача (Б) вытеснила (А), после того как (В) ушла в ожидание ресурса. Задача (Б) ничего не знает о конфликте, поэтому может выполняться сколь угодно долго на промежутке времени (T5-T4). Кроме того, кроме (Б) в системе могут быть и другие задачи, с приоритетами больше (А), но меньше (Б). Поэтому длительность периода (T6-T3) в общем случае не определена. Такую ситуацию называют **неограниченной инверсией приоритетов**.



ОСРВ должна быть снабжена механизмом **наследования приоритетов**, то есть блокирующая нить должна наследовать приоритет нити, которую она блокирует (конечно, только, в том случае, если заблокированная нить имеет более высокий приоритет). То есть поведение ОС должно быть предсказуемо.

Допустим в системе существуют две задачи с низким (А) и высоким (Б) приоритетом:

В момент T2 задача (Б) вытесняет низкоприоритетную задачу (А) и затем в момент времени T3 пытается захватить заблокированный (А) ресурс. Протокол наследования приоритета состоит в том, что приоритет задачи (А) повышается до приоритета задачи (Б) в момент времени T3, то есть когда (Б) пытается захватить заблокированный ресурс.

[](http://www.pic24.ru/lib/exe/detail.php/tnkernel/ref/mutex/mutex_pr_inv_3.png?id=tnkernel:ref:mutex:intro)

В качестве основного требования к ОСРВ выдвигается требование обеспечения **предсказуемости** или **детерминированности поведения** системы в наихудших внешних условиях, что резко отличается от требований к производительности и быстродействию универсальных ОС. Хорошая ОСРВ имеет предсказуемое поведение при всех сценариях системной загрузки (одновременные прерывания и выполнение потоков).

Существует различие между **системами реального времени** и встроенными системами. От встроенной системы не всегда требуется, чтобы она имела предсказуемое поведение, и в таком случае она не является системой реального времени.

Однако большинство встроенных систем нуждается в предсказуемом поведении, по крайней мере, для некоторой функциональности, и таким образом, эти системы можно отнести к системам реального времени.

**Требования, накладываемые на вычислительную установку реального времени.**

1. В зависимости от сложности программы управления, требование «реального времени» накладывает различные условия на **вычислительную мощность** процессора для СРВ.
2. Часто для ОСРВ более важна не мощность процессора, а характеристики компьютера, связанные с **подсистемой прерываний** (interrupt requests, IRQ). Желательными являются:

* наличие как можно большего количества уровней прерываний (IRQ levels) (т.е. аппаратного или/и программного декодирования источника запроса);
* как можно меньшее время реакции на прерывание (т.е. как можно меньшее время между поступлением запроса на обслуживание и началом выполнения обслуживающей программы).

1. СРВ часто сама является инициатором периодических процессов, которыми управляет (например, движением космического аппарата или луча радара).

Поэтому необходимо иметь в наличии один или несколько таймеров (аппаратных устройств, выдающих прерывание через заданные промежутки времени), которые могут работать в периодическом или ждущем режиме.

1. Ввиду того, что СРВ часто управляет ответственными промышленными процессами, данное обстоятельство выдвигает очень жесткие требования к надежности используемого оборудования.

Понятие «системы реального времени» является новым и в полной мере не устоявшимся. Однако основной мыслью существующих определений является наложение жестких или мягких ограничений на **время отклика системы при поступлении на нее внешнего воздействия**.

Отсюда критически важным параметром для СРВ является время ее реакции на прерывания (которое складывается из аппаратного времени задержки и программных задержек), а также предсказуемость этого времени.

Нельзя считать системами реального времени информационно-управляющие системы, не предусматривающие в алгоритмах своего функционирования возможность выдачи откликов в условиях лимита времени. Поэтому организация работы информационно-управляющих систем, функционирующих в режиме реального времени, существенно отличается от работы традиционных систем управления.

## Области применения систем реального времени

В течение длительного времени основными потребителями СРВ были военная и космическая области. Сейчас ситуация кардинально изменилась и СРВ можно встретить даже в товарах широкого потребления. Рассмотрим основные области применения СРВ.

**Военная и космическая области:**

* бортовое и встраиваемое оборудование;
* системы измерения и управления, радары;
* цифровые видеосистемы, симуляторы;
* ракеты, системы определения положения и привязки к местности.

**Промышленность и финансы:**

* автоматические системы управления производством (АСУП), автоматические системы управления технологическим процессом (АСУТП);
* автомобилестроение: симуляторы, системы управления двигателем, автоматическое сцепление, системы антиблокировки колес и т.д.;
* энергетика: сбор информации, управление данными и оборудованием;
* телекоммуникации: коммуникационное оборудование, сетевые коммутаторы, телефонные станции и т.д.;
* банковское оборудование (например, во многих банкоматах работает СРВ QNX).

**Товары широкого потребления:**

* мобильные телефоны (например, в телефонах стандарта GSM работает СРВ pSOS);
* цифровые телевизионные декодеры;
* цифровое телевидение (мультимедиа, видеосерверы);
* компьютерное и офисное оборудование (принтеры, копиры), например, в факсах применяется СРВ VxWorks, в устройствах чтения компакт-дисков – СРВ VRTX32.

## Аппаратурная среда систем реального времени

Систему реального времени можно разделить как бы на три слоя:

1. Ядро — содержит только строгий минимум, необходимый для работы системы: управление задачами, их синхронизация и взаимодействие, управление памятью и устройствами ввода/вывода. Размер ядра очень ограничен: часто несколько килобайт.
2. Система управления — содержит ряд дополнительных сервисов, расширяющих возможности ядра: расширенное управление памятью, вводом/выводом, задачами, файлами и т.д., обеспечивает также взаимодействие системы и управляющего/управляемого оборудования.
3. Система реального времени — содержит систему управления и набор утилит: средства разработки (компиляторы, отладчики и т.д.), средства визуализации (взаимодействия человека и операционной системы).

Вычислительные установки, на которых применяются СРВ, можно условно разделить на три группы.

1. **«Обычные» компьютеры**. По логическому устройству совпадают с настольными системами. Аппаратное устройство несколько отличается. Для обеспечения минимального времени простоя в случае технической неполадки процессор, память и т.д. размещены на съемной плате.

Сам компьютер помещается в специальный защищённый корпус.

Среди процессоров этих компьютеров доминирует семейство Intel 80x86.

Подобные вычислительные системы обычно служат как терминалы для взаимодействия с промышленными компью¬терами и встроенными контроллерами. В качестве операционных систем часто используются «обычные» операционные системы с дополнительными программными комплексами, адаптирующими их к требованиям «реального времени».

1. **Промышленные компьютеры**. Состоят из одной платы, на которой размещены: процессор, контроллер памяти, память 4-х видов:

* ПЗУ, постоянное запоминающее устройство (ROM, read-only memory), где обычно размещена сама операционная система реального времени; типичная емкость – 0,5-1 Мб;
* ОЗУ, оперативное запоминающее устройство (RAM, random access memory), куда загружается код и данные ОСРВ; обычно организована на базе динамической памяти (dynamic RAM, DRAM); типичная емкость – 16-128 Mб;
* статическое ОЗУ (static RAM, SRAM) (то же, что и ОЗУ, но питается от имеющейся на плате батарейки), где размещаются критически важные данные, которые не должны пропадать при выключении питания; типичная емкость - 2Mб; типичное время сохранения данных - 5 лет;
* флеш-память (flash RAM) (электрически программируемое ПЗУ), которая играет роль диска для ОСРВ; типичная емкость - 4Mб.

Контроллеры периферийных устройств: SCSI (Small Computer System Interface), Ethernet, COM портов, параллельного порта, несколько программируемых таймеров. На плате находится также контроллер и разъем шины, через которую компьютер управляет внешними устройствами. В качестве шины в подавляющем большинстве случаев используется шина VME, которую в последнее время стала теснить шина Compact PCI.

Несмотря на наличие контроллера SCSI, обычно ОСРВ работает без дисковых накопителей, поскольку они не удовлетворяют предъявляемым к системам реального времени требованиям по надежности, устойчивости к вибрации, габаритам и времени готовности после включения питания.

В специальный корпус (крейт) вставляются платы аналого-цифровых и/или цифро-аналоговых преобразователей (АЦП и/или ЦАП), через которые осуществляется ввод/вывод управляющей информации, платы управления электромоторами.

Среди процессоров промышленных компьютеров доминируют процессоры семейств Power PC (Motorola IBM) и Motorola 68xxx (Motorola). Также присутствуют процессоры семейств SPARC (SUN), Intel 80x86 (Intel), ARM (ARM), Intel 80960x (Intel).

При выборе процессора определяющими факторами являются получение требуемой производительности при наименьшей тактовой частоте, а, значит, и наименьшей рассеиваемой мощности, а также наименьшее время переключения задач и реакции на прерывания.

Требование малой рассеиваемой мощности процессора важно с точки зрения получения высокой отказоустойчивости системы в целом, поскольку слабый нагрев процессора позволяет обойтись без охлаждающего вентилятора.

Промышленные компьютеры используются для непосредственного управления промышленным или иным оборудованием. Они часто не имеют монитора и клавиатуры, и для взаимодействия с ними служат «обычные» компьютеры, соединенные с ними через последовательный порт (COM порт) или Ethernet.

1. **Встраиваемые системы**. Устанавливаются внутрь оборудования, которым они управляют. Для крупного оборудования (например, ракета или космический аппарат) могут по исполнению совпадать с промышленными компьютерами.

Для оборудования поменьше (например, принтер) могут представлять собой процессор с сопутствующими элементами, размещенный на одной плате с другими электронными компонентами этого оборудования.

Для миниатюрного оборудования (например, мобильный телефон) процессор с сопутствующими элементами может быть частью одной из больших интегральных схем этого оборудования.

Система ОСРВ часто должна работать **на бездисковом компьютере** и осуществлять начальную загрузку из ПЗУ. В силу этого:

* критически важным является размер системы;
* для экономии места в ПЗУ часть системы может храниться в сжатом виде и загружаться в ОЗУ по мере необходимости;
* система часто позволяет исполнять код как в ОЗУ, так и в ПЗУ;
* при наличии свободного места в ОЗУ система часто копирует себя из медленного ПЗУ в более быстрое ОЗУ;
* сама система компилируется, линкуется и превращается в загрузочный модуль на другом, «обычном» компьютере, связанном с промышленным компьютером через последовательный порт или Ethernet. Это требует специального кроссплатформенного инструментария разработчика, поскольку типы процессоров и/или операционных систем на этих двух компьютерах не совпадают.

## Февраль, ЛК2. Стандарты ОСРВ

Большие различия в спецификациях ОСРВ и огромное количество существующих микроконтроллеров выдвигают на передний план **проблему стандартизации** в области систем реального времени.

Некоторые наиболее успешные компании в области систем реального времени объявляют о своем решении принять в качестве стандарта спецификации одной из своих продвинутых ОСРВ. Так поступила японская ассоциация TRON (the RTOS Nucleus), которая в 1987 г. выпустила в свет первые ITRON спецификации – ITRON1. Далее в 1989 г. она разработала и выпустила спецификации µITRON для 8- и 16- битовых микроконтроллеров, а также спецификации ITRON2 для 32-битовых процессоров. ОСРВ ITRON описывается ниже в соответствующем разделе. Этот стандарт является очень распространенным в Японии.

Наиболее ранним и распространенным стандартом ОСРВ является **стандарт POSIX** (IEEE Portable Operating System Interface for Computer Environments, IEEE 1003.1).

Первоначальный вариант стандарта POSIX появился в 1990 г. и был предназначен для UNIX-систем, первые версии которых появились в 70-х годах прошлого века.

Спецификации POSIX определяют стандартный механизм взаимодействия прикладной программы и операционной системы и в настоящее время включают набор более чем из 30 стандартов. Для ОСРВ наиболее важны семь из них (1003.1a, 1003.1b, 1003.1c, 1003.1d, 1003.1j, 1003.21, 1003.2h), но широкую поддержку в коммерческих ОС получили только три первых.

Хотя стандарт POSIX тесно связан с ОС Unix, тем не менее, разработчики многих ОСРВ стараются выдержать соответствие этому стандарту.

Спецификации POSIX задают стандартный механизм взаимодействия приложения и ОС.

Несмотря на то, что стандарт POSIX вырос из Unix’а, он затрагивает основополагающие абстракции операционных систем, а расширения реального времени применимы ко всем ОСРВ.

Соответствие стандарту POSIX для ОС и аппаратной платформы должно быть сертифицировано с помощью прогона на них тестовых наборов [POSIXTestSuite]. Однако, если ОС не является Unix-подобной, выдержать это требование становится непростой задачей. Тестовые наборы существуют только для POSIX 1003.1a.

К настоящему времени стандарт POSIX рассматривается как семейство родственных стандартов: IEEE Std 1003.n (где n – это номер).

**Стандарт 1003.1a** (OS Definition) содержит базовые интерфейсы ОС – поддержку единственного процесса, поддержку многих процессов, управление заданиями, сигналами, группами пользователей, файловой системой, файловыми атрибутами, управление файловыми устройствами, блокировками файлов, устройствами ввода/вывода, устройствами специального назначения, системными базами данных, каналами, очередями FIFO, а также поддержку языка C.

**Стандарт 1003.1b** (Realtime Extensions) содержит расширения реального времени – сигналы реального времени, планирование выполнения (с учетом приоритетов, циклическое планирование), таймеры, синхронный и асинхронный ввод/вывод, ввод/вывод с приоритетами, синхронизация файлов, блокировка памяти, разделяемая память, передача сообщений, семафоры.

Чтобы стать POSIX-комплиантной, ОС должна реализовать не менее 32 уровней приоритетов. POSIX определяет три политики планирования обработки процессов:

* SCHED\_FIFO – процессы обрабатываются в режиме FIFO и выполняются до завершения,
* SCHED\_RR – round robin – каждому процессу выделяется квант времени,
* SCHED\_OTHER – произвольная реализационно-зависимая политика, которая не переносима на другие платформы.

**Стандарт 1003.1c** (Threads) касается функций поддержки многопоточной обработки внутри процесса – управление потоками, планирование с учетом приоритетов, мьютексы (специальные синхронизирующие объекты в межпроцессном взаимодействии, подающие сигнал, когда они не захвачены каким-либо потоком), приоритетное наследование в мьютексах, переменные состояния (condition variables).

**Стандарт 1003.1d** включает поддержку дополнительных расширений реального времени – семантика порождения новых процессов (spawn), спорадическое серверное планирование, мониторинг процессов и потоков времени выполнения, тайм-ауты функций блокировки, управление устройствами и прерываниями.

**Стандарт 1003.21** касается распределенных систем реального времени и включает функции поддержки распределенного взаимодействия, организации буферизации данных, посылки управляющих блоков, синхронных и асинхронных операций, ограниченной блокировки, приоритетов сообщений, меток сообщений, и реализаций протоколов.

**Стандарт 1003.2h** касается сервисов, отвечающих за работоспособность системы.

**Cвойства открытости**

Область применимости стандартов POSIX OSE (Open System Environment) – обеспечение следующих возможностей (называемых еще свойствами открытости) для разрабатываемых информационных систем:

* Переносимость приложений на уровне исходных текстов (Application Portability at the Source Code Level), т.е. предоставление возможности переноса программ и данных, представленных на исходных текстах языков программирования, с одной платформы на другую.
* Системная интероперабельность (System Interoperability), т.е. поддержка взаимосвязанности между системами.
* Переносимость пользователей (User Portability), т.е. обеспечение возможности для пользователей работать на различных платформах без переобучения.
* Адаптируемость к новым стандартам (Accommodation of Standards), связанным с достижением целей открытости систем.
* Адаптируемость к новым информационным технологиям (Accommodation of new System Technology) на основе универсальности классификационной структуры сервисов и независимости модели от механизмов реализации.
* Масштабируемость прикладных платформ (Application Platform Scalability), отражающая возможность переноса и повторного использования прикладного программного обеспечения применительно к разным типам и конфигурациям прикладных платформ.
* Масштабируемость распределенных систем (Distributed System Scalability), отражающая возможность функционирования прикладного программного обеспечения независимо от развития топологии и ресурсов распределенных систем.
* Прозрачность реализаций (Implementation Transparency), т.е. сокрытие от пользователей за интерфейсами систем особенностей их реализации.
* Системность и точность спецификаций функциональных требований пользователей (User Functional Requirements), что обеспечивает полноту и ясность определения потребностей пользователей, в том числе в определении состава применяемых стандартов.

**Стандартизация POSIX позволяет решать следующие задачи:**

* интеграция информационных систем из компонент различных изготовителей;
* эффективность реализаций и разработок, благодаря точности спецификаций и соответствию стандартным решениям, отражающим передовой научно-технический уровень;
* эффективность переноса прикладного программного обеспечения, благодаря использованию стандартизованных интерфейсов и прозрачности механизмов реализации сервисов систем.

Военная и аэрокосмическая отрасли предъявляют жесткие требования к вычислительным средствам, влияющим на степень безопасности целевой системы. В настоящее время имеются следующие стандарты для ОСРВ в авиации – **стандарт DO-178B** и **стандарт ARINC-653**. Поскольку эти стандарты разработаны в США, стоит отметить еще европейский **стандарт ED-12B**, который является аналогом DO-178B.

Стандарт DO-178B, создан Радиотехнической комиссией по аэронавтике (RTCA, Radio Technical Commission for Aeronautics) для разработки ПО бортовых авиационных систем [DO178B].

Первая его версия была принята в 1982 г., вторая (DO-178A) - в 1985-м, текущая DO-178B - в 1992 г. Готовится принятие новой версии, DO-178C. Стандартом предусмотрено **пять уровней серьезности отказа**, и для каждого из них определен набор требований к программному обеспечению, которые должны гарантировать работоспособность всей системы в целом при возникновении отказов данного уровня серьезности.

Стандарт DO-178B определяет следующие уровни сертификации:

* А (катастрофический),
* В (опасный),
* С (существенный),
* D (несущественный)
* Е (не влияющий).

До тех пор, пока все жесткие требования этого стандарта не будут выполнены, вычислительные системы, влияющие на безопасность, никогда не поднимутся в воздух.

**Стандарт ARINC-653** (Avionics Application Software Standard Interface) разработан компанией ARINC в 1997 г. Этот стандарт определяет универсальный программный интерфейс APEX (Application/Executive) между ОС авиационного компьютера и прикладным ПО.

Требования к интерфейсу между прикладным ПО и сервисами операционной системы определяются таким образом, чтобы разрешить прикладному ПО контролировать диспетчеризацию, связь и состояние внутренних обрабатываемых элементов. В 2003 г. принята новая редакция этого стандарта.

ARINC-653 в качестве одного из основных требований для ОСРВ в авиации вводит архитектуру изолированных (partitioning) виртуальных машин.

Распространенным также является **стандарт OSEK/VDX** [OSEK], который первоначально развивался для систем автомобильной индустрии.

Стандарт OSEK/VDX является комбинацией стандартов, которые изначально разрабатывались в двух отдельных консорциумах, впоследствии слившихся. OSEK берет свое название от немецкого акронима консорциума, в состав которого входили ведущие немецкие производители автомобилей – BMW, Bosch, Daimler Benz (теперь Daimler Chrysler), Opel, Siemens и Volkswagen, а также университет в Карлсруэ (Германия). Проект VDX (Vehicle Distributed eXecutive) развивался совместными усилиями французских компаний PSA и Renault. Команды OSEK и VDX слились в 1994 г.

Первоначально проект OSEK/VDX предназначался для разработки стандарта открытой архитектуры ОС и стандарта API для систем, применяющихся в автомобильной промышленности. Однако разработанный стандарт получился более абстрактным и не ограничивается использованием только в автомобильной индустрии.

Стандарт OSEK/VDX состоит из трех частей – стандарт для операционной системы (OS), коммуникационный стандарт (COM) и стандарт для сетевого менеджера (NM). В дополнение к этим стандартам определяется некий реализационный язык (OIL). Первым компонентом стандарта OSEK является стандарт для ОС, поэтому часто стандарт OSEK ошибочно воспринимается как стандарт ОСРВ. Хотя ОС и есть большая часть данного стандарта, мощность его состоит в интеграции всех его компонентов.

ОС OSEK оперирует такими объектами, как задачи, события, ресурсы. Кроме того, обеспечиваются такие возможности, как управление ошибками и средства для пользовательских функций отслеживания изменений в состоянии системы.

Задача в ОС OSEK может быть:

* базовой или расширенной,
* вытесняемой или невытесняемой.

Главное различие между базовой и расширенной задачами заключается в том, может ли задача впасть в состояние ожидания (в котором она ждет появления события). Только расширенная задача может ожидать события. Вытесняемая задача может быть вытеснена задачей более высокого приоритета или прервана прерыванием.

Каждая задача имеет приоритет. Стандарт ОС OSEK не ограничивает максимальное количество приоритетов – это определяет реализация.

ОС OSEK определяет два уровня программ управления прерываниями, которые различаются возможностями вызова системных сервисов. Прерывания уровня 1 выполняются независимо от ОС очень быстро. Уровень 2 обеспечивает выполнение функций приложений, которые содержат вызовы ОС.

События в ОС OSEK используются для синхронизации различных задач. События являются собственностью задач. Любая задача, в том числе и базовая, может установить событие, и только собственник события может ожидать или снять его.

Управление ресурсами обеспечивает доступ к разделяемым ресурсам, таким как память, аппаратура и т.п. Планировщик также считается специальным ресурсом, который может быть захвачен задачами.

В связи со стандартами для ОСРВ стоит отметить широко известный стандарт критериев оценки пригодности компьютерных систем (Trusted Computer System Evaluation Criteria – TCSEC) [DoD85]. Этот стандарт разработан Министерством обороны США и известен также под названием "Оранжевая книга" (Orange Book – из-за цвета обложки).

В "Оранжевой книге" перечислены семь уровней защиты:

* А1 – верифицированная разработка. Этот уровень требует, чтобы защиту секретной и другой критичной информации средствами управления безопасностью гарантировали методы формальной верификации.
* В3 – домены безопасности. Этот уровень предназначен для защиты систем от опытных программистов.
* В2 – структурированная защита. В систему с этим уровнем защиты нельзя допустить проникновение хакеров.
* В1 – мандатный контроль доступа. Защиту этого уровня, возможно, удастся преодолеть опытному хакеру, но никак не рядовым пользователям.

Стандарт Trusted Computer System Evaluation Criteria (продолжение):

* С2 – дискреционный контроль доступа. Уровень С2 обеспечивает защиту процедур входа, позволяет производить контроль за событиями, имеющими отношение к безопасности, а также изолировать ресурсы.
* С1 – избирательная защита. Этот уровень дает пользователям возможность защитить личные данные или информацию о проекте, установив средства управления доступом.
* D – минимальная защита. Этот нижний уровень защиты оставлен для систем, которые проходили тестирование, но не смогли удовлетворить требованиям более высокого класса.

В ряде других стран были разработаны аналогичные критерии, на основе которых был создан международный стандарт “Общие критерии оценки безопасности информационных технологий” (далее просто – Общие критерии) (Common Criteria for IT Security Evaluation, ISO/IEC 15408) [CC99].

* EAL7 – самый высокий уровень предполагает формальную верификацию модели объекта оценки. Он применим к системам очень высокого риска.
* EAL6 определяется, как полуформально верифицированный и протестированный. На уровне EAL6 реализация должна быть представлена в структурированном виде, анализ соответствия распространяется на проект нижнего уровня, проводится строгий анализ покрытия, анализ и тестирование небезопасных состояний.
* EAL5 определяется, как полуформально спроектированный и протестированный. Он предусматривает создание полуформальной функциональной спецификации и проекта высокого уровня с демонстрацией соответствия между ними, формальной модели политики безопасности, стандартизованной модели жизненного цикла, а также проведение анализа скрытых каналов.
* EAL4 определяется, как методически спроектированный, протестированный и пересмотренный. Он предполагает наличие автоматизации управления конфигурацией, полной спецификации интерфейсов, описательного проекта нижнего уровня, подмножества реализаций функций безопасности, неформальной модели политики безопасности, модели жизненного цикла, анализ валидации, независимый анализ уязвимостей. По всей вероятности, это самый высокий уровень, которого можно достичь на данном этапе развития технологии программирования с приемлемыми затратами.
* EAL3 определяется, как методически протестированный и проверенный. На уровне EAL3 осуществляется более полное, чем на уровне EAL2, тестирование покрытия функций безопасности, а также контроль среды разработки и управление конфигурацией объекта оценки.
* EAL2 определяется, как структурно протестированный. Он предусматривает создание описательного проекта верхнего уровня объекта оценки, описание процедур инсталляции и поставки, руководств администратора и пользователя, функциональное и независимое тестирование, оценку прочности функций безопасности, анализ уязвимостей разработчиками.
* EAL1 определяется, как функционально протестированный. Он обеспечивает анализ функций безопасности с использованием функциональной спецификации и спецификации интерфейсов, руководящей документации, а также независимое тестирование. На этом уровне угрозы не рассматриваются как серьезные.

В соответствии с требованиями Общих критериев, продукты определенного класса (например, операционные системы) оцениваются на соответствие ряду функциональных критериев и критериев доверия – профилей защиты. Существуют различные определения профилей защиты в отношении операционных систем, брандмауэров, смарт-карт и прочих продуктов

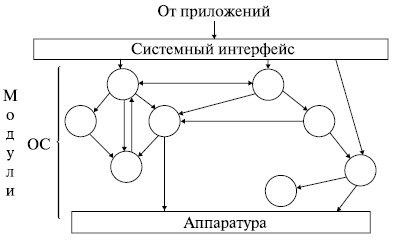
## Архитектура ОС

Под архитектурой операционной системы понимают структурную и функциональную организацию ОС на основе некоторой совокупности программных модулей.

Большинство современных ОС представляют собой хорошо структурированные модульные системы, способные к развитию, расширению и переносу на новые платформы. Какой-либо единой унифицированной архитектуры ОС не существует, но известны универсальные подходы к структурированию ОС. Принципиально важными универсальными подходами к разработке архитектуры ОС являются:

* модульная организация;
* функциональная избыточность (при наличии двух и более противоречивых целей создаются две или более программы, реализующие одну и ту же функцию, но работающие по разным принципам);
* функциональная избирательность (дополнительные функции и возможности для разных сфер применения могут быть включены при генерации системы);
* параметрическая универсальность (работа с разными наборами аргументов);
* концепция многоуровневой иерархической вычислительной системы, по которой ОС представляется многослойной структурой;
* разделение модулей на две группы по функциям: ядро – модули, выполняющие основные функции ОС, и модули, выполняющие вспомогательные функции ОС;
* разделение модулей ОС на две группы по размещению в памяти вычислительной системы: резидентные, постоянно находящиеся в оперативной памяти, и транзитные, загружаемые в оперативную память только на время выполнения своих функций;
* реализация двух режимов работы вычислительной системы: привилегированного режима (режима ядра – Kernel mode), или режима супервизора (supervisor mode), и пользовательского режима (user mode), или режима задачи (task mode);
* ограничение функций ядра (а следовательно, и количества модулей ядра) до минимального количества необходимых самых важных функций.

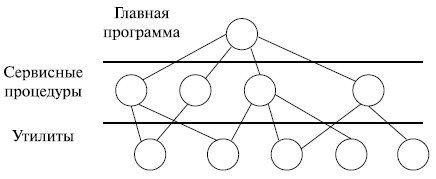
Первые ОС разрабатывались как монолитные системы без четко выраженной структуры. Для построения монолитной системы необходимо скомпилировать все отдельные процедуры, а затем связать их вместе в единый объектный файл. Каждая процедура видит любую другую процедуру. Однако даже такие монолитные системы могут быть немного структурированными.



Структурированная организация ОС предполагает следующие элементы:

* главная программа, которая вызывает требуемые сервисные процедуры;
* набор сервисных процедур, реализующих системные вызовы;
* набор утилит, обслуживающих сервисные процедуры.

В этой модели для каждого системного вызова имеется одна сервисная процедура. Утилиты выполняют функции, которые нужны нескольким сервисным процедурам.



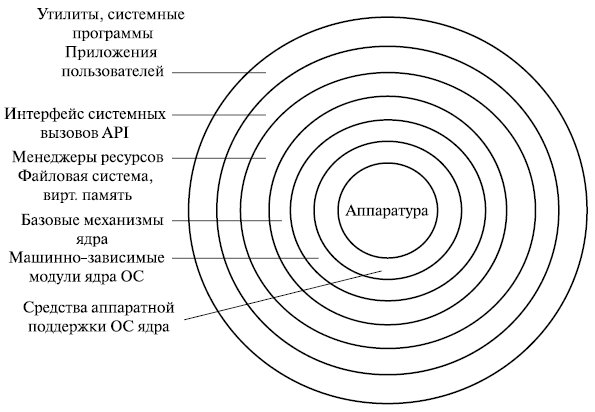
Классической считается архитектура ОС, основанная на концепции иерархической многоуровневой машины, привилегированном ядре и пользовательском режиме работы транзитных модулей. Модули ядра выполняют базовые функции ОС: управление процессами, памятью, устройствами ввода-вывода и т.п.

Ядро составляет сердцевину ОС, без которой она является полностью неработоспособной и не может выполнить ни одну из своих функций. В ядре решаются внутрисистемные задачи организации вычислительного процесса, недоступные для приложения.

Особый класс функций ядра служит для поддержки приложений, создавая для них так называемую прикладную программную среду. Приложения могут обращаться к ядру с запросами – системными вызовами – для выполнения тех или иных действий

Для обеспечения высокой скорости работы ОС модули ядра (по крайней мере, большая их часть) являются резидентными и работают в привилегированном режиме (Kernel mode). Этот режим, во-первых, должен обезопасить работу самой ОС от вмешательства приложений, и, во-вторых, должен обеспечить возможность работы модулей ядра с полным набором машинных инструкций, позволяющих собственно ядру выполнять управление ресурсами компьютера, в частности, переключение процессора с задачи на задачу, управлением устройствами ввода-вывода, распределением и защитой памяти и др.

Остальные модули ОС выполняют не столь важные функции, как ядро, и являются транзитными. Это утилиты, системные обрабатывающие программы, библиотеки процедур различного назначения и т.д. Они обращаются к функциям ядра посредством системных вызовов и выполняются в пользовательском режиме (user mode).



Многослойная структура ОС.

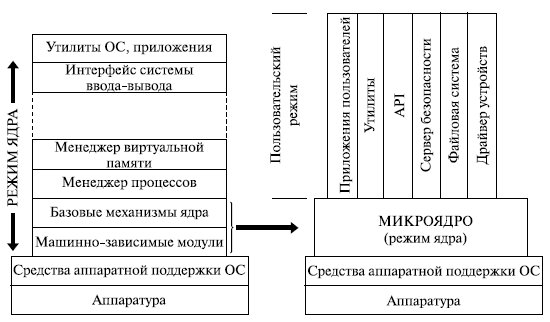
В концепции многоуровневой (многослойной) иерархической машины структура ОС также представляется рядом слоев. При такой организации каждый слой обслуживает вышележащий слой, выполняя для него некоторый набор функций, которые образуют межслойный интерфейс. На основе этих функций следующий верхний по иерархии слой строит свои функции – более сложные и более мощные и т.д.

Такая организация системы существенно упрощает ее разработку, т.к. позволяет сначала "сверху вниз" определить функции слоев и межслойные интерфейсы, а при детальной реализации, двигаясь "снизу вверх", – наращивать мощность функции слоев. Кроме того, модули каждого слоя можно изменять без необходимости изменений в других слоях.

Повышение устойчивости ОС обеспечивается переходом ядра в привилегированный режим. Системный вызов привилегированного ядра инициирует переключение процессора из пользовательского режима в привилегированный, а при возврате к приложению – обратное переключение. За счет этого возникает дополнительная задержка в обработке системного вызова. Однако такое решение стало классическим и используется во многих ОС (UNIX, VAX, VMS, IBM OS/390, OS/2 и др.).



Как альтернатива классическому варианту архитектуры ОС, часто используется микроядерная архитектура ОС. В привилегированном режиме остается работать только небольшая часть ОС, называемая микроядром. Микроядро защищено от остальных частей ОС и приложений. Все высокоуровневые функции ядра оформляются как модули, работающие в пользовательском режиме.



Внешние по отношению к микроядру компоненты ОС реализуются как обслуживающие процессы и называются серверами ОС. Между собой они взаимодействуют с помощью обмена сообщениями, которые передаются через микроядро.



В ОС с микроядерной архитектурой выполнение системного вызова сопровождается четырьмя переключениями режимов (4 t), в то время как в классической архитектуре – двумя. Следовательно, производительность ОС с микроядерной архитектурой при прочих равных условиях будет ниже, чем у ОС с классическим ядром.



В то же время признаны следующие достоинства микроядерной архитектуры:

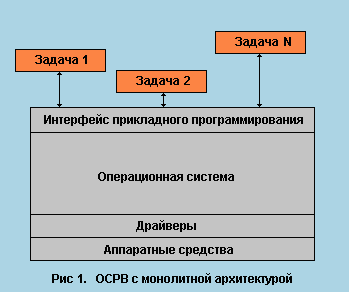
* единообразные интерфейсы;
* простота расширяемости;
* высокая гибкость;
* возможность переносимости;
* высокая надежность;
* поддержка распределенных систем;
* поддержка объектно-ориентированных ОС.

Развитие микроядер операционных систем.

* микроядро первого поколения (Mach) – порядка 100000 строк кода, на диске –500 Кбайт и 140 интерфейсов системных вызовов;
* микроядро ОС L4 (второе поколение) – порядка 10000 строк кода, на диске – 100 Кбайт, а в памяти – 12 Кбайт и 7 интерфейсов системных вызовов.

В микроядре L4 компонент IPC в 20 раз быстрее IPC из микроядра Mach.

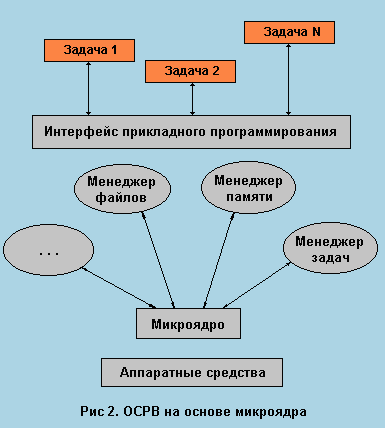
## Архитектура ОС реального времени



Простая монолитная ОС для встроенных систем (RTOS-32, RTEMS, Wind River Linux).

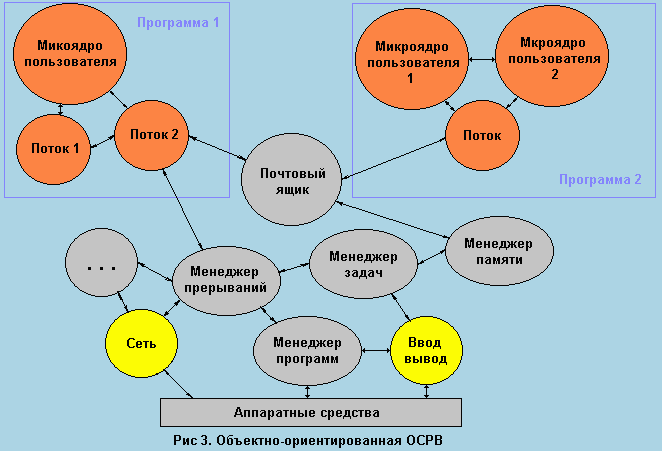
Имеет набор модулей ядра, предоставляющих приложениям входные интерфейсы для обращений к аппаратуре. Переход из пользовательского режима в режим ядра осуществляется через системные вызовы.

Основные недостатки такой ОС: плохая предсказуемость её поведения, вызванная сложным взаимодействием модулей между собой; плохая переносимость; сложность расширения. Преимуществом является высокое быстродействие.



ОСРВ на основе микроядра (QNX, VxWorks) имеет целый ряд преимуществ.

Микроядро обеспечивает только пять различных типов сервисов: управление виртуальной памятью; поддержка заданий и потоков; взаимодействие между процессами (Inter-Process Communication, IPC); управление поддержкой ввода-вывода и прерываниями; сервисы хоста (host) и процессора.



Объектно-ориентированные ОСРВ (OS-9, Taligent, WorkPlace, Cairo).

Каждый программный компонент является функционально изолированным от других. Объект может быть представлением как конкретных вещей – прикладной программы или документа, так и некоторых абстракций – процесса, события.

Внутренняя структура данных объекта скрыта от наблюдения. Нельзя произвольно изменять данные объекта. Для того, чтобы получить данные из объекта или поместить данные в объект, необходимо вызывать соответствующие объектные функции.

## Февраль, ЛК3. Операционные системы реального времени

Подразделяют ОСРВ на два класса — **системы «жесткого» и «мягкого» реального времени**. Системы «жесткого» РВ имеют минимальные объем и время отклика, но обладают ограниченными сервисными средствами. Типичным примером ОСРВ этого класса служит VxWorks.

Системы «мягкого» РВ требуют большего объема памяти, имеют более длительное время отклика, но удовлетворяют широкому спектру требований пользователя по режиму обслуживания задач, уровню предоставляемого сервиса. Примером такой ОСРВ может служить OS-9/9000.

Однако для современных ОСРВ данная классификация является весьма условной.

Также ОСРВ можно разделить на **системы специализированные и универсальные**.

Специализированной ОСРВ называется система, где конкретные временные требования априори определены.

Универсальная СРВ должна уметь выполнять произвольные временные задачи без применения специальной техники. Разработка таких систем является сложной задачей, хотя обычно требования, предъявляемые к таким системам, мягче, чем требования для специализированных систем.

### ОС реального времени OS-9

OS-9 компании Microware относится к классу UNIX-подобных операционных систем реального времени. Благодаря модульному объектно-ориентированному дизайну OS-9 является чрезвычайно гибко конфигурируемой высокопроизводительной системой реального времени.

Все функциональные компоненты OS-9, включая ядро, иерархические файловые менеджеры, систему ввода/вывода и средства разработки, реализованы в виде независимых модулей. Модульность системы означает, что она может быть масштабирована для удовлетворения нужд как маленьких встроенных систем, так и больших сетевых приложений.

Спектр применений OS-9 широк – промышленная автоматизация, инструментальные и измерительные системы, системы военного и космического назначения. В последние годы она применяется в таких современных системах, как мобильные телекоммуникационные устройства, встраиваемые терминалы доступа в Интернет, интерактивные цифровые телевизионные приставки.

Ядро OS-9 содержит более 90 системных вызовов, которые дают возможность управлять динамическим режимом диспетчеризации, распределением памяти, межпроцессорной коммуникацией и т.д. – вплоть до управления встраиваемым в ядро ОС режимом экономичного потребления питания.

Характеристики производительности ядра: 5,6 мкс – время задержки прерывания), 14 мкс – время переключения контекста процесса (для процессора MC68040, 30MHz).

Ядро OS-9 является масштабируемым, полностью вытесняемым, поддерживает функционирование до 65535 процессов, предоставляет 65535 уровней приоритета и обеспечивает работу до 255 пользователей.

Поддержка микропроцессоров серии Motorola, Intel 80386/486, Pentium, PowerPC.

Основным принципом построения OS-9 является максимальная открытость структур и функций.

Реентерабельность достигается за счет сосредоточения постоянных частей кода в модулях памяти. Перемещаемость подразумевает создание компиляторами кода, не содержащего абсолютных адресов. Любая программа может быть загружена и исполнена с любого абсолютного адреса, и эта возможность делает тривиальной задачу размещения ОС (вместе с приложениями) в ПЗУ.

В результате любые системные (за исключением самого ядра и модуля данных) и прикладные модули могут добавляться или удаляться из системы в процессе ее функционирования без какой-либо повторной компиляции или компоновки.

Спектр применений OS-9 широк – промышленная автоматизация, инструментальные и измерительные системы, системы военного и космического назначения. В последние годы она применяется в таких современных системах, как мобильные телекоммуникационные устройства, встраиваемые терминалы доступа в Интернет, интерактивные цифровые телевизионные приставки.

Для поддержки таких сложных приложений, как телекоммуникации, мультимедиа и системы выдачи видеоданных по запросу, фирма Microware разработала ряд дополнительных файловых менеджеров.

Пользовательский интерфейс мультимедиаприложения MAUI содержит расширенный набор протоколов API для соответствия требованиям высокопроизводительных мультимедиа-протоколов либо протоколов пользователя. С данным интерфейсом могут общаться такие являющиеся промышленным стандартом пакеты, как Apple QuickDraw и QuickTime, Macromedia Director, Oracle Media Objects и Sybase Gain.

Файловый менеджер для приложений мультимедиа MPFM, соответствующий спецификациям MPEG, рассчитан на использование в различных приложениях мультимедиа, включающих интерактивное телевидение, образование, обучение и выдачу видеоинформации по запросу.

OS-9 предоставляет широкий набор средств разработки: локализованные (резидентные), удаленные (внешние), использующих принципиально отличные операционные среды. Локализованные системы разработки функционируют в той же вычислительной среде, что и разрабатываемое приложение. Их достоинством является простота (следовательно, невысокая стоимость), быстрый переход от разработки к исполнению и наоборот, дополнительные возможности управления исполнением.

Удаленные (внешние) системы разработки – это и традиционные кросс-системы (такие, как PC-Bridge, работающая под управлением DOS, UniBridge , исполняющаяся в многопользовательской UNlX-среде) и профессиональные графические среды разработки ( такие как, FasTrak).

Как правило, разработка программ ведется в полнофункциональных конфигурациях. После того, как будет отлажен код программы реального времени, отсоединяются модули разработки и ввода-вывода, и полученный код готов к исполнению под управлением ядра в целевой системе.

Характеристики использования OS-9:

* Поддержка HOST-систем (IBM PC(MS Windows 3.xx, 95, NT), IBM RS6000/AIX, Sun4/SunOS/Solaris, HP9000 S/700, SGI IRIS/IRIX)
* Широкая поддержка сторонних разработчиков программного обеспечения
* Широкая поддержка разработчиков аппаратных средств промышленной автоматизации
* Программные продукты для «вертикальных» рынков (мобильная беспроводная коммуникация, устройства с минимальным потреблением энергии, мультимедиа)
* Специальные программные средства и лицензионная политика для OEM.
* Более 5 млн. установленных копий
* Более 800 OEM-партнеров

### ОС реального времени VxWorks

ОСРВ VxWorks компании Wind River Systems является встраиваемой ОС реального времени для систем жесткого реального времени. Значительная часть используемого в Интернет сетевого оборудования, коммутаторы, маршрутизаторы, серверы удаленного доступа и устройства широкополосного доступа работают под управлением VxWorks.

VxWorks построена по технологии микроядра, т.е. на нижнем непрерываемом уровне ядра выполняются только базовые функции планирования задач и управления коммуникацией/синхронизацией между задачами. Все остальные функции ОС более высокого уровня — управление памятью, вводом/выводом, сетевые средства, и т.д. — базируются на простых функциях нижнего уровня, Такая иерархическая организация позволяет обеспечить быстродействие и детерминированность ядра, а также легко строить необходимую конфигурацию ОС.

Микроядро с минимальным набором модулей занимает 20-40 Кбайт памяти. Для встроенных систем, имеющих жесткие ограничения на объем памяти, разработано редуцированное ядро Wind Stream, которое требует для работы всего 8 Кбайт ПЗУ и 2 Кбайт ОЗУ.

Разработка ведется на инструментальном компьютере (host) в среде Tornado для последующего исполнения на целевой машине (target) под управлением VxWorks.

Инструментальная среда Tornado имеет открытую архитектуру, что позволяет другим фирмам-производителям инструментальных средств разработки ПО реального времени интегрировать свои программные продукты с Tornado. Пользователь также может подключать к Tornado свои собственные специализированные средства разработки, а также расширять возможности инструментальных средств фирмы Wind River Systems.

В стандартную конфигурацию Tornado входят ядро VxWorks и системные библиотеки, GNU C/C++ Toolkit, дистанционный отладчик уровня исходного языка CrossWind, оболочка WindSh, конфигуратор BSP WindConfig и др. Существует также множество интегрированных с Tornado программных продуктов других фирм.

VxWorks поддерживает целевые архитектуры (targets):

* Motorola 680x0 и CPU32, Intel 386/486/Pentium/.., Intel 960, SPARC, Mips R3000/4000, ARM, Motorola 88110, HP PA-RISC, Hitachi SH7600, PowerPC, DEC Alpha, Siemens C16x.

Инструментальные платформы, поддерживаемые для Tornado (hosts):

* Sun SPARCstation (SunOS и Solaris), HP 9000/400,700 (HP-UX), IBM RS6000 (AIX), Silicon Graphics (IRIX), DEC Alpha (OSF/1), PC (Windows 95 и NT).

Поддерживаемые интерфейсы host-target:

* Ethernet, RS-232, внутрисхемный эмулятор ICE (In-Circuit Emulator), кросс-шина (backplane), ROM-эмулятор, BDM-интерфейс (Background Debug Mode).

### ОС реального времени Wind River Linux

Wind River Linux компании Wind River Systems Inc. — коммерческий дистрибутив ОС Linux, предназначенный для разработки встраиваемых приложений. Это полнофункциональный и комплект, позволяющий разработчикам устройств достигать большей результативности с меньшими затратами, исключая из Linux-проектов множество ресурсоемких рутинных задач.

В состав дистрибутива входят более 500 готовых программных пакетов, включая расширения жесткого реального времени и пакеты сетевых протоколов, а также интегрированная среда разработки Wind River Workbench.

Wind River Workbench — интегрированная среда разработки встраиваемого ПО для ОС VxWorks и Wind River Linux. Она содержит в себе инструментарий для полного цикла разработки ПО встраиваемых систем, от инициализации оборудования до управления конечным устройством, включая средства отладки, анализа кода, виртуализации, диагностики и тестирования.

В состав Wind River Workbench входят инструменты мониторинга, анализа и отладки для каждой фазы разработки проекта. Поддержка в Wind River Workbench различных ОС, процессорных архитектур и языков программирования предоставляет разработчикам большую гибкость.

Функциональность Wind River Workbench может быть расширена подключением к ней инструментария модульного и интеграционного тестирования IPL Cantata++. Wind River Workbench также поддерживает аппаратные модули внутрисхемной отладки Wind River ICE и Wind River Probe, позволяя разработчикам выявлять и решать проблемы на самых ранних стадиях разработки устройств.

Wind River Linux имеет монолитное ядро с загружаемыми модулями, реализует многопоточность.

Полностью соответствует стандарту POSIX.

Исходный текст доступен целиком.

Поддерживаемые процессоры: x86, ARM, MIPS, PowerPC, SPARC.

Многопроцессорность (SMP/AMP), многоядерные процессоры, виртуализация.

**Типовые применения**: сети/телекоммуникации, мобильные устройства, потребительская электроника, промышленные приложения.

### ОС реального времени OSEK/VDX

Проект OSEK/VDX (Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen/Vehicle Distributed eXecutive) предназначался для разработки стандарта открытой архитектуры ОС и стандарта API для систем автомобильной промышленности. Его разработку финансировали BMW, Bosch, DaimlerChrysler, Siemens, Opel и Volkswagen.

ОС OSEK прежде всего ориентирована на использование в автомобильных приложениях и поддерживает в настоящее время множество типов встраиваемых микроконтроллеров: HC08, HC12, 683XX, MCore, Motorola PowerPC.

ОС OSEK оперирует такими объектами, как задачи, события, ресурсы. Кроме того, обеспечиваются такие возможности, как управление ошибками и средства для пользовательских функций отслеживания изменений в состоянии системы.

Задача в ОС OSEK может быть

* базовой или расширенной,
* вытесняемой или невытесняемой.

Главное различие между базовой и расширенной задачами заключается в том, может ли задача впасть в состояние ожидания (в котором она ждет появления события). Только расширенная задача может ожидать события. Вытесняемая задача может быть вытеснена задачей более высокого приоритета или прервана прерыванием. Невытесняемая задача может быть вытеснена только с помощью прерывания (когда прерывания не запрещены).

ОС OSEK определяет два уровня программ управления прерываниями, которые различаются возможностями вызова системных сервисов. Прерывания уровня 1 выполняются независимо от ОС быстро. Уровень 2 обеспечивает выполнение функций приложений, которые содержат вызовы ОС.

События в ОС OSEK используются для синхронизации различных задач. События являются собственностью задач. Любая задача, в том числе и базовая, может установить событие, и только собственник события может ожидать или снять его.

Управление ресурсами обеспечивает доступ к разделяемым ресурсам, таким как память, аппаратура и т.п. Планировщик также считается специальным ресурсом, который может быть захвачен задачами.

Аварийные сигналы и счетчики в OSEK используются для синхронизации активации задач с повторяющимися событиями. Аварийный сигнал статически присваивается счетчику, задаче и воздействию. Воздействие может либо активировать задачу, либо установить событие. Счетчики оперируют тактами и могут представлять время, количество принятых импульсов и т.п. Каждая реализация обеспечивает один временной счетчик, который используется для планирования периодических событий.

В OSEK существует два типа аварийных сигналов: циклические и одинарные. Циклические аварийные сигналы применяются для диспетчеризации задачи, которая должна запускаться периодически. Счетчик аварийного сигнала может быть установлен в относительное или абсолютное значение.

ОС OSEK обеспечивает минимальные средства для управления ошибками времени выполнения. Во время выполнения большинство системных сервисов не возвращает коды ошибок. Однако имеется возможность дополнительного управления ошибками во время разработки благодаря расширенной функциональности возврата управления.

В ОС OSEK есть два типа ошибок – ошибки приложения и фатальные ошибки. При ошибке приложения, когда приложение пытается выполнить несанкционированную операцию, целостность данных все еще сохраняется. Фатальные ошибки возникают, если ОС обнаруживает нарушение целостности внутренних данных. При выявлении таких ошибок вызывается сервис завершения работы ОС.

Фирма Metrowerks, независимая дочерняя компания фирмы Motorola, включила в состав своих отладочных средств CodeWarrior операционную систему OSEK фирмы Motorola. ОС OSEK соответствует стандартам программной модели Automotive Open System Architecture (AutoSar).

### ОС реального времени LynxOS

LynxOS — операционная система жесткого реального времени, поддерживающая многопроцессные и многопотоковые приложения, которая предназначена для специализированной и телекоммуникационной аппаратуры. Эта ОС является полностью детерминированной и обладает POSIX-, UNIX- и Linux-совместимостью. Областями применения ОС LynxOS являются также сложные системы безопасности.

Разработана компанией LynuxWorks, Inc. Важным аспектом бизнес-политики компании LynuxWorks является возможность приобретения системным интегратором исходных текстов операционной системы LynxOS-178.

Полностью поддерживается стандарт POSIX.1003.1a, а также подразделы POSIX.1003.1b и POSIX.1003.1c. LynxOS сертифицирована на согласованность с POSIX 1003.1-1996 в 1996 году для платформ Intel и PowerPC.

Программы, написанные и откомпилированные в ОС Linux, могут запускаться и работать в среде LynxOS без каких-либо изменений в исходных текстах и без перекомпилирования. Это свойство LynxOS является уникальным для систем реального времени и очень удобным для пользователей (например, в случае отсутствия исходных текстов). LynxOS обеспечивает совместимость с Linux на уровне ABI (Application Binary Interface), уровне форматов объектных файлов, вызовов API, динамически подключаемых библиотек, компоновки и загрузки на этапе выполнения.

Поддерживается множество аппаратных архитектур (x86, PowerPC, MIPS) для оборудования различных фирм-производителей. LynxOS 5.0 может работать в режиме SMP на платах c несколькими процессорными ядрами или процессорами.

Сетевые средства: IPv4 и IPv6, IPSec/IKE/VPN, QoS, протоколы TCP, UDP, ICMP, IGMP, ARP, RARP, DHCP, NAT, RPC, NTPv3, RIP, RIPv2, DVMRP, BGP4, OSPF, OSPFv2, SNMP v1, v2 и v3.

Последняя выпущенная версия этого бренда ОС LynxOS-178 2.0 характеризуется производителем как коммерческая операционная система, обеспечивающая высокий уровень надежности и оперативности, необходимый для встраиваемых приложений с особыми требованиями к безопасности.

В LynxOS-178 2.0 реализована поддержка интерфейса APEX (APlication/EXecutive – интерфейс приложения/управляющей программы) спецификации ARINC-653. Это означает, что данная операционная система отвечает самым строгим требованиям к безопасности и надежности электронных систем для военной и гражданской авиации. Система LynxOS-178 2.0 полностью соответствует положениям уровня А спецификации DO-178B.

Также ОСРВ LynxOS-178 2.0 соответствует требованиям стандартов POSIX и DO-178B, что означает гарантию переносимости прикладного кода встраиваемых систем, многократного использования созданных программ, а также соответствие самым строгим нормативам операционных систем с повышенными требованиями к безопасности.

Ключевое свойство LynxOS-178 – это поддержка нескольких полностью изолированных по времени, памяти и ресурсам разделов в соответствии с требованиями ARINC-653 (рис. 1). Это особенно важно при реализации концепции ИМА (интегрированная модульная авионика). Операционная система LynxOS-178 (версия 2.0) поддерживает:

* до 16 разделов (виртуальных машин), включая корневой раздел;
* до 64 процессов в каждом разделе;
* до 51 потока (нити) внутри каждого процесса;
* диспетчеризацию реального времени потоков внутри раздела;
* POSIX-функции межпроцессного взаимодействия внутри раздела.

Каждый раздел полностью изолирован, поэтому распространение сбоев между ними исключено. Это разделение относится к процессорному времени, адресному пространству и ресурсам. С помощью LynxOS-178 фиксированные разделы обслуживаются как виртуальные машины LynxOS.

### ОС реального времени Windows CE

Windows CE (также известна как WinCE) — это вариант операционной системы Microsoft Windows для наладонных компьютеров, смартфонов и встраиваемых систем. Сегодня Windows CE (Compact Edition/Compact Embedded) не является «урезанной» версией Windows для настольных ПК, она основана на совершенно другом ядре и является операционной системой реального времени с набором приложений, основанных на Microsoft Win32 API.

Примерно с 2000 года ОС Windows CE развивалась самостоятельно, и современные версии ОС Windows CE сильно отличаются от современных версий настольных ОС семейства Windows NT, но при этом сохраняется программная совместимость с приложениями, основанными на Microsoft Win32 API.

Windows CE является модульной с небольшим ядром и необязательными модулями, которые выполняются как независимые процессы. Планирование в Windows CE осуществляется на основе приоритетов. Поддерживается защита ядра и процессов друг от друга. Кроме того, возможен режим работы, когда отсутствует защита между процессами и ядром.

Это компонентная многозадачная многопоточная многоплатформенная операционная система с поддержкой реального времени. Разработчикам доступны около 600 компонентов, при использовании которых они могут создавать собственные образы операционной системы, которые включают только необходимую данному конкретному устройству функциональность.

Windows CE предоставляет разработчикам приложений набор API, основанный на стандартном Win32 API и дополненный специализированным API для встраиваемых устройств. Поскольку CE поддерживает только часть Win32 API и имеет определённую специфику, связанную со встраиваемой природой операционной системы, приложения, написанные для настольных версий операционной системы Windows, могут потребовать дополнительной адаптации и модификации для запуска их на встраиваемых устройствах; и в любом случае, для запуска программ на устройстве потребуется их перекомпиляция.

Но так же, как и настольные версии Windows, Windows CE использует стандартный формат исполняемого файла — Portable Executable (PE). Это позволяет разработчикам использовать большинство стандартных утилит, работающих с форматом PE, например, Dependency Walker (проверка зависимостей) или DumpBin.

Часто названия Windows CE, Windows Mobile, Pocket PC используют как взаимозаменяемые. Это не совсем правильно. Windows CE — это модульная операционная система, которая служит основой для устройств нескольких классов.

Windows Mobile лучше всего представлять себе как набор платформ, основанных на Windows CE. В настоящее время в этот набор входят платформы: Pocket PC, SmartPhone и Portable Media Center. Каждая платформа использует свой набор компонентов Windows CE, плюс свой набор сопутствующих особенностей и приложений.

Любой разработчик может купить инструментарий (Platform Builder), который содержит все эти компоненты и программы, позволяющие построить собственную платформу. При этом такие приложения, как Word Mobile/Pocket Word, не являются частью этого инструментария. Согласно программе лицензирования Shared source исходный код может быть предоставлен сторонним разработчикам программного обеспечения.

На базе Windows CE основано множество платформ, включая Handheld PC, Palm-size PC, Pocket PC, Pocket PC 2002, Pocket PC 2003, Pocket PC 2003 SE, Smartphone 2002, Smartphone 2003, Windows Mobile, Meizu OS, а также множество промышленных устройств и встроенных систем.

Windows CE оптимизирована для устройств, имеющих минимальный объём памяти: ядро Windows CE может работать на 32 КБ памяти. С графическим интерфейсом (GWES) для работы Windows CE понадобится от 5 МБ. Устройства часто не имеют дисковой памяти и могут быть сконструированы как «закрытые» устройства, без возможности расширения пользователем (например, ОС может быть «зашита» в ПЗУ).

Хотя Windows CE имеет модульную структуру, которая позволяет создавать минимальные конфигурации для небольших систем, она все-таки остается сложной и требует относительно большого пространства на диске, поэтому она не является хорошим выбором для глубоко встраиваемых систем.

### RTX (Real Time Extension)

Microsoft Windows «в чистом виде» нельзя отнести к операционным системам реального времени.

Одним из возможных решений является использование совместно с Microsoft Windows подсистемы реального времени, исполняющейся на том же процессоре (если процессор один) или на выделенном процессоре(-ах) (если их несколько). Этот подход использован фирмой VenturCom в продукте RTX (Real Time Extension). Сущность подхода заключается в использовании модифицированного HAL (Hardware Abstraction Level). Изменять ядро Microsoft не разрешает, а исходный код HAL предоставляет своим партнерам, одним из которых является VenturCom.

После установки RTX стандартная Microsoft Windows Workstation или Server превращается в операционную систему реального времени с жестким детерминизмом (hard real-time). Сама Windows об этом, правда, не подозревает. Ни ядро, ни исполняющая подсистема ОС не изменяются. Подсистема реального времени видна из Microsoft Windows, как еще один драйвер устройства.

### ОС реального времени RTEMS

RTEMS (англ. Real-Time Executive for Multiprocessor Systems) — свободная операционная система реального времени с открытым исходным кодом, разработанная для встраиваемых систем.

Изначально буква "М" означала Missile (ракетный), потом Military (военный). В настоящее время в версиях RTEMS, реализованных на языке С, буква "М" подразумевает Multiprocessor, в версиях на языке Ada буква "M" подразумевает Military. Разработчик системы компания OAR (On-Line Applications Research Corporation, США). Система была создана по заказу министерства обороны США для использования в системах управления ракетными комплексами.

На данную ОСРВ отсутствуют какие-либо экспортные ограничения, она свободно распространяется в исходных кодах через Интернет (www.OARcorp.com). RTEMS находится под защитой модифицированной версии публичной лицензии GNU. Компания OAR обеспечивает поддержку, обучение специалистов и разработку программного обеспечения по заказам. Производится также бесплатная рассылка информационных материалов по запросу.

Согласно классификации ОСРВ RTEMS следует отнести к монолитным ядрам реального времени.

ОСРВ RTEMS обеспечивает высокоэффективную среду исполнения для «глубоко встраиваемых» систем (Deeply Embedded Segment), в которых не предполагается частая перенастройка или смена алгоритмов функционирования.

Это могут быть портативные приложения для портативных устройств (например, сотовые телефоны) или приложения, чувствительные к стоимости (например, автомобильная автоматика), в которых используются аппаратные средства с ограниченными ресурсами (микроконтроллеры). RTEMS отличается модульностью, высокой масштабируемостью и предсказуемостью.

ОС RTEMS написана на языке высокого уровня, поэтому ее перенос на различные процессорные платформы производится с минимальными трудозатратами.

Поддерживаются процессоры: ARM, Blackfin, Freescale ColdFire, цифровые сигнальные процессоры Texas Instruments серий C3x/C4x, Hitachi H8/300, i386, Pentium, Motorola 68000, MIPS, Nios II, PowerPC, SuperH, SPARC, OpenRisc.

Также поддерживаются мультипроцессорные конфигурации.

Ядро RTEMS автоматически учитывает различия в архитектуре используемых процессоров, выполняя в случае необходимости перестановку байтов и другие процедуры. Это позволяет осуществлять переход на другое семейство процессоров без значительных изменений системы.

Характерные особенности ядра RTEMS:

* поддержка однородных и неоднородных мультипроцессорных систем;
* управляемое событиями распределение машинного времени в соответствии с приоритетами (динамический алгоритм диспетчеризация);
* наличие менеджера задач, который обеспечивает гибкую конфигурацию параметров переключения контекстов, позволяя монотонно изменять минимальный размер выделенного задаче кванта времени (optional rate monotonic scheduling);
* надежное управление прерываниями (responsive interrupt management), гарантирующее своевременное обнаружение запросов и вызов программ обработки;
* большое разнообразие реализуемых механизмов взаимодействия и синхронизации задач;
* динамическое выделение памяти;
* широкие возможности настройки параметров системы.

В качестве достоинств RTEMS необходимо отметить следующие.

* Это операционная система жесткого реального времени, надежность которой подтверждается опытом длительного использования в аппаратуре специального назначения (ракетных системах и других ответственных сферах применения).
* Данная ОСРВ является бесплатной и поставляется в исходных кодах.
* ОСРВ RTEMS позволяет легко применять разработанные приложения на разных платформах.
* RTEMS это мультипроцессорная ОС, на базе которой можно создавать разветвленные системы контроля и управления, устойчивые к различного рода воздействиям.
* Модульная структура ядра RTEMS позволяет в широких пределах варьировать размер и возможности этой ОСРВ в зависимости от конкретной области применения.
* Поддержка стандартного стека сетевых протоколов TCP/IP обеспечивает возможность широкого применения RTEMS в сетевых приложениях.
* Соответствие международным стандартам, поддержка стандартных методов синхронизации между задачами, наличие пользовательского API и свободно распространяемых средств разработки и отладки, реализованных на базе проекта GNU, позволяют пользователю создавать на базе RTEMS эффективные системы управления с минимальными затратами.
* Малое время загрузки/реанимации приложения в случае аварии/отключения питания (менее 1c) обеспечивает быстрое восстановление работоспособности систем, использующих RTEMS.

Следует также отметить ряд недостатков, ограничивающих возможности применения RTEMS.

* RTEMS не поддерживает динамическую загрузку приложений и модулей, поэтому сферой её применения являются встроенные системы, в которых не предполагается частая модификация программного обеспечения.
* ОСРВ RTEMS обеспечивает достаточно слабую поддержку файловых систем, что ограничивает область ее возможного применения в сфере систем централизованного сбора и хранения данных стандартными высокоуровневыми средствами. На настоящий момент RTEMS поддерживает только файловые системы IMFS и TFTP, что явно недостаточно.
* В RTEMS фактически отсутствуют резиденты средства отладки. Имеются только стандартные функции rtems\_panic и printf, которые позволяют выводить отладочную информацию на терминал в процессе работы системы. Следует, однако, отметить, что наличие мощных средств кросс-разработки делает этот недостаток не очень существенным.

Наиболее перспективными областями применения RTEMS являются:

* промышленные контроллеры;
* телекоммуникационное оборудование;
* контрольно-измерительная аппаратура;
* разветвленные системы контроля и управления.

Базовой средой разработчика при работе с RTEMS является ОС Linux (Unix). Возможна настройка рабочего места для использования Windows 9x, Windows NT, Windows 2000 при работе в среде Cygwin.

### ОС реального времени FreeRTOS

FreeRTOS — это многозадачная, мультиплатформенная, бесплатная операционная система жесткого реального времени с открытым исходным кодом. Разработана компанией Real Time Engineers Ltd. специально для встраиваемых систем.

FreeRTOS 6.1.0 официально поддерживает 23 архитектуры и 57 платформ (в подавляющем большинстве — микроконтроллеры).

Существуют так называемые официально поддерживаемые аппаратные платформы — официальные порты и неофициальные, которые поставляются «как есть» и не поддерживаются напрямую. Кроме того, для одного и того же порта могут поддерживаться несколько средств разработки. Поставляется с отлаженными примерами проектов для каждого порта и для каждой среды разработки.

Большая часть кода FreeRTOS написана на языке Си, ассемблерные вставки минимального объема применяются лишь там, где невозможно применить Си из-за специфики конкретной аппаратной платформы.

Система FreeRTOS является относительно небольшим приложением. Минимальный вариант монолитного ядра системы FreeRTOS состоит всего лишь из трех файлов исходного кода (.c) и горстка файлов заголовков, общий размер которых составляет менее 9000 строк кода, включая комментарии. Размер типичного образа системы в исполняемом коде меньше 10 КБ.

Дизайн системы FreeRTOS позволяет ее легко настраивать в широких пределах. Система FreeRTOS может собрана как для одного процессора, для выполнения только основных функций и поддержки только нескольких задач, либо она может быть собрана для работы на многоядерном железе с TCP/IP, файловой системой и USB.

Основные характеристики FreeRTOS:

* Планировщик FreeRTOS поддерживает три типа многозадачности:
  + вытесняющую;
  + кооперативную;
  + гибридную.
* Возможность отслеживать факт переполнения стека.
* Нет программных ограничений на количество одновременно выполняемых задач.
* Нет программных ограничений на количество приоритетов задач.
* Нет ограничений в использовании приоритетов: нескольким задачам может быть назначен одинаковый приоритет.

FreeRTOS полностью бесплатна, модифицированная лицензия GPL позволяет использовать FreeRTOS в проектах без раскрытия исходных кодов. Документация в виде отдельного документа платная, но на официальном сайте доступно исчерпывающее техническое описание на английском языке.

Проект «SafeRTOS» — доработанный, документированный, протестированный и прошедший сертификацию на соответствие стандарту безопасности IEC 61508 вариант FreeRTOS. Это коммерческий продукт, отличается от свободной версии помимо лицензии и предоставляемых гарантий лишь некоторыми деталями.

### ОС реального времени QNX

Операционная система QNX является разработкой канадской компании QNX Software System Ltd. по заказу Минобороны США. Впервые система появилась на рынке в 1981 году. Среди пользователей QNX значатся такие компании, как Du Pont, Eastman Kodak, General Mills, General Motors, Motorola, Texaco.

Операционная система QNX представляет собой гибрид 16/32-битовой операционной системы, которую пользователь может конфигурировать по своему усмотрению. Время, необходимое для полной инсталляции системы, включая сетевые средства, составляет всего 10-15 минут, после чего можно начинать работу. Нетребовательность системы к ресурсам проявляется уже в том, что система с необходимой и достаточной средой разработки в виде компилятора Watcom C/C++ (основной компилятор для QNX) умещается в 10 Мбайт.

QNX является ОСРВ на основе микроядра (размером около 10 Кбайт). В качестве основного средства взаимодействия между процессами система использует передачу сообщений. Благодаря этому в 32-битовой среде возможно взаимодействие процессов с 32 и 16-битовым кодом.

Причем сообщения передаются между любыми процессами, не зависимо от того, находятся ли процессы на одном компьютере или на разных узлах сети. Пользователь, работая на одном из узлов сети, может иметь доступ к любым ресурсам остальных узлов, включая порты, файловую систему и задачи.

Расширенная поддержка сетей TCP/IP (IPv4/6, SNMP, коммутация и маршрутизация, фильтрация IP, виртуальные сети, безопасные беспроводные сети).

Система построена по технологии FLEET: Fault-tolerance(отказоустойчивая), Load-balancing (регулирующая нагрузку), Efficient (эффективная), Extensible (расширяемая), Transparent (прозрачная).

Операционная система QNX объединяет всю сеть персональных компьютеров в единый набор ресурсов с абсолютной прозрачностью доступа к ним. Узлы могут добавляться и исключаться из сети, не влияя на целостность системы.

Сетевая обработка данных в QNX является настолько гибкой, что вы можете объединить в одну сеть любой разнородный набор Intel совместимых компьютеров соединенных через Arcnet, Ethernet, Token Ring или через последовательный порт, к которому также может быть подключен модем. Причем возможно участие компьютера одновременно в нескольких сетях, и если одна из них окажется перегруженной или выйдет из строя, то QNX автоматически будет использовать другие доступные сети без потери информации.

QNX имеет архитектуру, которая абсолютно отличается от архитектуры UNIX, но эта операционная система реализует программный интерфейс POSIX.

POSIX 1003.1 — **базовый стандарт** определяет программный интерфейс для систем управления процессами, ввода/вывода данных в устройствах и файловых системах, а также общего межзадачного взаимодействия.

**Расширения реального времени** (Realtime Extensions) — этот стандарт определяет набор дополнительных расширений реального времени к базовому стандарту 1003.1.

**Управление потоками** (Threads) — это еще одно расширение среды стандартов POSIX.

**Дополнительные расширения реального времени** (Additional Realtime Extensions) — этот стандарт определяет такие функции, как подключение обработчиков прерываний.

**Профили прикладных окружений** (Application Environment Profiles) — этот стандарт определяет несколько профилей среды POSIX (Realtime AEP, Embedded Systems AEP и др.), соответствующих разным наборам встраиваемых функций.

Файловая система QNX полностью соответствует стандарту POSIX. Программист, поработавший в UNIX, не заметит никаких отличий в работе. Хотя внутри файловая система значительно отличается от таковой в UNIX. Эти отличия в основном сказываются на ее живучести, то есть на целостности данных, хранимых на диске, и на производительности.

QNX обеспечивает работу с различными типами файловых систем: POSIX, Embedded (FLASH, ROM, SROM), CD-ROM (с поддержкой стандарта ISO 9660 и его расширения Rock Ridge), DOS (доступ ко всем носителям информации в формате DOS), NFS (доступ к различным типам удаленных файловых систем), SMB (прозрачный доступ к Windows 95 или NT-серверам).

Для QNX разработано множество баз данных (db\_Vista, Watcom SQL, Faircom C-tree, OnCmd и др.), которые по производительности часто превосходят аналоги под управлением других операционных систем.

Вряд ли какая-нибудь операционная система сможет посоревноваться с QNX по количеству различных графических интерфейсов. В ограниченной в ресурсах встраиваемой системе QNX может использоватьо Photon — компактный (256К) оконный пакет, поддерживающий стандарт Motif. Photon — это принципиально новая оконная графическая система, которая по своему подходу к реализации графического интерфейса коренным образом отличается от всех существовавших ранее систем.

Поддерживаемые QNX процессоры: х86/IA, ARM, Xscale, SH-4, MIPS, PowerPC.

Поддерживаются многопроцессорность (SMP/AMP/BMP), многоядерные процессоры, распределённые вычисления, виртуализация.

* Асимметричная многопроцессорность (AMP): каждая ОС или экземпляр одной ОС работает на отдельном процессоре.
* Симметричная многопроцессорность (SMP): один экземпляр ОС управляет всеми процессорами одновременно и приложения могут работать на любом из них.
* C привязкой к заданным процессорам (BMP): один экземпляр ОС управляет всеми процессорами, но каждое приложение привязано к определенному процессору.

**Применения QNX**: ответственные системы, промышленные и военные приложения, транспорт, сети/ телекоммуникации.

**Поддерживаемые языки**: C/C++, Java (IBM WebSphere Studio Device Developer), UML (IBM Rational Rhapsody), Python, Ruby, Fortran.

**Инструментарий разработчика**: штатная интегрированная кросс-среда на базе Eclipse (Windows). Компилятор GNU C/C++ или Intel C++ Compiler, библиотеки GNU и Dinkum, встраиваемый C++. Символьный кросс-отладчик.

Встроенная система управления версиями. Средства верификации и диагностики: анализатор покрытия кода, профайлер, монитор целевых систем, анализатор ОЗУ, системный профайлер.

Визуальный построитель встраиваемых конфигураций.

Разработчики могут легко изменять конфигурацию QNX в диапазоне от минимальной конфигурации микроядра с несколькими небольшими модулями до полнофункциональной сетевой системы, предназначенной для обслуживания сотен пользователей.

ОС QNX идеально подходит для встраиваемых приложений реального времени.

Она может быть масштабирована до самых компактных конфигураций и способна работать в многозадачном режиме, управлять потоками, осуществлять планирование процессов по приоритетам (priority-driven preemptive scheduling) и выполнять быстрое переключение контекстов (fast context-switching).

Разработка приложения QNX может легко выполняться на настольной системе на основе резидентной модели разработки.

Даже при использовании кросс-платформенной модели разработки программный интерфейс (API) остается принципиально тем же самым. Какая бы платформа (QNX Neutrino, Solaris, Windows и т. д.) или целевой процессор (x86, ARM, MIPS, PowerPC и др.) ни применялись, программисту не нужно беспокоиться, например, о проблемах следования байтов (endian), выравнивания (alignment) или ввода/вывода данных.

Уникальные возможности эффективности, модульности и простоты достигаются в ОС QNX благодаря двум фундаментальным принципам:

* микроядерная архитектура;
* межзадачное взаимодействие на основе обмена сообщениями.

**Типы лицензий QNX6**

* **Коммерческая** лицензия предполагает покупку ОСРВ QNX6 у дистрибьютора. Это необходимо делать в случае коммерческой разработки устройств или ПО на базе QNX.
* **Академическая** лицензия предоставляется вузам для обучения студентов и работы преподавателей.
* **Пробная лицензия** (на 30 дней) позволяет ознакомиться с полнофункциональной версией QNX и средой разработки, чтобы оценить целесообразность приобретения ОСРВ.
* **Некоммерческая** лицензия позволяет работать с QNX и вести некоммерческую разработку.

Исходный текст доступен полностью.

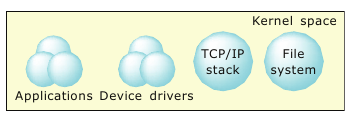
## Февраль, ЛК4. Микроядро QNX Neutrino

**Микроядерная операционная система** построена на основе миниатюрного ядра, обеспечивающего минимальные службы для произвольной группы взаимодействующих процессов, которые, в свою очередь, обеспечивают функциональность более высокого уровня.

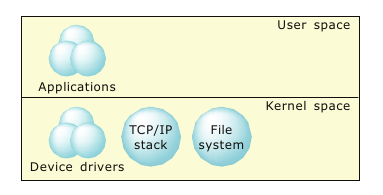
Первичным здесь становится модульный принцип, тогда как стремление к компактности является вторичным.

В результате ОС становится «открытой» и легко наращиваемой. Более того, расширения ОС, написанные пользователем, не влияют на надежность ядра ОС.

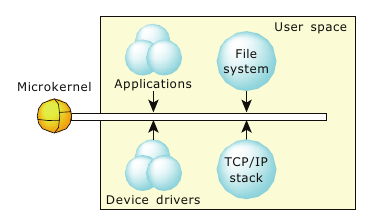
QNX Neutrino использует возможности работы с MMU-оборудованием для обеспечения полнофункциональной POSIX-модели в защищенной среде.



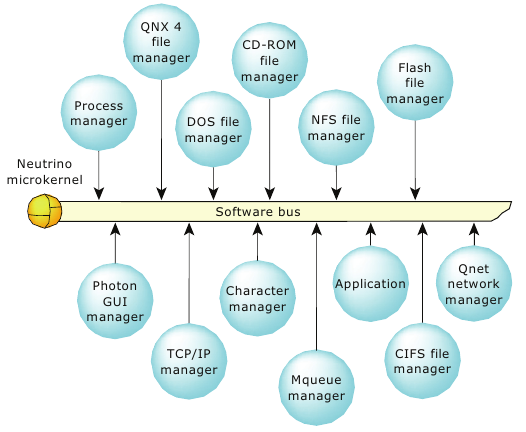
Обычные исполнительные модули реального времени не обеспечивают защиту памяти.



В монолитной ОС системные процессы не защищены.



Микроядро обеспечивает полную защиту памяти.



ОС QNX Neutrino действует как своего рода «программная шина», позволяющая динамически присоединять/отсоединять модули ОС по мере необходимости.

Большинство служб по обеспечению работы в реальном масштабе времени и по планированию потоков реализуется прямо в микроядре, эти службы могут работать даже без дополнительных модулей ОС.

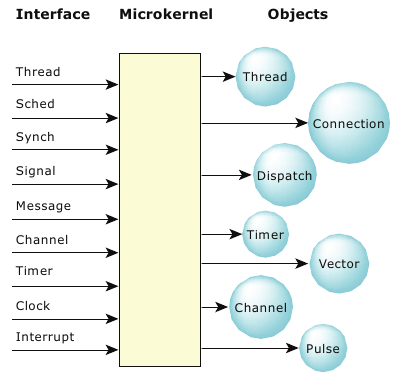
Микроядро в ОС QNX Neutrino — очень компактно и предназначено для выполнения только нескольких базовых функций.

* управление потоками (thread services) посредством POSIX-примитивов для создания потоков;
* управление сигналами (signal services) посредством примитивов сигналов;
* обмен сообщениями (message-passing services), с помощью которого микроядро выполняет трассировку всех сообщений, пересылаемых между всеми потоками внутри системы;
* синхронизация (synchronization services) посредством примитивов синхронизации потоков;
* планирование (scheduling services), с помощью которого микроядро осуществляет планирование потоков в реальном времени на основе различных алгоритмов;
* управление таймерами (timer services), с помощью которого микроядро обеспечивает большой набор POSIX-таймеров;
* управление процессами (process management services), выполняемое администратором процессов, вместе с которым ядро образует единый модуль procnto. Администратор процессов предназначен для управления процессами, памятью и пространством имен путей (pathname space).

На самых нижних уровнях микроядра расположено всего несколько базовых объектов, а также четко отрегулированные процедуры для управления ими.

Микроядро ОС QNX Neutrino реализовано преимущественно на C. Компактность и высокая производительность достигаются с помощью четко отлаженных алгоритмов и структур данных, а не посредством оптимизации на уровне ассемблерного кода.

В микроядро были включены те службы, которые порождали наиболее короткую ветвь исполняемого кода. Операции, требующие значительных ресурсов (например, загрузка процесса), были переданы внешним процессам и потокам, в которых работа по переключению на контекст другого потока пренебрежительно мала в сравнении с работой по обработке запроса, выполняемой внутри этого потока.



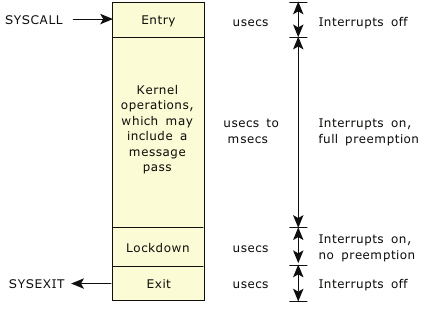
В микроядре ОС QNX Neutrino существуют системные вызовы (kernel calls), которые служат для управления следующими объектами:

* потоками (threads);
* сообщениями (message passing);
* сигналами (signals);
* часами (clocks);
* таймерами (timers);
* обработчиками прерываний (interrupt handlers);
* семафорами (seamphores);
* блокировками взаимного исключения, или мутексами (mutexes);

Все службы ОС, за исключением тех, которые выполняются обязательным модулем микроядра/администратора процессов (procnto), обрабатываются посредством стандартных процессов. Система может содержать следующие компоненты:

* администраторы файловых систем;
* администраторы устройств символьного ввода/вывода;
* графический пользовательский интерфейс (Photon);
* сетевой администратор;
* стек протоколов TCP/IP.

В отличие от потоков, микроядро никогда не планируется на выполнение. Процессор выполняет код в микроядре только в случае явного вызова ядра, при возникновении исключения или в результате аппаратного прерывания.



Запрет прерываний или отсутствие вытеснения происходит только на короткие периоды времени (как правило, эти периоды не превышают порядка сотен наносекунд).

Системные процессы по сути никак неотличимы от пользовательских процессов — они используют те же самые унифицированные службы программного интерфейса и ядра, которые доступны для любого пользовательского процесса, имеющего соответствующие привилегии.

Именно такая архитектура дает ОС QNX Neutrino уникальные возможности расширяемости. Поскольку большинство служб ОС выполняются стандартными системными процессами, конфигурация ОС может быть легко дополнена новыми компонентами, для чего достаточно написать соответствующие программы, предназначенные для выполнения новых служб.

Если в большинстве ОС драйверы устройств должны быть встроены в структуру ОС, в QNX Neutrino они могут запускаться и останавливаться как стандартные процессы. В результате добавление новых драйверов устройств не нарушает работу какой-либо части ОС, так как драйверы могут разрабатываться и подвергаться отладке как обычные приложения.

## Потоки и процессы

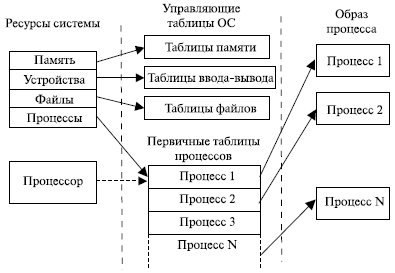
**Мультипрограммирование** (многозадачность, multitasking) – это такой способ организации вычислительного процесса, при котором на одном процессоре попеременно выполняются несколько программ. Чтобы поддерживать мультипрограммирование, ОС должна определить для себя внутренние единицы работы, между которыми будут разделяться процессор и другие ресурсы компьютера.

В настоящее время в большинстве операционных систем определены два типа единиц работы: более крупная единица – процесс, или задача, и менее крупная – поток, или нить. Причем процесс выполняется в форме одного или нескольких потоков.

**Мультипрограммирование** призвано повысить эффективность использования вычислительной системы. Однако эффективность может пониматься по-разному.

Наиболее характерными показателями эффективности вычислительных систем являются:

* пропускная способность – количество задач, выполняемых системой в единицу времени;
* удобство работы пользователей, заключающееся, в частности, в том, что они могут одновременно работать в интерактивном режиме с несколькими приложениями на одной машине;
* реактивность системы – способность выдерживать заранее заданные (возможно, очень короткие) интервалы времени между запуском программы и получением конечного результата.



Для управления процессами и ресурсами компьютера ОС должна располагать информацией о текущем состоянии каждого процесса и ресурса. Универсальный подход к предоставлению такой информации заключается в создании и поддержке таблиц с информацией по каждому объекту управления.

Поток можно рассматривать как минимальную «единицу выполнения», т. е. единицу планирования (scheduling) и выполнения в микроядре. Процесс, в свою очередь, можно рассматривать как объект, который содержит в себе эти потоки и который определяет для их выполнения свое «адресное пространство». В простейшем случае процесс состоит из одного потока, и именно таким образом трактовалось понятие «процесс» до середины 80-х годов (например, в ранних версиях UNIX).

ОС QNX Neutrino можно конфигурировать определенным образом для реализации некоторого набора потоков и процессов (в соответствии со стандартами POSIX).

Все процессы отделены друг от друга с помощью блока управления памятью (Memory Management Unit, MMU), и каждый процесс может содержать один или несколько потоков, использующих адресное пространство процесса.

**Подсистема управления процессами и потоками** мультипрограммной ОС выполняет функции:

создание процессов и потоков;

* обеспечение процессов и потоков необходимыми ресурсами;
* изоляция процессов;
* планирование выполнения процессов и потоков;
* диспетчеризация потоков;
* организация межпроцессного взаимодействия;
* синхронизация процессов и потоков;
* завершение и уничтожение процессов и потоков.

К созданию процесса приводят пять основных событий:

* инициализация ОС (загрузка);
* выполнение запроса работающего процесса на создание процесса;
* запрос пользователя на создание процесса, например, при входе в систему в интерактивном режиме;
* инициирование пакетного задания;
* создание операционной системой процесса, необходимого для работы каких-либо служб.

Обычно при загрузке ОС создаются несколько процессов. Некоторые из них являются высокоприоритетными процессами, обеспечивающими взаимодействие с пользователями и выполняющими заданную работу. Остальные процессы являются фоновыми, их называют демонами.

Переход от выполнения одного потока к другому осуществляется в результате планирования и диспетчеризации. Работа по определению момента, в который необходимо прервать выполнение текущего потока, и потока, которому следует предоставить возможность выполняться, называется планированием. Планирование потоков осуществляется на основе информации, хранящейся в описателях процессов и потоков. При планировании принимается во внимание приоритет потоков, время их ожидания в очереди, накопленное время выполнения, интенсивность обращения к вводу-выводу и другие факторы.

Диспетчеризация заключается в реализации найденного в результате планирования решения, т.е. в переключении процессора с одного потока на другой. Диспетчеризация проходит в три этапа:

* сохранение контекста текущего потока;
* загрузка контекста потока, выбранного в результате планирования;
* запуск нового потока на выполнение.

Процесс завершается благодаря одному из следующих событий:

* обычный выход,
* выход по ошибке,
* выход по неисправимой ошибке,
* уничтожение другим процессом.

Подсистема управления процессами закрывает все файлы, с которыми работал процесс, освобождает области оперативной памяти, отведенные под коды, данные и системные информационные структуры процесса.

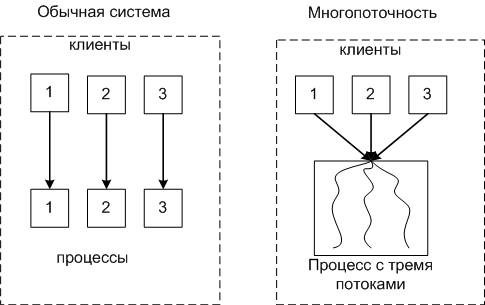
Выполняется коррекция всевозможных очередей ОС и списка ресурсов, в которых имелись ссылки на завершаемый процесс.

Современные ОС предлагают механизм многопоточной обработки (multithreading).

Понятию «поток» соответствует последовательный переход процессора от одной команды к другой. Процессу ОС назначают адресное пространство и набор ресурсов, которые совместно используются всеми его потоками. Все потоки одного процесса всегда принадлежат одному приложению, поэтому ОС изолирует потоки в гораздо меньшей степени, чем процессы. Все потоки одного процесса используют общие файлы, таймеры, устройства, одну и ту же область оперативной памяти, одно и то же адресное пространство.

Между потоками одного процесса нет полной защиты, во-первых, потому что это невозможно, а во-вторых, потому что не нужно.

Таким образом, мультипрограммирование более эффективно на уровне потоков, а не процессов.



Сравнение многопоточной системы с однопоточной

С каждым потоком связывается:

* Счетчик выполнения команд
* Регистры для текущих переменных
* Стек
* Состояние

Потоки делят между собой элементы своего процесса:

* Адресное пространство
* Глобальные переменные
* Открытые файлы
* Таймеры
* Семафоры
* Статистическую информацию.

В остальном модель идентична модели процессов.

При управлении процессами ОС использует два основных типа информационных структур: блок управления процессом (дескриптор процесса) и контекст процесса. Дескрипторы процессов объединяются в таблицу процессов, которая размещается в области ядра. На основании информации, содержащейся в таблице процессов, ОС осуществляет планирование и синхронизацию процессов.

В дескрипторе (блоке управления) процесса содержится такая информация о процессе, которая необходима ядру в течение всего жизненного цикла процесса:

* информация по идентификации процесса;
* информация по состоянию процесса;
* информация, используемая при управлении процессом.

Каждому процессу присваивается числовой идентификатор, который может быть просто индексом в первичной таблице процессов.

Информация по состоянию и управлению процессом включает данные:

* состояние процесса, определяющее готовность процесса к выполнению (выполняющийся, готовый к выполнению, ожидающий какого-либо события, приостановленный);
* данные о приоритете (текущий приоритет, по умолчанию, максимально возможный);
* информация о событиях – идентификация события, наступление которого позволит продолжить выполнение процесса;
* указатели, позволяющие определить расположение образа процесса в оперативной памяти и на диске;
* указатели на другие процессы (в частности, находящиеся в очереди на выполнение);

Информация по состоянию и управлению процессом (продолжение):

* флаги, сигналы и сообщения, имеющие отношение к обмену информацией между двумя независимыми процессами;
* данные о привилегиях, определяющих права доступа к определенной области памяти или возможности выполнять определенные виды команд;
* указатели на ресурсы, которыми управляет процесс (например, перечень открытых файлов);
* сведения по истории использования ресурсов и процессора;
* информация, связанная с планированием. Эта информация во многом зависит от алгоритма планирования.

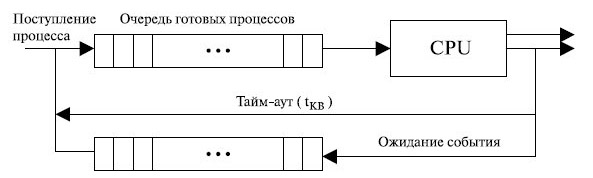
Контекст процесса содержит информацию, позволяющую системе приостанавливать и возобновлять выполнение процесса с прерванного места.

В контексте процесса содержится следующие основные данные:

* содержимое регистров процессора, доступных пользователю;
* содержимое счетчика команд;
* состояние управляющих регистров и регистров состояния;
* коды условий, отражающие результат выполнения последней арифметической или логической операции (например, знак равенства нулю, переполнения);
* указатели вершин стеков, хранящие параметры и адреса вызова процедур и системных служб.



Состояния процесса

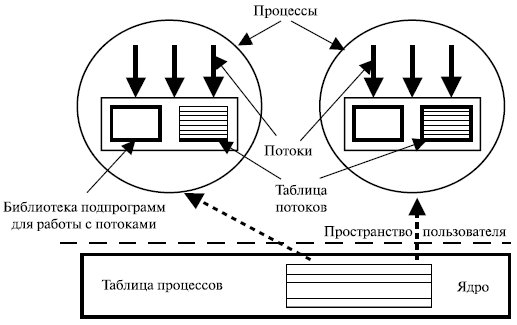


Очереди процессов

Есть два способа реализации пакета потоков:

* в пространстве пользователя или на уровне пользователя (User-level threads – ULT);
* в ядре или на уровне ядра (kernel-level threads – KLT).

В программе, полностью состоящей из ULT-потоков, все действия по управлению потоками выполняются самим приложением. Ядро о потоках ничего не знает и управляет обычными однопоточными процессами.



Преимущества ULT-потоков:

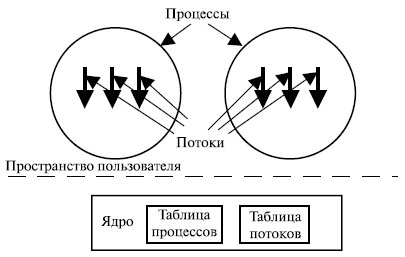
* Такую многопоточность можно реализовать на ядре, не поддерживающим многопоточность.
* Более быстрое переключение, создание и завершение потоков.
* Процесс может иметь собственный алгоритм планирования.

Недостатки ULT-потоков:

* Отсутствие прерывания по таймеру внутри одного процесса.
* При использовании блокирующего системного запроса (процесс переводится в режим ожидания, например: чтение с клавиатуры, а данные не поступают) все остальные потоки блокируются.
* Сложность реализации.

Если потоки реализованы на уровне ядра (KLT), в области приложения система поддержки исполнения программ не нужна, нет необходимости и в таблицах потоков в каждом процессе. Вместо этого есть единая таблица потоков, отслеживающая все потоки в системе.

Когда потоку необходимо создать новый поток или завершить имеющийся, он выполняет запрос ядру, который создает или завершает поток, внося изменения в таблицу потоков.



Любое приложение можно запрограммировать как многопоточное, при этом все потоки приложения поддерживаются в рамках единого процесса. Ядро владеет информацией контекста процесса как единого целого, а также контекстами каждого отдельного потока процесса. Планирование осуществляется ядром, исходя из состояния потоков. С помощью такого подхода удается избавиться от основных недостатков потоков пользовательского уровня.

Возможно планирование работы нескольких потоков одного и того же процесса на нескольких процессорах:

* реализуется мультипрограммирование в режимах нескольких процессов (вообще – всех);
* при блокировке одного из потоков процесса ядро может выбрать для выполнения другой поток этого же процесса;
* процедуры ядра могут быть многопоточными.

Главный **недостаток** связан с необходимостью двукратного переключения режимов пользовательский – ядро, ядро – пользовательский для передачи одного потока к другому в рамках одного и того же процесса.

**Запуск процесса в QNX/Neutrino**

Функции, которые QNX/Neutrino обеспечивает для запуска других процессов (или подмены одного процесса другим):

* system();
* семейство функций exec();
* семейство функций spawn();
* fork();
* vfork().

Функция **system()** — самая простая функция; она получает на вход одну командную строку.

Семейство функций **exec()** подменяет текущий процесс другим. Идентификатор процесса (PID) при этом не меняется, просто процесс преобразуется в другую программу.

Вызов функции семейства **spawn()** создает другой процесс (с новым идентификатором), который соответствует программе, указанной в аргументах функции.

Функция **fork()** просто копирует текущий процесс. У результирующего процесса как код, так и данные полностью совпадают с таковыми для родительского процесса.

Функция **vfork()** позволяет сэкономить на ресурсах, поскольку она делает разделяемым адресное пространство родителя. Она приостанавливает родительский поток до тех пор, пока «сын» не вызовет функцию exec() или не завершится.

**Создание процесса и потоки в QNX/Neutrino**

Предположим, что у вас есть процесс, и вы еще не создали никаких потоков. Если вызвать функцию **fork()**, то будет создан другой процесс, и тоже с одним потоком.

Теперь предположим, что в вашем процессе вы вызвали pthread\_create() для создания другого потока. Если вы теперь вызовете функцию fork(), она возвратит ENOSYS (функция не поддерживается). Си-библиотека QNX/Neutrino не рассчитана на ветвление процесса с потоками.

Функция **vfork()** задержит выполнение до тех пор, пока дочерний процесс не вызовет **exec()** или не завершится, тогда как семейство **spawn()** может позволить работать обоим процессам одновременно.

**Запуск потока в QNX/Neutrino**

Любой поток может создать другой поток в том же самом процессе; на это не налагается никаких ограничений (за исключением объема памяти). Наиболее общий путь — использование вызова функций POSIX pthread\_create(), которая имеет четыре аргумента:

* thread — указатель на pthread\_t, где хранится идентификатор потока;
* attr — атрибутная запись;
* start\_routine — подпрограмма, с которой начинается поток;
* arg — параметр, который передается подпрограмме start\_routine().

Указатель thread и атрибутная запись (attr) — необязательные элементы, можно передавать вместо них NULL.

Новый поток начинает выполнение с функции start\_routine(), с параметром arg.

**Атрибуты потока в QNX/Neutrino**

Хотя потоки внутри процесса и используют совместно одно общее адресное пространство этого процесса, каждый из этих потоков имеет некоторые «собственные» данные. В некоторых случаях эти данные защищаются внутри ядра (например, идентификатор потока tid), в то время как другие данные остаются незащищенными в адресном пространстве процесса (например, каждый поток имеет свой собственный стек).

* **идентификатор потока** (tread identifier, tid) — каждый поток обозначается своим собственным целочисленным идентификатором, начиная с 1. Внутри процесса потоки имеют уникальные идентификаторы;
* **приоритет** (priority) — у каждого потока есть приоритет, который определяет, когда поток работает. Поток наследует приоритет от родительского потока, но приоритет может измениться, в зависимости от алгоритма планирования, определенных изменений, производимых потоком, или сообщений, переданных потоку. Процессы Neutrino не имеют такого параметра, как значение приоритета.

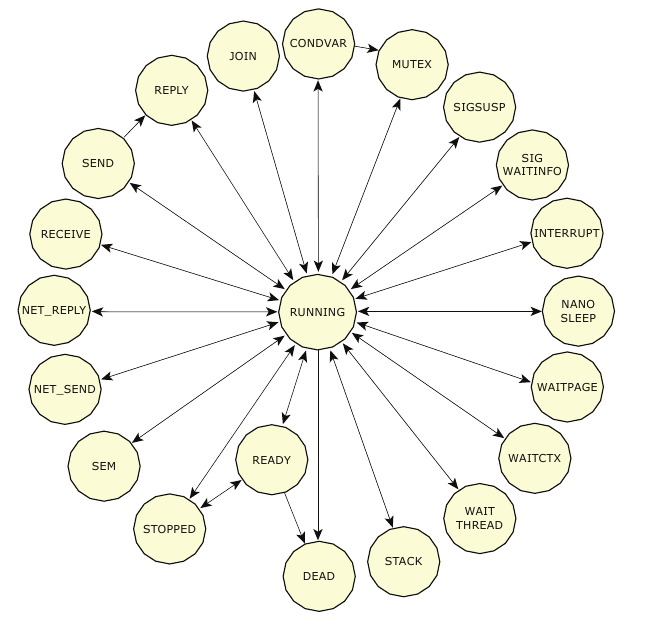
**Атрибуты потока в QNX/Neutrino**

* **имя** (name) — начиная с QNX Neutrino версии 6.3.2 можно задавать имя потоку; см. разделы pthread\_getname\_np() и pthread\_setname\_np() в Neutrino Library Reference. Такие утилиты как dumper и pidin поддерживают имена потоков. Имена потоков — это расширение Neutrino.
* **набор регистров** (register set) — каждый поток имеет свой указатель команд (Instruction Pointer, IP), указатель стека (Stack Pointer, SP) и другие регистры, составляющие контекст потока;
* **стек** (stack) — каждый поток имеет свой собственный стек, хранимый в адресном пространстве процесса, к которому поток относится;
* **маска сигналов** (signal mask) — каждый поток имеет свою собственную маску сигналов;

**Атрибуты потока в QNX/Neutrino**

* **локальная память потока** (ЛПП) (Thread Local Storage, TLS) — это системная область данных потока. Локальная память потока используется для хранения информации, относящейся к каждому отдельному потоку (например, tid, pid, базовый адрес стека (stack base), errno, привязка ключей и данных, относящихся к потоку). К локальной памяти потока не нужно обращаться напрямую из пользовательского приложения. Поток может использовать содержимое ЛПП как ключ для привязывания пользовательских данных;
* **обработчик завершений** (cancellation handler) — содержит функции обратного вызова, выполняемые при завершении потока.

Данные, относящие к потоку, реализуются в библиотеке *pthread* и хранятся в локальной памяти потока.



**Жизненный цикл потока в QNX/Neutrino**

Потоки создаются и уничтожаются динамически, их количество внутри одного процесса может изменяться в значительных пределах. В дополнение к показанным переходам, поток может перейти из любого состояния (кроме DEAD) в состояние READY.

**RUNNING** – поток выполняется процессором;

**READY** – поток ожидает выполнения, пока процессор занят выполнением другого потока равного или более высокого приоритета;

**CONDVAR** – поток блокирован на условной переменной (например, при вызове функции pthread\_condvar\_wait());

**INTERRUPT** – поток блокирован и ожидает прерывания (т. е. поток вызвал функцию InterruptWait());

**JOIN** – поток блокирован и ожидает завершения другого потока (например, при вызове функции pthread\_join());

**MUTEX** – поток блокирован блокировкой взаимного исключения (например, при вызове функции pthread\_mutex\_lock());

**NANOSLEEP** – поток находится в режиме ожидания в течение короткого периода времени (например, при вызове функции nanosleep());

**NET\_REPLY** – поток ожидает ответ на сообщение от другого узла сети (т.е. поток вызвал функцию MsgReply\*());

**NET\_SEND** – поток ожидает получения импульса или сигнала от другого узла сети (т. е. поток вызвал функцию MsgSendPulse(), MsgDeliverEvent() или SignalKill());

**RECEIVE** – поток блокирован на операции получении сообщения (например, при вызове функции MsgReceive());

**REPLY** – поток блокирован при ответе на сообщение (т. е. при вызове функции MsgSend() и получении сообщения сервером);

**SEM** – поток ожидает освобождения семафора (т. е. поток вызвал функцию SyncSemWait());

**DEAD** – поток завершен и ожидает завершения другого потока;

**SEND** – поток блокируется при отправке сообщения (т. е. после того, как поток вызвал функцию MsgSend(), но получения сообщения сервером еще не произошло);

**SIGSUSPEND** – поток блокирован и ожидает сигнал (т. е. поток вызвал функцию sigsuspend());

**SIGWAITINFO** – поток блокирован и ожидает сигнал (т. е. поток вызвал функцию sigwaitinfo());

**STACK** – поток ожидает выделения виртуального адресного пространства для своего стека (родительский поток вызывает функцию ThreadCreate());

**STOPPED** – поток блокирован и ожидает сигнал SIGCONT;

**WAITCTX** – поток ожидает доступности нецелочисленного контекста (например, с плавающей запятой);

**WAITPAGE** – поток ожидает выделения физической памяти для виртуального адреса;

**WAITTHREAD** – поток ожидает завершения создания дочернего потока (т. е. поток вызвал функцию ThreadCreate()).

## Март, ЛК1. Планирование потоков

При каждом вызове ядра, исключении или аппаратном прерывании, в результате которого управление передается ядру QNX, выполнение активного потока временно приостанавливается. Действие планирования совершается в момент изменения состояния любого потока независимо от того, в каком процессе поток расположен. Планирование потоков осуществляется по всем процессам сразу.

Как правило, выполнение приостановленного потока через некоторое время возобновляется. При этом планировщик (scheduler) выполняет переключение контекстов с одного потока на другой всякий раз, когда активный поток:

* блокируется (blocked);
* вытесняется (preempted);
* отдает управление (yields).

**Когда поток блокируется**

Активный поток блокируется, если он должен ожидать какое-либо событие (например, ответ на запрос механизма обмена сообщениями, освобождение мутекса и т. д.). Блокированный поток удаляется из очереди готовности (ready queue), после чего запускается поток с наивысшим приоритетом.

Когда блокированный поток разблокируется, он помещается в конец очереди готовности на соответствующий приоритетный уровень.

**Когда поток вытесняется**

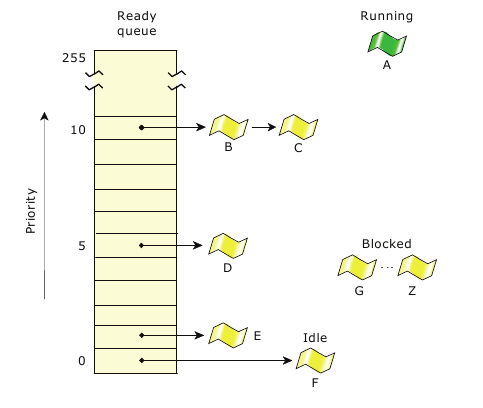
Активный поток вытесняется, когда поток с более высоким приоритетом помещается в очередь готовности (т. е. он переходит в состояние готовности (READY) в результате снятия условия блокировки).

Прерванный поток остается на соответствующем приоритетном уровне в начале очереди готовности, а поток с более высоким приоритетом начинает выполняться.

**Когда поток отдает управление**

Активный поток самостоятельно освобождает процессор (sched\_yield()) и помещается в конец очереди готовности на данном уровне приоритета.

После этого запускается поток с наивысшим приоритетом (в том числе им может быть поток, который только что отдал управление).



**Планирование и приоритеты**

Каждому потоку назначается свой приоритет. Планировщик выбирает поток для выполнения в соответствии с приоритетом каждого потока, находящегося в состоянии готовности (READY).

Всего в ОС QNX Neutrino поддерживается до **256 уровней приоритетов**.

Приоритет выполнения каждого непривилегированного (nonroot) потока может изменяться в пределах от 1 до 63 (наивысший приоритет), независимо от его дисциплины планирования (scheduling policy).

Только привилегированные (root) потоки (т. е. потоки, действующий uid которых равен 0) могут иметь приоритет выше 63.

Специальный поток с именем idle в администраторе процессов, имеет приоритет 0 и всегда готов к выполнению.

По умолчанию поток наследует приоритет своего родительского потока.

Потоки, находящиеся в очереди, упорядочиваются по приоритету. В действительности очередь готовых потоков состоит из 256 отдельных очередей — по одной на каждый приоритет. Потоки выстраиваются в очереди по порядку поступления (FIFO) и в соответствии с их приоритетом. Для выполнения выбирается первый поток с наивысшим приоритетом.

Большую часть времени потоки выстраиваются в очереди по порядку поступления (FIFO) и в соответствии с их приоритетом, но есть исключения:

* Поток сервера, выходящего из состояния блокировки по RECEIVE по получению сообщения от клиента, помещается в начале очереди для данного приоритета — то есть в порядке обратном порядку поступления (LIFO).
* Если поток отправляет сообщение с помощью “nc” (non-cancellation point) варианта MsgSend\*(), когда сервер отвечает, поток помещается в начале очереди готовности, а не в конце. Если алгоритм планирования циклический, квант времени потока не будет восстановлен; например, если поток уже израсходовал половину своего кванта времени перед отправкой, то только половина кванта останется до вытеснения.

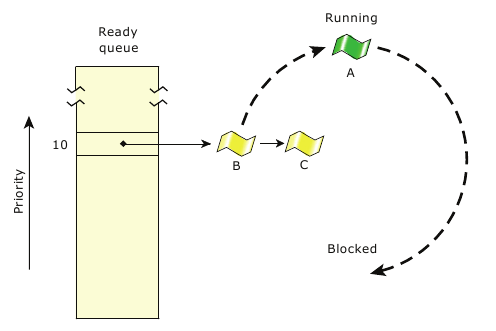
**Алгоритмы планирования**

Для работы с различными приложениями в ОС QNX Neutrino используются следующие алгоритмы планирования:

* FIFO-планирование;
* циклическое планирование (планирование по круговому алгоритму) (round-robin scheduling);
* спорадическое планирование (sporadic scheduling).

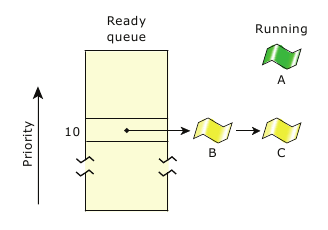
Каждый поток в системе может выполняться по любому из методов. Методы применяются для каждого отдельного потока, а не для всех потоков или процессов одновременно.

Хотя поток наследует алгоритм планирования от своего родительского процесса, он может сделать запрос на его изменение.



FIFO-планирование и циклическое планирование применяются только в случаях, когда два или более потоков, имеющих одинаковый приоритет, находятся в состоянии готовности (READY) (т. е. в этом случае потоки напрямую конкурируют между собой за использование процессора).

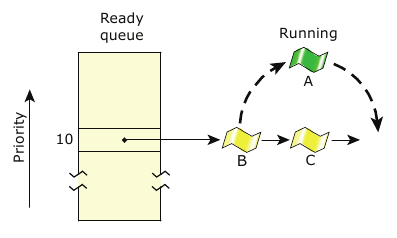
Когда поток с более высоким приоритетом переходит в состояние READY, он вытесняет (preempts) все другие потоки с более низким приоритетом.



**FIFO-планирование**

В алгоритме FIFO-планирования поток продолжает выполняться до тех пор, пока он:

* принудительно не освободит ресурсы управления (например, при блокировке);
* не будет вытеснен потоком с более высоким приоритетом.



**Циклическое планирование**

В алгоритме планирования циклического типа поток продолжает выполняться до тех пор, пока он:

* принудительно не освободит ресурсы управления;
* не будет вытеснен потоком с более высоким приоритетом;
* не израсходует свой квант времени (timeslice).

Квант времени определяется как 4 × тактовый интервал (clock period).

**Спорадическое планирование**

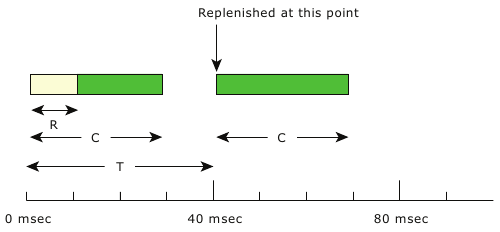
Алгоритм спорадического планирования обычно используется для задания верхнего лимита на время выполнения потока в пределах заданного периода времени. Этот метод необходим при выполнении монотонного частотного анализа (Rate Monotonic Analysis) системы, обслуживающей как периодические, так и апериодические события. По сути, данный алгоритм позволяет потоку обслуживать апериодические события, не препятствуя своевременному выполнению других потоков или процессов в системе.

Как и в FIFO-планировании, поток, для которого применяется спорадическое планирование, выполняется до тех пор, пока он не блокируется или прерывается потоком с более высоким приоритетом. Кроме того, так же как и в адаптивном планировании, поток, для которого применяется спорадическое планирование, получает пониженный приоритет. Однако спорадическое планирование дает значительно более точное управление потоком.

При спорадическом планировании приоритет потока может динамически изменяться между приоритетом переднего плана (foreground) (или нормальным приоритетом) и фоновым (background) (или пониженным) приоритетом. Для управления этим спорадическим переходом используются следующие параметры:

* **начальный бюджет потока** (initial budget) (C) —время, за которое поток может выполняться с нормальным приоритетом (N), перед тем как получить пониженный приоритет (L);
* **пониженный приоритет** (low priority) (L) — приоритетный уровень, до которого приоритет потока будет снижен. При пониженном приоритете (L) поток выполняется в фоновом режиме. Если поток имеет нормальный приоритет (N), он выполняется с приоритетом переднего плана;
* **период пополнения** (replenishment period) (T) — период времени, в течение которого поток может расходовать свой бюджет выполнения (execution budget).
* **максимальное число текущих пополнений** (max number of pending replenishments) — это значение устанавливает ограничение на количество выполняемых операций пополнения, тем самым ограничивая объем системных ресурсов, выделяемых на дисциплину спорадического планирования.

Алгоритм спорадического планирования устанавливает начальный бюджет выполнения потока (C), который расходуется потоком в процессе его выполнения и пополняется с периодичностью, определенной параметром T. Когда поток блокируется, израсходованная часть бюджета выполнения потока (R) пополняется через какое-то установленное время (например, через 40 мс), отсчитываемое от момента, когда поток перешел в состояние готовности.



При нормальном приоритете (N) поток выполняется в течение периода времени, установленного его начальным бюджетом выполнения (C). После того как этот период истекает, приоритет потока снижается до пониженного уровня (L) до тех пор, пока не произойдет операция пополнения.

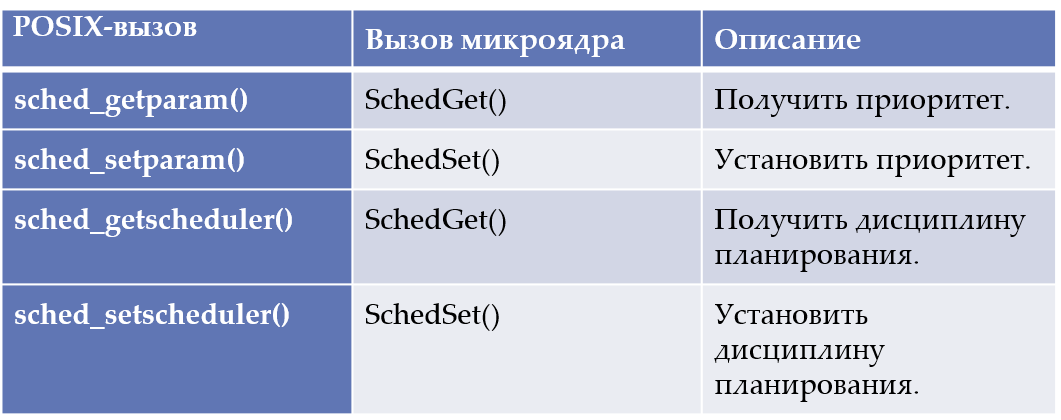
Как только происходит пополнение, приоритет потока повышается до начального уровня. Таким образом, в правильно настроенной системе поток выполняется каждый период времени Т в течение максимального времени С. Это обеспечивает такой порядок, при котором каждый поток, выполняемый с приоритетом N, будет использовать только C/T процентов системных ресурсов.

**Управление приоритетами и алгоритмами планирования**

Во время выполнения потока его приоритет может изменяться либо в результате прямого действия самого потока, либо в результате вмешательства ядра при получении сообщения от какого-либо потока с более высоким приоритетом.

В дополнение к приоритету можно также изменять алгоритм планирования, применяемый ядром для потока. Хотя библиотеки предоставляют множество различных путей получения и установки параметров алгоритма планирования, лучшим вариантом будут pthread\_getschedparam(), pthread\_setschedparam() и pthread\_setschedprio().

Изменяться может не только приоритет, но и алгоритм планирования, применяемый ядром для выполнения потока.



Изначально потоки возникли в UNIX-системах как «легковесный» механизм управления параллелизмом, предназначенный для решения проблемы слишком медленного переключения контекстов между «тяжеловесными» процессами.

С точки зрения архитектуры, ОС QNX Neutrino в первую очередь решает вопрос производительности переключения контекстов. В действительности, переключения контекстов между потоками и между процессами происходят с почти одинаковой скоростью. Однако в QNX Neutrino скорость переключения межзадачных контекстов выше, чем скорость переключения контекстов между потоками в UNIX. В результате в ОС QNX Neutrino отпадает необходимость использовать потоки для решения проблем производительности межзадачного взаимодействия. Вместо этого они служат в качестве инструментов для достижения более высокой степени параллелизма внутри прикладных и серверных процессов.

## Службы синхронизации

**Механизм межзадачного взаимодействия (IPC)**

Потоки внутри процесса имеют свободный доступ к общему пространству данных, но это не решает всех проблем межзадачного взаимодействия. Использовать только разделяемую память для обмена данными и обойтись без всех других механизмов взаимодействия зачастую не получится.

Проблема заключается в том, что доступ отдельных потоков к общим данным должен быть синхронизован. Если один поток пытается считать данные в тот момент, когда другой поток изменяет их, это может привести к катастрофическому отказу системы. Например, если один поток выполняет обновление связного списка, другие потоки не должны читать или изменять этот список, пока поток не закончит свою операцию.

Отрывок программного кода, который должен выполняться "последовательно" (serially), т. е. не более чем одним потоком одновременно, называется "критической секцией программного кода" (critical section).

В OC QNX Neutrino используются POSIX-примитивы для синхронизации на уровне потоков.



Эти примитивы синхронизации реализуются непосредственно ядром, за исключением:

* барьеров, ждущих блокировок (sleepon locks) и блокировок чтения/записи (которые основаны на мутексах и условных переменных);
* атомарных операций (которые либо реализуются непосредственно процессором, либо эмулируются ядром).

**Блокировки взаимного исключения (мутексы)**

Наиболее простыми из служб синхронизации являются мутексы. Мутекс (от англ. mutex, mutual exclusion lock) служит для обеспечения монопольного доступа к данным, которые совместно используются несколькими потоками. Операциями захвата мутекса (с помощью функции pthread\_mutex\_lock()) и освобождения мутекса (с помощью функции pthread\_mutex\_unlock()) обычно обрамляются участки кода, который обращается к совместно используемым данным (обычно это критическая секция кода).

В каждый момент времени мутекс может быть захвачен только одним потоком. Потоки, которые пытаются захватить уже захваченный мутекс, блокируются до тех пор, пока этот мутекс не освобождается. Когда поток освобождает мутекс, поток с наивысшим приоритетом, который ожидает возможности захватить мутекс, будет разблокирован и станет новым владельцем мутекса. Таким образом, потоки обрабатывают критическую секцию кода в порядке своих приоритетов.

В большинстве процессоров захват мутекса не требует обращения к ядру. Это достигается благодаря операции "сравнить и переставить" (compare-and-swap opcode) в семействе x86, а также посредством условных инструкций "загрузить/сохранить" на большинстве процессоров семейства RISC.

При захвате мутекса передача управления коду ядра происходит только при условии, что мутекс уже захвачен и поток может быть включен в список блокированных потоков. Управление передается коду ядра также при освобождении мутекса, если другие потоки ожидают разблокирования на этом мутексе. Это позволяет выполнять захват и освобождение критических секций с очень высокой скоростью. Таким образом, действие ОС сводится всего лишь к управлению приоритетами.

Для определения текущего состояния мутекса используется **специальная неблокирующая функция** pthread\_mutex\_trylock(). Для повышения производительности системы время выполнения критической секции кода должно быть небольшим. Если поток может блокироваться во время выполнения критической секции, то должна использоваться **условная переменная**.

**Наследование приоритетов**

Если поток, имеющий более высокий приоритет, чем владелец мутекса, пытается захватить мутекс, то действующий приоритет текущего владельца устанавливается равным приоритету блокированного потока, ожидающего мутекс. Владелец вернется к своему исходному приоритету после того, как он освободит мутекс. Такая схема не только обеспечивает минимальное время ожидания мутекса, но и решает классическую проблему инверсии приоритетов.

Для обеспечения такого поведения, функция pthread\_mutexattr\_init() устанавливает атрибут PTHREAD\_PRIO\_INHERIT. Для изменения этого значения может использоваться функция pthread\_mutexattr\_setprotocol(). Значение приоритета потока при вызове функции pthread\_mutex\_trylock() не изменяется, поскольку данный вызов является неблокирующим.

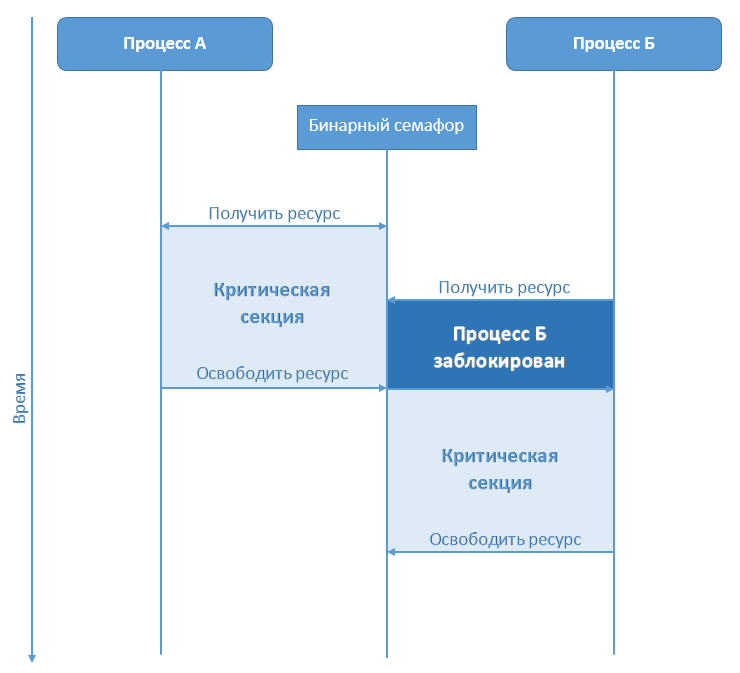
Для того, чтобы один и тот же поток мог выполнять рекурсивный захват мутекса, необходимо изменить атрибуты мутекса, используя функцию pthread\_mutexattr\_settype(). Это дает потоку возможность вызывать процедуры, которые выполняют **повторный захват мутекса**, уже захваченного данным потоком.

**Семафоры**

Еще одним средством синхронизации являются семафоры (semaphores), которые позволяют потокам увеличивать (с помощью функции sem\_post()) или уменьшать (с помощью функции sem\_wait()) значение счетчика на семафоре для управления блокировкой потока (операции "post" и "wait" соответственно).

При вызове функции sem\_wait() поток не блокируется, если счетчик имел положительное значение. При неположительном значении счетчика поток блокируется до тех пор, пока какой-либо другой поток не увеличит значение счетчика. Увеличение значения счетчика может выполняться несколько раз подряд, что позволяет уменьшать значение счетчика, не вызывая блокировки потоков.

Существенным отличием семафоров от других примитивов синхронизации является то, что семафоры безопасны для применения в асинхронной среде ("async safe") и могут управляться обработчиками сигналов. Семафоры как раз подходят для тех случаев, когда требуется пробудить поток с помощью обработчика сигнала.



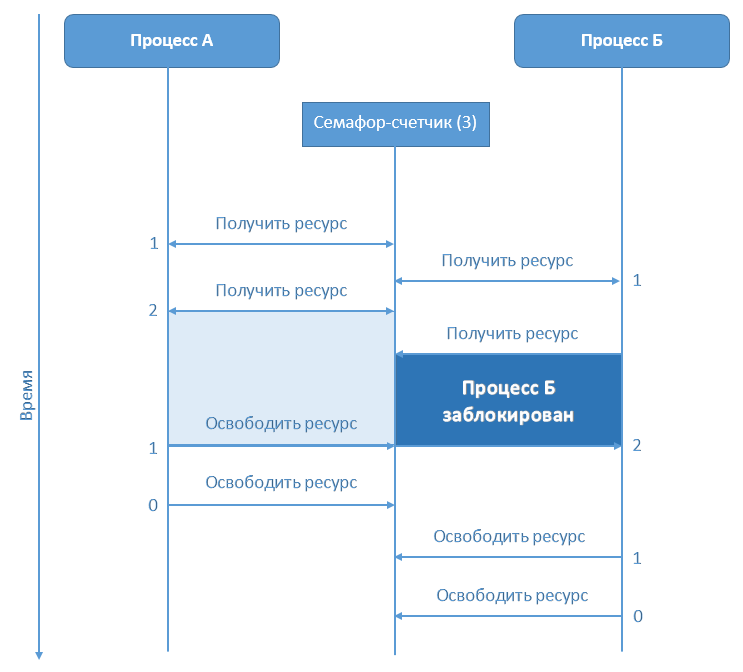
**Взаимное исключение на семафоре**

Для реализации взаимного исключения создается двоичный семафор S. Начальное значение этого семафора = 1. Критические секции кода обрамляются sem\_wait() (в начале секции) и sem\_post() (в конце секции). Процесс, входящий в критическую секцию, выполняет операцию (S) и переводит семафор в 0. Если в критической секции уже находится другой процесс, то значение семафора уже 0, тогда второй процесс, желающий войти в критическую секцию, блокируется до тех пор, пока процесс, находящийся сейчас в критической секции, не выйдет из нее, выполнив на выходе операцию sem\_post().

**Синхронизация на семафоре**

Для обеспечения синхронизации создается двоичный семафор S с начальным значением 0. Значение 0 означает, что событие еще не наступило. Процесс, сигнализирующий о наступлении события, выполняет операцию sem\_post(), устанавливающую семафор в 1. Процесс, ожидающий наступления события, выполняет операцию sem\_wait(). Если к этому моменту событие уже произошло, ожидающий процесс продолжает выполняться, если же событие еще не произошло, процесс переводится в состояние ожидания до тех пор, пока сигнализирующий процесс не выполнит sem\_post().

В случае, если одного и того же события ожидают несколько процессов, процесс, успешно выполнивший операцию sem\_wait(), должен вслед за ней выполнить sem\_post(), чтобы продублировать сигнал о событии для следующего ожидающего процесса.



**Семафор — счетчик ресурсов**

Если у нас имеется N единиц некоторого ресурса, то для контроля его распределения создается общий семафор S с начальным значением N. Выделение ресурса сопровождается операцией sem\_post(), освобождение — операцией sem\_wait(). Значение семафора отражает число свободных единиц ресурса. Если значение семафора = 0, то есть свободных единиц больше не остается, то очередной процесс, запрашивающий единицу ресурса, будет заблокирован до тех пор, пока какой-либо из процессов не освободит единицу ресурса, выполнив при этом sem\_wait().

Как правило, мутексы работают быстрее, чем семафоры (которые всегда требуют обращения к коду ядра). Семафоры не оказывают влияние на значение приоритета потока, поэтому для защиты от инверсии приоритетов необходимо использовать мутексы.

Другим полезным свойством семафоров является то, что они были определены для работы между процессами. Хотя мутексы в QNX Neutrino тоже работают между процессами, стандарты POSIX рассматривают эту возможность как дополнительную функцию, которая может оказаться непереносимой между системами. Что касается синхронизации между потоками внутри одного процесса, то здесь мутексы более эффективны, чем семафоры.

Полезной разновидностью семафоров является служба именованных семафоров (named semaphore service). Эта служба использует администратор ресурсов и позволяет применять семафоры между процессами, выполняемыми на разных машинах внутри сети.

Именованные семафоры работают медленнее, чем неименованные.

**Условные переменные**

Условная переменная (condvar – сокр. от condition variable) используется для блокировки потока по какому-либо условию во время выполнения критической секции кода. Условие может быть сколь угодно сложным и не зависит от условной переменной. Однако условная переменная всегда должна использоваться совместно с мутексом для проверки условия.

Условные переменные поддерживают следующие функции:

* ожидание условной переменной (wait) (pthread\_cond\_wait());
* единичная разблокировка потока (signal) (pthread\_cond\_signal());
* множественная разблокировка потока (broadcast) (pthread\_cond\_broadcast()).

Следует иметь в виду, что единичная разблокировка потока, обозначаемая термином "signal", никак не связана с понятием сигнала в стандартах POSIX.

Пример типичного использования условной переменной:

**pthread mutex lock( &m );**

**. . .**

**while (!arbitrary condition) {**

**pthread cond wait( &cv, &m );**

**}**

**. . .**

**pthread mutex unlock( &m );**

В этом примере захват мутекса происходит до проверки условия. Таким образом, проверяемое условие применяется только к текущему потоку. Пока данное условие является истинным, эта секция кода блокируется на вызове ожидания до тех пор, пока какой-либо другой поток не выполнит операцию единичной или множественной разблокировки потока по условной переменной.

Другой вид операции ожидания условной переменной (pthread\_cond\_timedwait()) позволяет установить таймаут. По окончании этого периода ожидающий поток может быть разблокирован.

Цикл while в приведенном примере требуется по двум причинам.

Во-первых, стандарты POSIX не гарантируют отсутствие ложных пробуждений (например, в многопроцессорных системах).

Во-вторых, если другой поток изменяет условие, необходимо заново выполнить его проверку, чтобы убедиться, что изменение соответствует принятым критериям. При блокировании ожидающего потока связанный с условной переменной мутекс атомарно освобождается функцией pthread\_cond\_wait() для того, чтобы другой поток мог войти в критическую секцию программного кода.

Поток, который выполняет единичную разблокировку потока, разблокирует поток с наивысшим приоритетом, который стоит в очереди на условной переменной. Операция множественной разблокировки потока разблокирует все потоки, стоящие в очереди на условной переменной. Связанный с условной переменной мутекс освобождается атомарно разблокированным потоком с высшим приоритетом. После обработки критической секции этот поток должен освободить мутекс.

**Барьеры**

Барьер — это механизм синхронизации, который позволяет скоординировать работу нескольких взаимодействующих потоков (например, при матричных вычислениях) таким образом, чтобы каждый из них остановился в заданной точке в ожидании остальных потоков, прежде чем продолжить свою работу.

В отличие от функции pthread\_join(), при которой поток ожидает завершения другого потока, барьер заставляет потоки встретиться в определенной точке. После того, как заданное количество потоков достигает установленного барьера, все эти потоки разблокируются и продолжат свою работу.

Барьер создается с помощью функции pthread\_barrier\_init():

**#include <pthread.h>**

**int**

**pthread barrier init (pthread barrier t \*barrier,**

**const pthread barrierattr t \*attr,**

**unsigned int count);**

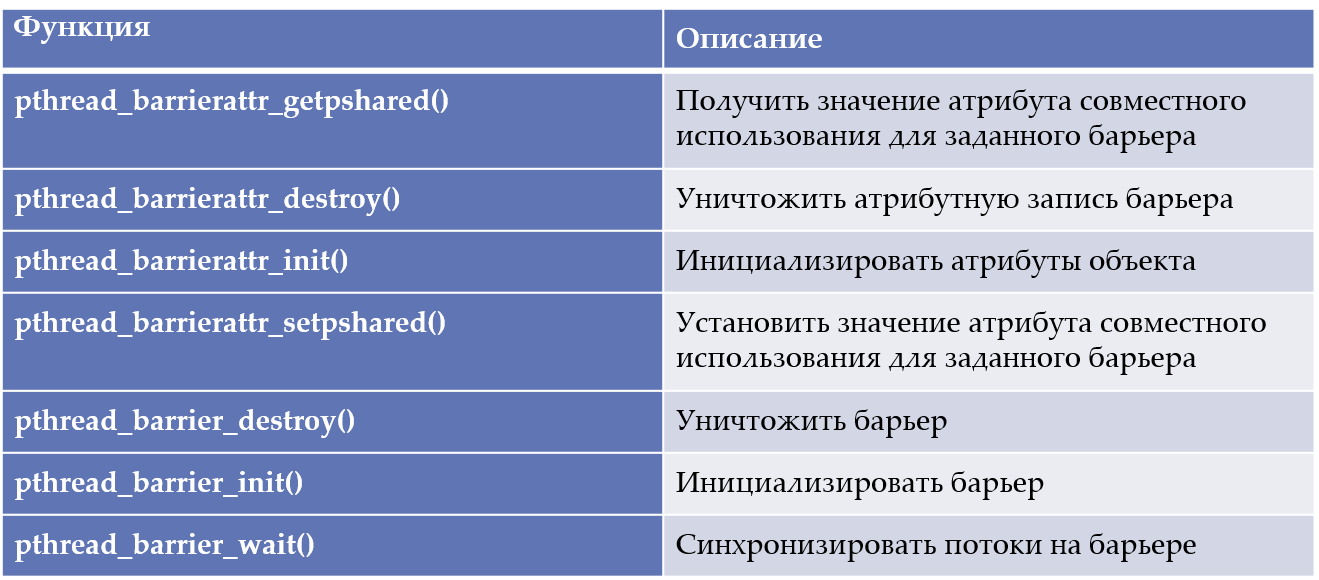
В результате выполнения этого кода создается барьер по заданному адресу (указатель на барьер находится в аргументе barrier) и с атрибутами, установленными аргументом attr. Аргумент count задает количество потоков, которые должны вызвать функцию pthread\_barrier\_wait().

После создания барьера каждый поток вызывает функцию pthread\_barrier\_wait(), тем самым сообщая о завершения этого действия:

**#include <pthread.h>**

**int pthread barrier wait (pthread barrier t \*barrier);**

Когда поток вызывает функцию pthread\_barrier\_wait(), он блокируется до тех пор, пока то число потоков, которое было задано функцией pthread\_barrier\_init(), не вызовет функцию pthread\_barrier\_wait() и, соответственно, не заблокируется. После того как заданное количество потоков вызовет функцию pthread\_barrier\_wait(), все они разблокируются одновременно.



Функции работы с барьерами QNX Neutrino.

**Ждущие блокировки**

Ждущие блокировки (sleepon locks) работают аналогично условным переменным, за исключением некоторых деталей. Как и условные переменные, ждущие блокировки (pthread\_sleepon\_lock()) могут использоваться для блокировки потока до тех пор, пока условие не станет истинным (аналогично изменению значения ячейки памяти). Но в отличие от условных переменных (которые должны существовать для каждого проверяемого условия), ждущие блокировки применяются к одному мутексу и динамически создаваемой условной переменной независимо от количества проверяемых условий.

Максимальное число условных переменных в конечном итоге равно максимальному числу блокированных потоков. Этот вид блокировок аналогичен тем, которые применяются в ядре UNIX.

**Блокировки по чтению/записи**

Блокировки по чтению/записи (reader/writer locks) (или более точное название "блокировки на множественное чтение и однократную запись") используются в тех случаях, когда доступ к структуре данных должен определяться по следующей схеме: чтение данных выполняет множество потоков, запись — не более одного потока. Хотя этот вид блокировок несет больше накладных расходов, чем мутексы, он позволяет организовывать описанную схему доступа к данным.

Блокировка по чтению/записи (pthread\_rwlock\_rdlock()) предоставляет доступ по чтению всем потокам, которые его запрашивают. Однако если поток запрашивает блокировку по записи (pthread\_rwlock\_wrlock()), запрос отклоняется до тех пор, пока все потоки, выполняющие чтение, не снимут свои блокировки по чтению (pthread\_rwlock\_unlock()).

Множество потоков, выполняющих запись, выстраиваются в очередь (в порядке своих приоритетов), ожидая возможности выполнить операцию записи в защищенную структуру данных. Все блокированные потоки, выполняющие запись, запускаются до того, как читающие потоки снова получат разрешение на доступ к данным. Приоритеты читающих потоков не учитываются.

Существуют специальные вызовы (pthread\_rwlock\_tryrdlock() и pthread\_rwlock\_trywrlock()), которые позволяют потоку тестировать возможность доступа к необходимой блокировке, оставаясь в активном состоянии. Эти вызовы возвращают код завершения, сообщающий о возможности или невозможности установки блокировки.

Реализация блокировок по чтению/записи происходит не в ядре, а посредством мутексов и условных переменных, предоставляемых ядром.

**Синхронизация с помощью алгоритма планирования**

Применение алгоритма FIFO-планирования стандарта POSIX в системе без симметричной многопроцессорной обработки предотвращает выполнение критической секции кода одновременно несколькими потоками с одинаковым приоритетом. Алгоритм FIFO-планирования предписывает, что все потоки, запланированные к выполнению по этому алгоритму и имеющие одинаковый приоритет, выполняются до тех пор, пока они самостоятельно не освободят процессор для другого потока.

Такое «освобождение» процессора также может произойти в случае, когда поток блокируется в результате обращения к другому процессу за сервисом или при получении сигнала. Поэтому критическая секция кода должна быть тщательно проработана и документирована для того, чтобы последующее обслуживание этого кода не приводило к нарушению данного условия.

Кроме того, потоки с более высоким приоритетом в том (или любом другом процессе) все же могут вытеснять потоки, выполняемые по алгоритму FIFO-планирования. Поэтому все потоки, которые могут "столкнуться" между собой внутри критической секции кода, должны планироваться по алгоритму FIFO с одинаковым приоритетом. При таком условии потоки могут обращаться к этой разделяемой памяти без необходимости делать предварительный явный запрос на синхронизацию.

Метод монопольного доступа **неприменим в многопроцессорных системах**, поскольку в таких системах несколько процессоров могут одновременно исполнить код, который в однопроцессорной машине был бы запланирован на последовательное исполнение.

**Синхронизация с помощью атомарных операций**

В некоторых случаях необходимо выполнить какую-либо небольшую операцию (например, увеличить значение переменной) атомарно, т. е. с гарантией того, что она не будет вытеснена другим потоком или каким-либо обработчиком прерываний (Interrupt Service Routine, ISR).

В ОС QNX Neutrino атомарные операции применяются для:

* сложения;
* вычитания;
* обнуления битов;
* установки битов;
* переключения (дополнения) битов.

Атомарные операции можно задействовать, подключив заголовочный файл <atomic.h>.

Особенно атомарные операции полезны в следующих двух случаях:

* при взаимодействии между обработчиком прерываний (ISR) и потоком;
* при взаимодействии между двумя потоками (как при многопроцессорной, так и при однопроцессорной обработке).

Поскольку обработчик прерываний может вытеснять поток в любой точке, единственным способом защиты потока от прерываний остается запрещение прерываний. Поскольку запрещение прерываний неприемлемо для системы реального времени, рекомендуется применение атомарных операций, которые предусмотрены в QNX Neutrino.

В многопроцессорных системах множество потоков выполняются одновременно, поэтому здесь ситуация аналогичная — чтобы избежать запрещения прерываний, следует там, где это возможно, использовать атомарные операции.

**Синхронизация с помощью механизма обмена сообщениями**

Службы обмена сообщениями (Send/Receive/Reply), используемые в ОС QNX Neutrino, осуществляют неявную синхронизацию посредством блокировок.

Во многих случаях они могут заменить собой другие службы синхронизации. Кроме того, службы обмена сообщениями — единственные примитивы синхронизации и межзадачного взаимодействия (кроме именованных семафоров, основанных на механизме обмена сообщениями), которые могут работать в сети.