Зыль Сергей Николаевич, к.т.н., специалист по защищенным системам жесткого реального времени на базе операционной системы QNX, преподаватель авторизованного учебного центра QNX компании SWD Software Ltd. (QNX Approved Trainer)».

Операционная система реального времени QNX

от теории к практике

2-е издание

Книга основана на материалах учебного курса "Основы администрирования и разработки приложений для ОСРВ QNX Neutrino", преподаваемого автором в учебном центре компании SWD Software Ltd (http://www.swd.ru). Прочитав эту книгу, Вы узнаете о том, что представляет собой ОС QNX Neutrino, познакомитесь с возможностями, которые предоставляет разработчикам встраиваемых приложений инструментальный комплект QNX Momentics. А некоммерческий дистрибутив QNX Momentics, содержащийся на компакт-диске, позволит Вам "потрогать QNX Neutrino собственными руками".

Компакт-диск содержит дистрибутив комплекта разработчика QNX Momentics NC

ISBN 5-94157-486-X

9 785941 574865

БХВ-Петербург

190005, Санкт-Петербург, Измайловский пр., 29

E-mail: mail@bhv.ru
Internet: www.bhv.ru

тел.: (812) 251-42-44
факс: (812) 251-12-95

Сергей Зыль

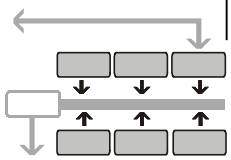
Операционная система реального времени QNX

от теории к практике

2-е издание

Установка и настройка
Архитектура и управление ресурсами
Графическая оболочка Photon microGUI
Сетевые механизмы
Целевые конфигурации

+CD-ROM



Глава 1

"О терминах не спорят, о них договариваются"

— *Плесни-ка мне холодного кипятку.*
— *Как кипяток может быть холодным?! Ты, верно, хочешь кипяченой воды?*
— *Какая еще кипяченая вода? Не пудри мне мозги — налей холодного кипятку.*

(разговор петербуржца и москвича)

В этой главе будут рассмотрены вопросы:

- что значит "ОС жесткого реального времени";
- версии QNX;
- инструментальные и целевые системы;
- дистрибутивы QNX;
- что такое POSIX-совместимость.

Что значит "ОС жесткого реального времени"

Прежде чем начать говорить об операционной системе жесткого реального времени QNX, давайте определимся, что это вообще такое — "операционная система жесткого реального времени"?

Под *системой реального времени* понимается такая информационная система, в которой корректность выходной информации

зависит не только от правильности примененных алгоритмов, но и от времени появления результатов обработки информации. То есть при опоздании результатов они либо могут быть бесполезными, либо ущерб в результате опоздания может быть бесконечно велик.

Этот временной критерий условно разделил операционные системы на два класса — операционные системы общего назначения (General Purpose Operation Systems — GPOS) и операционные системы реального времени (Real Time Operation Systems — RTOS).

Основная задача *операционных систем общего назначения* — эффективное разделение ресурсов ЭВМ (таких как процессорное время, оперативная память и т. п.) между несколькими одновременно выполняющимися программами. Такие операционные системы поставляются с богатым набором прикладных программ (приложений) и имеют развитый графический интерфейс, позволяющий пользователям работать с приложениями, не задумываясь над внутренними механизмами ОС.

А *операционные системы реального времени* (ОСРВ) разрабатываются в расчете на наличие внешних источников данных. Основная задача ОСРВ — своевременно обработать запрос, все остальные аспекты функционирования ЭВМ отходят на второй план. Поэтому ОСРВ поставляются в комплекте с разнообразными средствами разработки приложений. Другими словами, покупателями ОСРВ являются не конечные пользователи, а разработчики программного обеспечения.

Часто в обиходе говорят об операционных системах "мягкого" и "жесткого" реального времени. Чем же они различаются? ОС *жесткого* реального времени ГАРАНТИРУЕТ выполнение каких-то действий за определенный интервал времени. А ОС *мягкого* реального времени, КАК ПРАВИЛО, УСПЕВАЕТ выполнить заданные действия за заданное время. То есть ОС мягкого реального времени выполняет задачу с каким-то значением вероятности.

Версии QNX

Что же мы имеем в виду, произнося слово "QNX"? QNX — это семейство операционных систем жесткого реального времени, ориентированных главным образом на рынок встраиваемых систем.

Такое назначение требует от ОС максимального использования особенностей аппаратуры. Поэтому ОС QNX разных версий (правильнее было бы сказать — поколений) не имеют между собой двоичной совместимости. Конечно, разработчики при создании новых "версий" QNX используют лучшие решения из предыдущих версий и остаются верными фундаментальным концепциям QNX.

Для коммерческого использования доступно три семейства ОС QNX: QNX2, QNX4 и QNX6.

- ОС QNX2 мало распространена в России. Во-первых, до 1991 года система была запрещена к вывозу из Северной Америки как стратегический ресурс. Во-вторых, ЭВМ на базе процессоров Intel 286 почти повсеместно вышли из употребления.
- ОС QNX4 использует защищенный режим, поэтому может применяться на процессорах не ниже Intel 386. QNX4, пожалуй, самая распространенная на сегодняшний день ОС QNX в промышленности. Последняя ее версия — 4.25 патч G (вышла в начале 2003 года). Компания QSS не развивает эту ОС, но продолжает выпускать к ней драйверы для новых устройств.
- ОС QNX6 (или QNX Neutrino) благодаря интересным и необычным архитектурным решениям может использоваться не только на *x*86-совместимых ЭВМ и еще более удобна для встраивания, чем QNX4. QNX Neutrino разрабатывается с полной поддержкой спецификаций POSIX; кроме того, разработчики системы стремятся обеспечить максимальную переносимость в QNX исходных кодов, написанных для ОС Linux. Это значительно облегчает портирование свободно распространяемого ПО в QNX и, с другой стороны, обеспечивает "переносимость" для разработчиков — прикладной программист, знающий любую UNIX-подобную ОС, может приступать к разработке прикладного ПО для QNX без переучивания. Последняя версия QNX Neutrino — 6.3.

Инструментальные и целевые системы

С точки зрения QNX все инсталлированные копии ОС относятся к двум категориям: инструментальные системы и целевые системы.

Инструментальная система (или среда разработки) — это ЭВМ со средствами, позволяющими формировать образ целевой системы. Такие средства включены только в коммерческие дистрибутивы пакета разработчика QNX Momentics.

Целевая система (или среда исполнения) предназначена для эксплуатации. Среда исполнения фактически представляет собой инструментальную систему, из которой удалено все ненужное для данной прикладной задачи.

Дистрибутивы QNX

Дистрибутив — это способ поставки инструментальных систем. Другими словами, QSS продает именно инструментарий разработки. Среды исполнения формируются разработчиком самостоятельно, но для поставки их кому-либо еще требуется покупка лицензий на соответствующее число копий среды исполнения (их называют "модули Run-Time"). Такая политика позволяет существенно снизить стоимость целевых систем.

Следует обратить внимание, что средства разработки поставляются в четырех вариантах — для Microsoft Windows XP, Sun Solaris (только для SRARC-версии), Linux и, разумеется, для QNX.

Целевые системы могут генерироваться для нескольких аппаратных платформ (ARM, MIPS32, StrongARM, SH4, PowerPC, Xscale, *x*86). Этот список, вероятно, будет пополняться.

Выдвигая на рынок ОС QNX6, компания QSS выпустила два дистрибутива:

- QNX Real Time Platform (QNX RTP) — полнофункциональная среда разработки; распространялась бесплатно для некоммерческого использования. В случае коммерческого использования системы разработчик должен был приобрести соответствующую лицензию у QSS;
- QNX Networking Infrastructure Platform (QNX NIP) — чисто коммерческий дистрибутив, представляющий собой расширение QNX RTP дополнительными программными пакетами, ориентированными на производителей сетевого оборудования.

С выходом версии QNX 6.2 компания QSS несколько изменила подход к формированию дистрибутивов. Пакет разработчика,

включающий в себя ОСРВ QNX, графическую среду Photon, инструменты разработки и различное дополнительное ПО, получил название QNX Momentics. В настоящее время выпущено несколько дистрибутивов QNX Momentics:

- Non-Commercial Edition (NC) — ознакомительный комплект разработчика, бесплатный для некоммерческого использования;
- Standard Edition (SE) — пакет разработчика, позволяющий вести коммерческую разработку ПО, в том числе формировать целевые системы для разных платформ;
- Professional Edition (PE) — расширенный пакет разработчика, дополненный интегрированной средой разработки QNX IDE, основанной на технологии Eclipse, а также расширенной базой примеров в исходных текстах и рядом дополнительных компонентов.

Подробная информация по комплектации дистрибутивов Momentics доступна как у QSS, так и у ее партнеров. Профессиональный комплект QNX Momentics считается базовым (т. е. SE — фактически является урезанным PE). PE может дополняться пакетами расширения (Bundles) для разных целевых рынков.

Теперь вы знаете, чем пакет разработчика QNX Momentics отличается от ОСРВ QNX ☺.

Что такое POSIX-совместимость

POSIX (Portable Operation Systems Interface) — это развивающийся стандарт, призванный обеспечить переносимость исходных текстов программ между ОС разных производителей. За основу стандартов POSIX были взяты ОС семейства UNIX, в дальнейшем стандарты были дополнены расширениями, включая расширения, касающиеся реального времени.

Разработкой стандартов POSIX занимаются рабочие группы Института инженеров по электротехнике и радиоэлектронике (ИИЭР, Institute of Electrical and Electronics Engineers — IEEE) США. Поэтому стандарты POSIX после утверждения имеют маркировку IEEE. Кроме того, существуют Международная организация по стандартизации (International Organization for

Standardization — ISO) и Международная электротехническая комиссия — МЭК (International Electrotechnical Commission — IEC). Эти организации могут утверждать стандарты IEEE в качестве международных.

Стандарт ISO/IEC 9945-1 определяет интерфейс прикладного программирования (API) для операционных систем. Этот стандарт включает следующие стандарты:

- POSIX.1 (IEEE 1003.1) — базовый API операционных систем;
- POSIX.1a (IEEE 1003.1a) — некоторые расширения API;
- POSIX.4 (IEEE 1003.1b) — расширения для поддержки реального времени;
- POSIX.4a (IEEE 1003.1c) — интерфейсы потоков, выполняющихся внутри POSIX-процессов;
- POSIX.1b (IEEE 1003.1d) — дополнительные расширения реального времени;
- POSIX.12 (IEEE 1003.1g) — независимый от протокола интерфейс сокетов;
- IEEE 1003.1j — еще одно дополнительное расширение реального времени.

Стандарт ISO/IEC 9945-2 (POSIX.2 или IEEE 1003.2) определяет набор утилит и командных интерпретаторов.

Стандарт ISO/IEC 13210 (POSIX.3 или IEEE 1003.0) определяет набор тестов, позволяющих определить POSIX-совместимость операционной системы.

Заметим, что в документации часто используется комбинированное обозначение номеров стандартов, например вместо POSIX.4a или IEEE 1003.1c пишут POSIX 1003.1c.

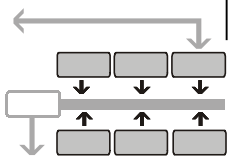
Однако многие ОСРВ работают внутри некоторого промышленного оборудования с весьма ограниченными ресурсами, т. е. являются *встраиваемыми системами*. Встраиваемые системы не могут и не должны обеспечивать всю POSIX-функциональность. Поэтому было решено определить, если можно так выразиться, правила "урезания" стандартов POSIX в необходимых случаях. Такие "урезания" названы *профилями прикладных контекстов реального времени* и регламентируются стандартом ISO/IEC ISP 15287-2

(POSIX.13 или IEEE 1003.13). Этот стандарт определяет такие профили:

- *минимальная система* — встроенная система без поддержки механизма управления памятью (MMU), без файловой системы и терминала. В такой системе разрешается только один многопоточный процесс;
- *контроллер реального времени* — минимальная система, дополненная файловой системой и терминалом ввода/вывода;
- *специализированная система* — большая встроенная система, в которой может выполняться несколько многопоточных процессов. Файловая система отсутствует;
- *многоцелевая система* — операционная система реального времени, обеспечивающая поддержку всей предусмотренной стандартом POSIX функциональности.

ОСРВ QNX6 изначально разрабатывалась как POSIX-совместимая ОС, позволяющая создавать целевые системы, соответствующие любому из четырех профилей прикладного контекста реального времени. Термины, используемые в документации, поставляемой с QNX, соответствуют терминологии стандартов POSIX.

Глава 2



Инсталляция среды разработки

В этой главе рассмотрены следующие вопросы:

- общие сведения;
- инсталляция QNX Neutrino RTOS;
- настройка QNX после инсталляции;
- инсталляция дополнительного программного обеспечения.

Общие сведения

Прежде чем приступить к установке QNX Momentics, следует определить, на какой аппаратно-программной платформе мы хотим работать. Как мы уже знаем, поддерживаются четыре хост-платформы:

- для операционной среды Solaris (только для SPARC-версии);
- для операционной системы Windows XP;
- для операционной системы RedHat Linux;
- для операционной системы QNX Neutrino.

Производители всех этих операционных систем публикуют списки поддерживаемого оборудования — ознакомьтесь с ними ДО того, как приступите к инсталляции.

При установке ОСРВ QNX инструменты разработки будут инсталлированы автоматически, а для установки кросс-платформенной среды разработки QNX Momentics необходимо:

- инсталлировать и настроить операционную систему Windows, Linux или Solaris;
- инсталлировать собственно QNX Momentics с соответствующего компакт-диска.

Примечание

В дополнение к основным компакт-дискам могут прилагаться диски с документацией, видеокурсами, программным обеспечением иных производителей и т. п. Хотя в последнее время компания QSS предпочитает распространять дополнительные материалы через свой сайт.

Для начала рассмотрим инсталляцию операционной системы QNX Neutrino. Если выбрана другая операционная система, то ее необходимо устанавливать в соответствии с указаниями фирмы-производителя.

Инсталляция QNX Neutrino RTOS

ОС QNX инсталлируется в собственный раздел диска. Перед инсталляцией (а лучше — перед покупкой "железа") ознакомьтесь со списком аппаратуры, протестированной QSS на совместимость с QNX. Этот список находится на сайте компании QSS по адресу: **http://www.qnx.com/support/sd_hardware/**. Неплохая идея — обратиться в службу технической поддержки фирмы, продавшей вам QNX. Заодно сможете оценить то, как данный поставщик решает проблемы своих клиентов ☺.

Для использования QNX Momentics PE требуется:

- процессор — не ниже Pentium III 700 МГц (желательно Pentium-4 2 ГГц);
- ОЗУ — не меньше 256 Мбайт (желательно 512 Мбайт);
- свободное дисковое пространство — не меньше 1,5 Гбайт.

Такие требования вызваны значительной ресурсоемкостью интегрированной среды разработки (IDE), написанной на языке Java. Если вам не нужна IDE, то не нужны и такие аппаратные ресурсы.

Хотелось бы заметить, что ОС QNX 6.3 можно без конфликтов устанавливать на одном диске с QNX 6.2.*x*. Для этого достаточно в среде QNX 6.2.*x* предварительно запустить утилиту `fdisk` и изменить номер раздела QNX 6.2.*x* с 79 на 77 или 78 (QNX 6.3 будет устанавливаться в раздел 79).

В процессе инсталляции операционной системы выполняются два действия:

1. Создается структура каталогов на жестком диске.
2. Базовое программное обеспечение копируется на жесткий диск.

Для начала инсталляции загрузите ваш компьютер с компакт-диска QNX. На экране появится сообщение:

```
Please select a boot option. Option F2 is great for testing QNX
compatibility on new hardware without writing anything to the
hard disk. It can also be used for system recovery.
F2 — Run from CD (Hard Disk filesystems mounted under /fs)
F3 — Install QNX to a new disk partition
Select?
```

(Выберите, пожалуйста, вариант загрузки. Выбор клавиши <F2> удобен для проверки совместимости QNX с новой аппаратурой без записи какой-либо информации на диск. Этот вариант можно также использовать для восстановления системы после сбоя.

<F2> — загрузить систему непосредственно с компакт-диска (файловые системы монтируются в /fs);

<F3> — установить QNX в новый раздел диска.

Ваш выбор?)

Если нажать клавишу <F2>, то QNX загрузится с компакт-диска. При этом будет предложено изменить настройки видеорежима. Инсталляция QNX выполняться не будет.

Нажмем клавишу <F3>. На экране появится сообщение:

```
This installation will create a partition on your hard disk and
create a bootable QNX image. You may abort this installation at
any prompt by pressing the F12 key.
Press F1 to continue.
Press F2 to set verbose (debug) mode.
Choice (F1, F2)?
```

(Во время инсталляции будут созданы раздел на жестком диске и загрузочный образ QNX. Процесс инсталляции можно прервать в любой момент нажатием клавиши <F12>.

Нажмите клавишу <F1> для продолжения инсталляции.

Нажмите клавишу <F2> для перехода в режим подробного вывода информации (отладочный режим).

Ваш выбор (<F1>, <F2>)?)

В дальнейшем мы будем рассматривать инсталляцию в обычном режиме. Обычно этого вполне достаточно. Итак, нажмем клавишу <F1>. На экране появится информация по лицензированию QNX. Программа инсталляции предложит принять или отвергнуть условия лицензии.

```
F1 - accept   F2 - reject
```

(<F1> — принять условия; <F2> — отвергнуть условия)

Если нажать клавишу <F2>, то инсталляция прекратится, программа-инсталлятор предложит вынуть компакт-диск QNX из дисковода и перезагрузить ЭВМ. Если вы согласны с условиями лицензии, то нажмите клавишу <F1>. На экране появится сообщение:

```
*** WARNING***

You have a disk which is greater than 8.4 Gigabytes. The origi-
nal BIOS calls to access the disk are unable to read data above
8.4G. If your BIOS is older then 1998 you may be forced to
choose option 2. Newer BIOS's support on extended disk read
calls which can access the entice drive.

F1 Allow the QNX partition to be anywhere on the disk.

F2 Keep the QNX partition below 8.4G.

Choice (F1, F2)?
```

(Объем диска на вашем компьютере превышает 8,4 Гбайта. Обычные функции BIOS, предназначенные для доступа к дискам, не могут читать данные, расположенные за пределами 8,4 Гбайт. Если BIOS произведена ранее 1998 года, то вам следует нажать клавишу <F2>. Более новые BIOS поддерживают расширенные вызовы чтения, позволяющие работать с любыми дисками.

<F1> — раздел QNX может находиться в любом месте диска;

<F2> — раздел QNX должен находиться в пределах 8,4 Гбайт.

Ваш выбор (<F1>, <F2>)?)

Надеюсь, что ваш компьютер был выпущен после 1998 года ☺. В противном случае вам нужно нажать клавишу <F1> для того, чтобы избежать аппаратных конфликтов при работе с жесткими дисками.

Допустим, вы нажали клавишу <F1>. Следующее сообщение будет зависеть от того, сколько свободного места вы оставили на диске. У меня сообщение выглядело так:

```
Your disk has room for a 10001 megabyte QNX partition?
```

```
Please select the size of the partition you would like to create
for QNX.
F1              all             10001 M
F2              half            5000 M
F3              quarter         2500 M
F4              eighth          1250 M
F5 Display partition table allowing you to delete an existing
partition.
```

(На диске есть 10 001 Мбайт свободного пространства для установки QNX[1]. Выберите размер раздела для QNX.

<F1> все место 10001 M

<F2> половина 5000 M

<F3> четверть 2500 M

<F4> одна восьмая 1250 M

<F5> — показать таблицу разделов для удаления существующего раздела.)

Вы можете использовать под раздел QNX либо все свободное место на диске, либо его часть. При необходимости, нажав клавишу <F5>, можно удалить какой-нибудь из существующих разделов диска. Я выбрал вариант с клавишей <F2> (5 Гбайт — более чем достаточно для QNX), и на экране появилось сообщение:

```
You have more then one partition on your hard disk. To select
which partition to had when you boot requires a special parti-
tion boot loader.

You have three choices:

F1 Install QNX partition boot loader. It will prompt you on boot
to select which partition (OS) to load? If your partition may
start above 8.4 G we recommend this choice.

F2 Install the QNX partition boot loader for machines will old
BIOS (before 1996/1997). It should not be used with drives
greater then 8.4 G.

F3 Use your existing boot loader which may already provide this
capability? Examples include System Commander or LILO. If it
does not provide this capability you will only be able to boot
```

[1] Зачем здесь вопросительный знак, я, право, не знаю.

```
the currently set active partition. If you partition starts
above 8.4 G this existing loader will need to use the new ex-
tended BIOS disk calls.


Choosing F1 will write the QNX partition loader to your primary
hard disk. If you installed QNX to another disk the loader will
be written to it as well incase you later decide to make it you
primary hard disk.


Choice (F1, F2, F3)?
```

(На диске есть больше одного раздела. Чтобы обеспечить выбор раздела для загрузки, необходим специальный загрузчик.

У вас есть несколько вариантов:

<F1> — установить "родной" загрузчик QNX. Он будет выдавать приглашение выбрать раздел (т. е. ОС) для загрузки. Если раздел QNX находится за пределами 8,4 Гбайт, то мы рекомендуем этот вариант;

<F2> — инсталлировать "родной" загрузчик QNX для компьютеров со старой BIOS (выпущенной до 1996–1997 гг.). Его нельзя использовать с дисками объемом свыше 8,4 Гбайт.

<F3> — оставить прежний загрузчик. QNX могут загружать "не родные" загрузчики, например System Commander или LILO. Однако если такой загрузчик не установлен, то вы сможете загружать только ту ОС, которая установлена в активном на данный момент разделе диска.

При выборе клавиши <F1> загрузчик QNX будет записан на первичный жесткий диск. Если ОС QNX инсталлирована на другой диск, то загрузчик будет записан и на тот диск на случай, если тот диск будет сделан первичным.

Ваш выбор (<F1>, <F2>, <F3>)?)

Лично меня вполне устраивает загрузчик QNX, однако некоторых пользователей он не удовлетворяет, т. к. не имеет графического интерфейса. Что ж, такие пользователи могут поэкспериментировать с мультизагрузчиками третьих производителей.

Вот и все. Программа инсталляции получила информацию, достаточную для установки как самой операционной системы, так и инструментов разработчика. На самом деле, процесс инсталляции может несколько отличаться от описанного выше, в зависимости от используемого вами компьютера (например, на вашей ЭВМ есть несколько жестких дисков, или программа-инсталлятор обнаружила уже готовый раздел QNX). Но в главном последовательность действий сохраняется. Итак, нажимаем

клавишу <F1> (установить загрузчик QNX) — на экране появляется сообщение:

```
Restarting driver and mounting filesystems ...
Copying files to the new QNX partition ...
Please wait.
```

(Перезапускается драйвер и монтируются файловые системы ...

Копируются файлы в новый раздел QNX ...

Пожалуйста, подождите.)

В процессе копирования файлов на экране в реальном времени отображается, сколько процентов файла уже скопировано:

```
100% COPY /cd/boot/fs/qnxbase.ifs to /hdisk/boot/fs/
100% COPY /cd/boot/fs/qnxbase.ifs to /hdisk/.altboot
100% COPY /cd/boot/fs/qnxbase.qfs to /hdisk/boot/fs/
100% COPY /cd/boot/fs/qnxbasedma.ifs to /hdisk/boot/fs/
100% COPY /cd/boot/fs/qnxbasedma.ifs to /hdisk/.boot
100% COPY /cd/boot/fs/qnxbasesmp.ifs to /hdisk/boot/fs/
```

После окончания копирования файлов на экран выдается сообщение:

```
Installation complete.
Please remove the install floppy and CD, then reboot your machine.

Notes:
1. When you get login prompt you should login as root.
2. By default we enable DMA on IDE driver. If your machine fails
to boot correctly press the ESC key when prompted to boot the
alternative OS boot image which disables DMA.

Press any key to reboot ...
30
```

(Инсталляция закончена.

Выньте, пожалуйста, из дисковода дискету и компакт-диск и перезагрузите ЭВМ.

Замечания:

1. При получении приглашения для входа в систему (`login:`) введите имя `root`.

2. По умолчанию у драйвера IDE включен режим DMA[1]. Если ваша ЭВМ поведет себя некорректно, то нажмите во время загрузки клавишу <ESC>

---

[1] DMA (Direct Memory Access) — прямой доступ к памяти (ПДП).

для загрузки альтернативного образа ОС, содержащего драйвер с отключенным режимом DMA.

Нажмите любую клавишу для перезагрузки ...)

Если вы ничего не нажмете, то ЭВМ перезагрузится автоматически через 30 с. Если инсталляция прошла правильно, то вы получите приглашение загрузчика QNX выбрать раздел диска для загрузки. Выберите раздел с QNX (впрочем, если вы ничего не выбрали, QNX через пару секунд начнет загружаться автоматически).

Примечание

Напоминаю: после установки в системе зарегистрирован только один пользователь с именем `root`, не имеющий пароля (и не говорите потом: "Ой, у нас QNX установился и загрузился, но мы не знаем, какое имя и пароль вводить, — в бумагах, которые мы получили с диском, ничего об этом не написано!").

Появится сообщение о загрузке образа QNX и сканировании оборудования, за ним — сообщения об операциях, выполняемых однократно при первой загрузке:

`Creating swap file ...` (Создание файла для свопинга)

`Creating helpviewer database` (Создание базы данных helpviewer'а)

Сразу оговорюсь, что хотя файл для свопинга страниц ОЗУ и создается, но по умолчанию он не используется. Ведь QNX — система жесткого реального времени, поведение которой должно быть полностью предсказуемо. Поэтому свопинг процессы выполняют "вручную", если он им нужен.

После этого система переключается в графический режим и выводит окно для первоначального конфигурирования графического драйвера. Вам необходимо установить 4 параметра.

1. Выбрать из списка видеодрайвер (мне система предложила три драйвера: VGA, TNT и VESA).
2. Выбрать разрешение экрана.
3. Выбрать глубину цвета.
4. Установить частоту обновления экрана.

Кроме того, конфигуратор позволяет выбрать — запускать автоматически графическую оболочку Photon при каждой загрузке

системы или нет (я со временем стал ленивым и теперь всегда указываю системе запускать графическую среду автоматически).

После того как вы установили нужные значения параметров, нажмите кнопку **Change Mode** (Изменить видеорежим). При этом система переключится на новые настройки и выведет окно для проверки их корректности. Если настройки были выбраны правильно, то сообщение в контрольном окне будет читабельным и вы сможете нажать в этом окне кнопку **Continue** (Продолжить). Если вы не нажмете эту кнопку, то система автоматически переключится на прежние настройки через 15 с.

Затем QNX предложит ввести имя пользователя и пароль. В свежеустановленной системе есть, как мы помним, только один пользователь — `root` без пароля. Вводим имя, игнорируем запрос на ввод пароля — и (поздравляю!) мы в QNX. Инсталляция системы успешно завершена.

Настройка QNX после инсталляции

Для нормальной работы необходимо выполнить несколько простых операций. Если ЭВМ подключена к сети, то нам потребуется знать IP-адрес и сетевую маску нашей машины, IP-адреса шлюза и сервера имен. Если вы — системный администратор, то эти сведения вам известны, в противном случае обратитесь к системному администратору. Заметим, что задать хотя бы имя хоста и его IP-адрес желательно даже для одиночного компьютера, потому что многие программы изначально рассчитаны для работы в сети и работают корректно только при наличии сетевых настроек (характерный пример — QNX IDE).

Итак, запустим конфигуратор TCP/IP и зададим IP-адрес и маску подсети нашей машины (рис. 2.1).

Затем перейдем на вкладку **Network** и введем имя нашей ЭВМ, адреса шлюза и сервера имен (рис. 2.2).

Кстати, имя ЭВМ, которое мы ввели (в данном случае — `myname`), будет использоваться не только протоколами TCP/IP, но и собственным протоколом QNX — Qnet. Замечу, что никаких других настроек для Qnet не требуется, но об этом потом.

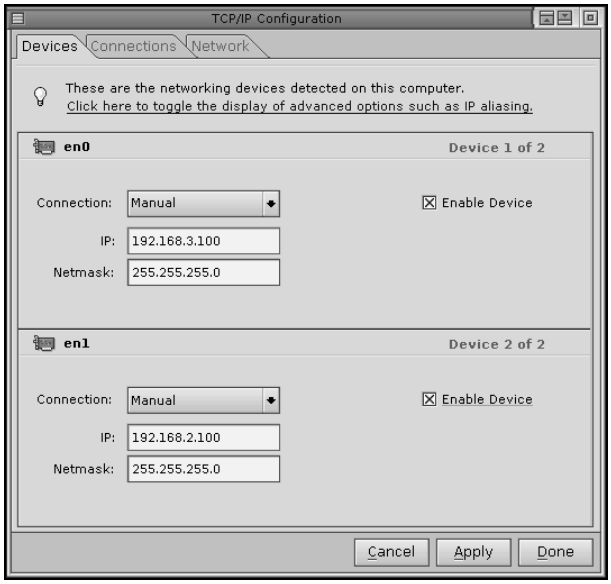


Рис. 2.1. Окно **TCP/IP Configuration**, вкладка **Devices**

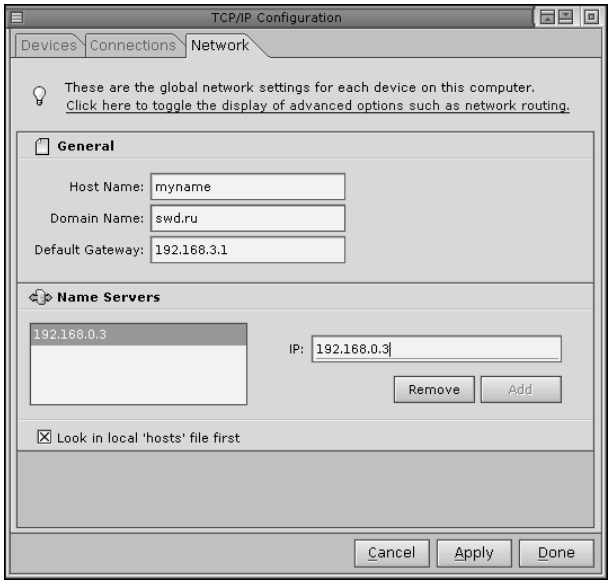


Рис. 2.2. Окно **TCP/IP Configuration**, вкладка **Network**

Теперь можно перейти к настройкам локализации. Конечно, локализация QNX далека от совершенства, но кое-что есть. Сначала запустим программу User's Configuration, для чего в меню рабочего стола нажмем кнопку **Localization**. В окне программы можно выбрать соответствующую вкладку для установки часового пояса, времени и даты. Вкладку **Language** можете не трогать — русского языка там нет. Зато есть замечательная вкладка **Keyboard**. Откройте эту вкладку и выберите строку **Russian**. Не забудьте нажать кнопки **Apply** (Применить изменения) и **Done** (Закрыть окно).

А теперь придется немного "поработать ручками". Дело в том, что программа User's Configuration считает, что пользователь использует либо английскую, либо русскую раскладку. А нам все же нужна английская раскладка, поэтому надо выполнить нехитрые действия: открыть файл `/etc/system/trap/.KEYBOARD.myname` (обратите внимание: `myname` — это имя нашей машины) и отредактировать его так, чтобы он содержал две строчки:

```
en_US_101.kbd
ru_RU_102.kbd
```

Обратите внимание, что имя файла начинается с точки (.). Это означает, что файл скрытый и средства просмотра файловой системы по умолчанию его не отображают. Чтобы видеть скрытые файлы в штатном файловом менеджере графической среды Photon, необходимо в меню **Edit** выбрать элемент **Preferences** и в открывшемся окне снять самый верхний флажок **Hide 'dot' Files** (Скрывать файлы с точкой), *см. гл. 3.*

Теперь при запуске приложений Photon'а можно будет вводить как русский, так и английский текст (для переключения между раскладками клавиатуры используется комбинация левых клавиш <Alt> + <Shift>). Однако в командной строке вы не сможете ни вводить, ни читать русские буквы. Для решения этой проблемы используется пакет SWD Cyrillic Pack, который устанавливается с помощью стандартного инсталлятора QNX.

Ну что ж, теперь на ЭВМ установлена полноценная ОС QNX.

Инсталляция дополнительного программного обеспечения

Разнообразные дополнительные программы для QNX Neutrino, как коммерческие, так и свободно распространяемые, обычно поставляются в виде специальных пакетов, хранящихся в репозиториях (подробнее о структуре репозиториев и пакетов рассказывается в *гл. 7*).

Установку пакета можно выполнять как в командной строке (утилитой **`cl-installer`**), так и в среде графической оболочки Photon, для чего следует запустить программу-инсталлятор QNX Software Installer (**`qnxinstall`**). Графическая утилита доступна через меню **Launch** (рис. 2.3).

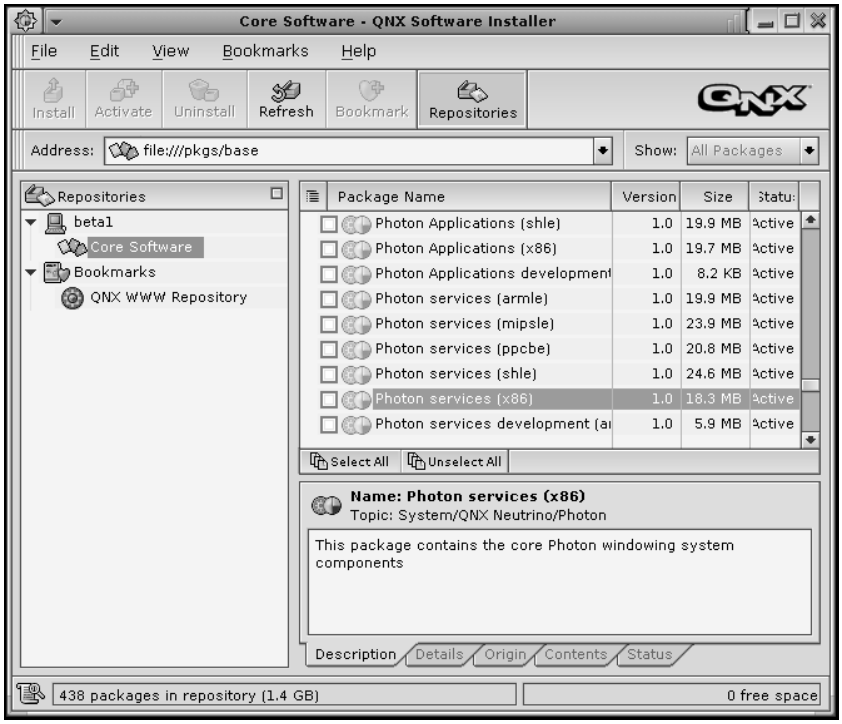


Рис. 2.3. Окно программы QNX Software Installer

Программе-инсталлятору все равно, где находится репозитарий или отдельный пакет — в сети или на локальной ЭВМ. На страничке сайта QSS **http://www.qnx.com/developer/community/repositories.html** содержатся сведения о некоторых общедоступных репозитариях с программами для QNX Neutrino. Для того чтобы задавать инсталлятору информацию о них вручную, можно в меню **File** выбрать элемент **Find Web Repositories**. Инсталлятор загрузит с сервера QSS информацию об известных репозитариях. Если щелкнуть на имени репозитария, то инсталлятор скачает с соответствующего сервера информацию о доступных пакетах. Самый содержательный репозитарий — QNX WWW Repository. Щелкнем на его названии. Инсталлятор начнет скачивать из Интернета архив, содержащий сведения о пакетах, размещенных на сервере. Для примера щелкнем на имени пакета AbiWord (x86) и нажмем кнопку **Install** (рис. 2.4).

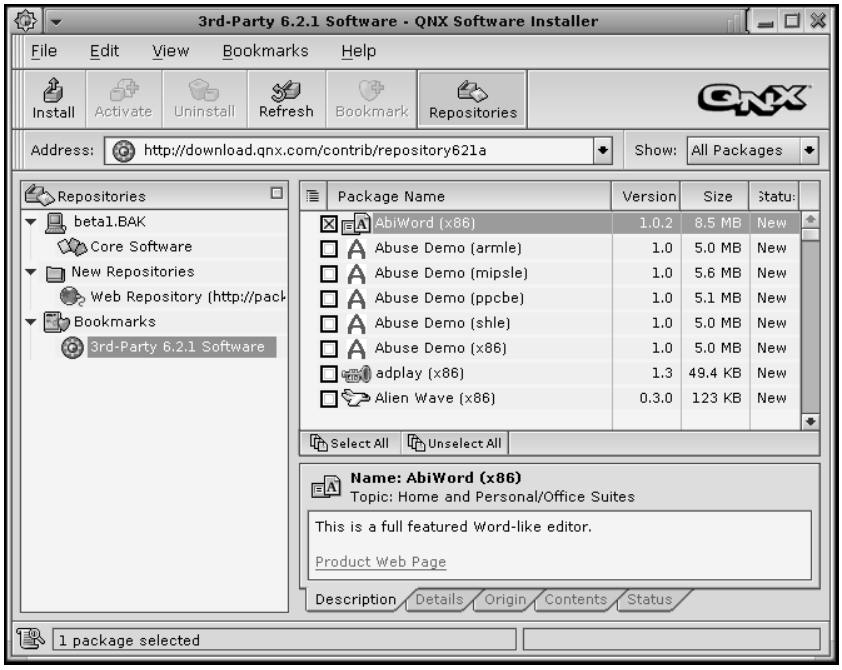
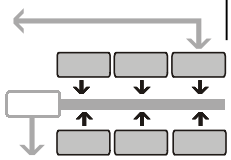


Рис. 2.4. Установка пакета AbiWord (x86)

Далее следуем указаниям инсталлятора. Сначала инсталлятор на всякий случай сообщит, какие пакеты и какого суммарного размера он собирается установить, затем предложит принять/отклонить условия лицензий. Некоторые пакеты могут потребовать ввести лицензионный ключ. Поскольку пакет скачивается из Сети, в зависимости от размеров пакетов и скорости вашего канала установка может занять, скажем так, разное количество времени. После окончания инсталляции флажок в строке с названием пакета будет снят, а состояние (**Status**) изменится с **New** на **Active**.

Таким же образом устанавливают обновления и патчи, распространяемые QSS. Информация о доступных пакетах содержится на страничке **http://www.qnx.com/developer/download/updates/**.



Глава 3

Работа в QNX Neutrino

Для тех, кто еще не работал ни в одной из UNIX-подобных операционных систем, необходимо будет освоить некоторые несложные действия. Для тех же, кто знает, что такое Motif или OpenLook, нового будет мало.

Работу в среде QNX можно рассматривать по нескольким аспектам:

- загрузка инструментальной системы;
- вход пользователя в систему и выход из нее;
- замечания по работе в QNX;
- работа в графической среде Photon microGUI.

Загрузка инструментальной системы

После включения питания ЭВМ, на которой инсталлирована ОСРВ QNX, мультизагрузчик предлагает выбрать ОС, которую следует загрузить. Вернее, он предлагает выбрать раздел диска, с которого следует выполнять загрузку. На моей машине по умолчанию грузится QNX, это выглядит так:

```
Press F1-F4 for select drive or select partition 1,2,3,4? 1
```

(Нажмите клавишу <F1>, <F2>, <F3> или <F4> для выбора дисковода или выберите раздел 1, 2, 3 или 4? 1)

Единица после вопросительного знака — это вариант, предлагаемый по умолчанию. Следует заметить, что в QNX обычно используется два загрузочных образа: основной образ помещается в файл `/.boot`, а резервный — в файл `/.altboot`. Загрузчик предлагает нажать клавишу <Esc> для загрузки резервного образа:

```
Hit Esc for .altboot
```

Если ничего не нажимать, то загружаться будет основной образ. Затем будет предложено нажать клавишу <пробел> для ввода опций загрузки:

```
Press the space bar to input boot options
```

Если ничего не нажимать, загрузка продолжится в автоматическом режиме. В противном случае на экране появится сообщение:

```
F1       Safe modes
F5       Start a debug shell after mounting filesystems
F6       Be Verbose
F7       Mount read-only partitions read/write if possible
F8       Enable a previous package configuration
F9       Target output to debug device defined in startup code
F10      Force a partition install
F11      Enumerator disables
F12      Driver disables
Enter    Continue boot process
Please select one or more options via functional keys.
Selection?
```

(<F1> — загружаться в "безопасном режиме";

<F5> — запустить командный интерпретатор после подключения файловых систем;

<F6> — установить режим подробного вывода диагностических сообщений;

<F7> — попытаться подключить с возможностью записи разделы, доступные только для чтения;

<F8> — вернуться к предыдущей конфигурации пакетов;

<F9> — выводить диагностическую информацию на устройство, указанное в модуле `startup`;

<F10> — сразу приступить к инсталляции системы;

<F11> — отключить автоматическое распознавание устройств;

<F12> — отключить драйверы;

<Enter> — продолжить загрузку.

Выберите, пожалуйста, одну или более опций, нажав соответствующие функциональные клавиши.

Ваш выбор?)

Нажатие клавиши <F1> позволяет использовать несколько вариантов загрузки системы с ограниченной функциональностью. Опция <F10> имеет смысл только при загрузке с инсталляционного компакт-диска — без этой опции можно просто загрузить систему с того же компакт-диска. Опция <F11> позволяет отключить автоопределение некоторых типов устройств, а <F12> — отключить некоторые типы драйверов.

В ходе загрузки выполняется ряд сценариев и запускаются различные программы — мы поговорим об этом в главе, посвященной построению целевых систем.

Если нужно, чтобы при запуске выполнялись дополнительные команды, эти команды помещают в сценарий `/etc/rc.d/rc.local` (по умолчанию он не существует). На моей рабочей станции, например, из сценария `rc.local`, запускается процесс **`inetd`** ("суперсервер" TCP/IP).

По завершении загрузки на экране появляется приглашение ввести регистрационное имя (login) и пароль (password). Это приглашение может быть командно-строковым или графическим.

Вход пользователя в систему и выход из нее

Итак, вход пользователя в систему (а правильнее сказать, регистрация пользователя в системе) может выполняться двумя способами: в командной строке или через графический интерфейс. То, какой способ предложит система, зависит от того, установлен ли соответствующий флажок при настройке видеодрайвера при первом старте QNX после инсталляции *(см. гл. 2)*. Если вы не установили флажок (т. е. запретили QNX автоматически запускать Photon), то в каталоге `/etc/system/config` будет создан пустой файл с именем `nophoton`. Во время загрузки системы стартовые скрипты просто-напросто проверяют, существует ли файл `nophoton`.

При командно-строковой регистрации, если введены правильные имя пользователя и пароль (а при первом входе, как вы помните, существует только пользователь **`root`**, не имеющий пароля), то стартует командный интерпретатор shell. Этот интер-

претатор называют *входным* (login shell). В нем можно запускать различные команды, в том числе, можно запустить графическую среду Photon командой **ph** (при этом пароль вводить не потребуется).

Кстати, каким образом Photon определяет, требовать у вас регистрацию или нет? Очень просто: он проверяет значение переменной окружения LOGNAME. Если вы зарегистрировались посредством **login**, то переменная LOGNAME будет содержать ваше регистрационное имя. Если же Photon был запущен автоматически, то, разумеется, значение переменной LOGNAME еще не задано. Отсюда вытекает интересная возможность для встроенных систем с графическим интерфейсом — можно задать значение переменной LOGNAME в стартовых сценариях и сразу запускать Photon от имени какого-либо пользователя.

Для выхода из системы есть следующие способы:

- в графической среде — выбрать элемент **Shutdown** меню **Launch** (запустится утилита **phshutdown**);
- в login shell — выполнить команду **logout** или **exit**;
- выполнить команду **shutdown**.

Утилиту **shutdown** может использовать только суперпользователь (пользователь **root**). Она посылает сигнал SIGPWR (Сейчас будет отключено питание) всем выполняющимся процессам, десять секунд ожидает их завершения и перезагружает систему. Если команде **shutdown** указать опцию **-f** (т. е. fast — "быстро"), то время ожидания будет сокращено до одной секунды.

Возможно несколько вариантов поведения утилиты **shutdown**, задаваемых опцией **-S**:

- **system** — остановить систему для выключения питания;
- **reboot** — перезагрузить систему;
- **photon** — закрыть графическую оболочку Photon и продолжать работу в "черном экране";
- **user** — закончить пользовательский сеанс. При этом на экране появится регистрационное приглашение для входа в систему.

Если существует файл /var/log/wtmp, то утилита **shutdown** произведет в нем запись о своей работе.

Утилита **phshutdown** по своей функциональности аналогична утилите **shutdown**. Она тоже посылает сигнал SIGPWR всем выполняющимся процессам, десять секунд ожидает их завершения и перезагружает систему, однако выполнять ее имеет право любой пользователь. Чтобы запретить использование этой утилиты всем пользователям, кроме **root**, необходимо создать пустой файл **/usr/photon/config/phshutdown.restrict**. Если существует файл /var/log/wtmp, то утилита **phshutdown** произведет в нем запись о своей работе.

Итак, **phshutdown** предлагает выбрать один из трех вариантов действий:

- завершить сеанс Photon с выходом в командную строку;
- остановить систему. К сожалению, в QNX не реализовано автоматическое выключение блока питания, хотя все необходимое для этого имеется. Поэтому после появления на экране сообщения о том, что ОС QNX завершила работу, вам следует, как в старые добрые "доATX-овые" времена, нажать кнопку Power;
- перезагрузить систему.

Замечания по работе в QNX

Работают в QNX двумя способами: в командной строке или в графической оболочке. При работе в графике командная строка все равно доступна посредством программы псевдотерминала **pterm**.

Интерфейс командной строки предоставляет программа, именуемая "командным интерпретатором". В UNIX-подобных системах это весьма мощное средство управления системой, которым советую не пренебрегать. Описание некоторых методов работы в Korn Shell (основном интерпретаторе QNX) содержится в *гл 11.*

Поскольку графическая оболочка далеко не всегда доступна (при удаленной работе, нехватке ресурсов и т. п.), администратору необходимо уметь пользоваться редактором **vi**. Не пожалейте немного времени — иногда это единственный доступный инструмент для редактирования файлов конфигурации.

Редактор `vi` запускается очень просто: `vi` ***имя_файла***. Если файл с именем ***имя_файла*** не существует, то он будет создан. Редактор может находиться в одном из двух режимов: в режиме ввода или в командном режиме. После запуска редактор находится в командном режиме. Переход из командного режима в режим ввода осуществляется нажатием клавиши <I> (input) или <A> (add). После нажатия клавиши <I> текст будет вводиться с текущей позиции курсора, а после нажатия клавиши <A> текст будет вводиться со следующей позиции после курсора. Возврат в командный режим осуществляется нажатием клавиши <Esc>.

Освоение работы в редакторе `vi` сводится к запоминанию команд. Надо сказать, что возможностей у `vi` достаточно много. Если вы хотите хорошо изучить этот редактор, то можете прочитать соответствующую литературу. А для выполнения базовых операций используют лишь несколько команд, приведенных в табл. 3.1.

Таблица 3.1. *Основные команды редактора* `vi`

Команда	Назначение
`i`	Перейти в режим ввода (insert)
`a`	Перейти в режим ввода (add)
`yy`	Скопировать текущую строку в буфер
`{`*n*`}yy`	Скопировать *n* строк, начиная с текущей, в буфер
`dd`	Удалить текущую строку в буфер
`{`*n*`}dd`	Удалить *n* строк, начиная с текущей, в буфер
`p`	Вставить содержимое буфера
`x`	Удалить текущий символ
`G`	Перейти в конец файла
`{`*n*`}G`	Перейти на строку с номером *n*
`/`***шаблон***	Поиск шаблона ***шаблон***
`n`	Повторить поиск
`:w`	Сохранить изменения

Таблица 3.1 *(окончание)*

Команда	Назначение
:w *файл*	Сохранить изменения в файле ***файл*** ("Сохранить как...")
:q	Выход
:q!	Выход без сохранения изменений

Так что ничего сложного или страшного в работе с редактором **vi** нет. По правде сказать, в QNX используется не **vi**, а его модернизированная версия **elvis**; **vi** — просто символическая ссылка на него (что такое символическая ссылка, мы обсудим в *гл. 4*). Многие программисты и администраторы предпочитают другой клон **vi** — Vi IMprovement (**vim**). Одно из достоинств **vim** — поддержка выделения синтаксиса для разных языков программирования (для этого в командном режиме введите **:syntax on**).

При работе в командной строке важную роль играют так называемые *переменные системного окружения*. Список этих переменных (табл. 3.2) можно получить с помощью команды **set**.

Таблица 3.2. *Переменные системного окружения*

Имя переменной	Что означает переменная
HOME	Домашний каталог пользователя
PATH	Перечень каталогов для поиска запускаемых программ
LD_LIBRARY_PATH	Перечень каталогов для поиска динамических библиотек
IFS	Внутренний разделитель полей
LOGNAME	Имя пользователя
PWD	Имя текущего каталога
PS1	Вид первичного приглашения
PS2	Вид вторичного приглашения (для ввода незаконченной команды)
UID	Идентификатор пользователя
TZ	Временной пояс
?	Код завершения последней выполненной команды

Изменять значения переменных в интерпретаторе Korn Shell очень просто — надо выполнить присваивание:

```
ИМЯ_ПЕРЕМЕННОЙ=значение_переменной
```

Однако присвоенное таким образом значение будет иметь смысл только в данной работающей оболочке. Для того чтобы переменная распространялась на все запущенные процессы, используют такую команду:

```
export ИМЯ_ПЕРЕМЕННОЙ
```

Можно экспортировать переменную непосредственно при создании:

```
export ИМЯ_ПЕРЕМЕННОЙ=значение_переменной
```

Посмотреть значение одной из переменных (чтобы не выводить весь список) можно такой командой:

```
echo $ИМЯ_ПЕРЕМЕННОЙ
```

При запуске в качестве входной оболочки (login shell) командный интерпретатор Korn Shell выполняет команды, содержащиеся в файле `/etc/profile`, затем команды, содержащиеся в файле `$HOME/.profile` (разумеется, если эти файлы существуют и доступны для чтения). Для того чтобы при входе в систему автоматически устанавливались необходимые переменные, как вы уже догадались, следует откорректировать файл `.profile`, находящийся в вашем домашнем каталоге. Для этого можно, например, воспользоваться редактором **`vi`** ☺.

Если вы пишете программы, то, пожалуй, лучше воспользоваться редактором **`vim`** (Vi Improvement) — он поддерживает выделение синтаксиса для различных языков программирования.

Работа в графической среде Photon microGUI

Итак, вы вошли в систему через **`phlogin`** или запустили команду **`ph`**, войдя через **`login`**. Вы оказываетесь в рабочем пространстве (workspace) графической оболочки Photon. Как работать в графической оболочке, интуитивно понятно. Снизу находится панель задач, в левой ее части находится кнопка **Launch**. При нажатии кнопки **Launch** появляется меню, позволяющее вызывать

различные прикладные программы. В правой части экрана находится меню быстрого запуска приложений — **Shelf**. Следует обратить внимание на следующие программы:

- Photon Terminal (`pterm`) — штатный псевдотерминал, позволяющий работать с инструментами командной строки (в том числе, с нашим старым знакомым `vi`);
- Helpviewer — программа доступа к штатной электронной документации. Надо сказать, что в составе QNX поставляется достаточно обширная и подробная документация;
- Photon File Manager (`pfm`) — штатный файловый менеджер. Для любителей Norton Commander есть продукт "третьего" производителя `mqc` (MiShell QNX Commander) — аналог Midnight Commander для Linux.

Стоит сказать пару слов про конфигурирование файлового менеджера `pfm`. Чтобы задать для файлового менеджера некоторые настройки (например, запрашивать или нет подтверждение при удалении файла) следует выбрать элемент **Preferences** меню **Edit** — откроется соответствующее окно (рис. 3.1).

Рис. 3.1. Окно **Preferences**

Обычно я снимаю первый сверху флажок, предписывающий файловому менеджеру не показывать файлы, имя которых начинается с точки, и устанавливаю второй сверху флажок, предписывающий сначала выводить список каталогов, а затем список файлов.

Вспомним, что любой приличный файловый менеджер умеет что-то делать при двойном щелчке на имени файла. Если это "что-то" (назовем его "обработчиком по умолчанию") нам не подходит, то на имени файла можно щелкнуть правой кнопкой мыши и в появившемся меню выбрать подходящий элемент (читайте — "обработчик"). Файловый менеджер `pfm` позволяет изменять обработчики, соответствующие элементам меню **Open**, **View** и **Edit**. Обработчик **Open** является обработчиком по умолчанию, т. е. вызывается двойным щелчком на имени файла. Итак, для настройки обработчиков выберем элемент **Associations** меню **Edit** — откроется соответствующее окно (рис. 3.2).

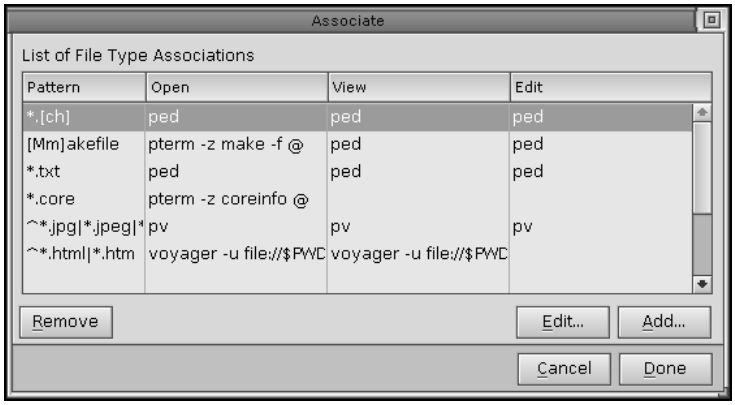


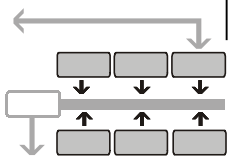
Рис. 3.2. Окно **Associate**

Окно ассоциаций представляет собой таблицу, в которой для файлов с различными расширениями задаются программы-обработчики. Для добавления новых ассоциаций нажмем кнопку **Add**. В открывшемся окне заполним соответствующие поля. Если нужно установить одинаковые обработчики для файлов с разными расширениями, то в поле **Pattern** (Шаблон) можно перечислить шаблоны имен через логическое "И" (знак конъ-

юнкции — "|"). И, будьте любезны, не забывайте нажимать кнопку **Done** (уж очень часто пользователи забывают это делать).

Следующий элемент графической оболочки, который можно конфигурировать, — это workspace-меню. Это то самое меню, которое появляется при щелчке правой кнопкой мыши в рабочем пространстве экрана. Для его конфигурирования используйте утилиту **phmenu**.

Есть весьма полезный файл `$HOME/.ph/phapps` — это командный файл, в котором можно перечислить приложения, которые Photon должен запускать автоматически при своей загрузке. Я таким образом запускаю программу чтения почты и программу для обмена сообщениями с друзьями через Интернет. Файл `phapps` должен иметь атрибут "исполняемый" для владельца.



Глава 4

Файлы и каталоги

Все данные в операционной системе хранятся в виде файлов. Одной из важнейших функций любой ОС является способность манипулировать файлами, размещенными на различных физических носителях (магнитных дисках, микросхемах ПЗУ и т. п.). В этой главе мы рассмотрим:

- типы файлов, поддерживаемые в QNX;
- разграничение доступа к файлам;
- файловую систему QNX4;
- монтирование файловых систем;
- диагностику файловой системы.

Типы файлов

Файл — это набор байтов, имеющий общие атрибуты:

- имя файла;
- идентификатор владельца и идентификатор группы;
- атрибуты доступа (для владельца, для членов группы и для остальных пользователей);
- метки времени (время создания файла, время последней модификации файла, время последнего доступа к файлу, время последней записи в файл);
- тип файла;
- счетчик ссылок;
- другие атрибуты.

Имя файла с добавлением списка разделенных символом "/" имен вложенных каталогов, содержащих файл (начиная с корневого каталога), называется *полным* или *путевым именем файла*. Например, `/usr/photon/bin/slideviewer`. Здесь файл с именем `slideviewer` имеет путь `/usr/photon/bin`.

QNX обеспечивает поддержку следующих типов файлов:

- обычные (regular) файлы;
- каталоги;
- жесткие ссылки;
- символические ссылки;
- именованные программные каналы (FIFO);
- блок-ориентированные специальные файлы;
- байт-ориентированные специальные файлы;
- именованные специальные устройства (Named Special Device).

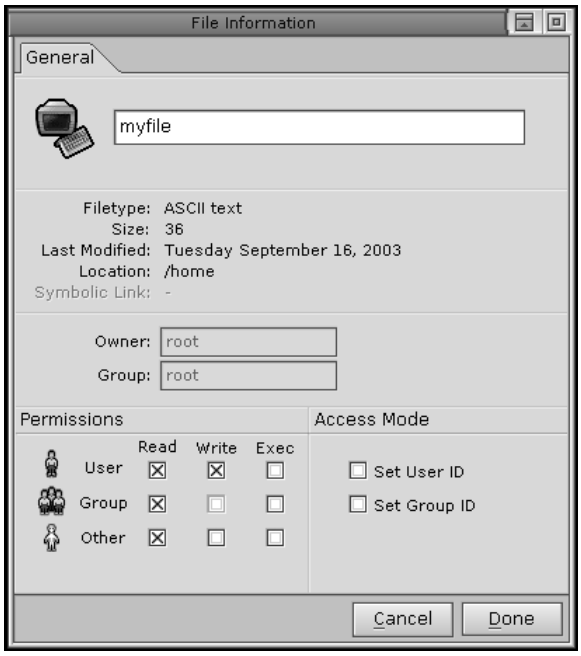


Рис. 4.1. Окно **File Information**

Расширенную информацию о файлах можно посмотреть, например, выполнив команду `ls -l`. При этом самый первый символ строки, соответствующей каждому файлу, обозначает тип файла. Для просмотра той же информации в файловом менеджере нужно щелкнуть правой кнопкой мыши на имени файла и выбрать элемент **Inspect** в появившемся меню. Откроется диалоговое окно, показанное на рис. 4.1.

Это окно позволяет изменять некоторые атрибуты.

Обычные файлы

Файл этого типа — последовательность байтов, не имеющая (с точки зрения QNX) предопределенной структуры. За интерпретацию содержимого обычных файлов отвечают конкретные приложения. В сложившейся практике приложение "узнает" свой файл по расширению его имени (т. е. по части имени файла, идущей после точки). Например: `myfile.c` — это исходный текст программы на языке C, `myfile.pdf` — это документ в формате PDF (Portable Document Format) фирмы Adobe. В отличие от операционных систем семейства Windows, QNX узнает исполняемые файлы не по расширению (вроде `exe`), а по специальному атрибуту. Для того чтобы при двойном щелчке мышью на имени файла в окне файлового менеджера Photon (**`pfm`**) автоматически запускалось нужное приложение, необходимо добавить соответствующую ассоциацию *(см. гл. 3)*.

Примечание

При выполнении команды `ls -l` обычный файл будет обозначен символом `-`.

Каталоги

Каталоги — это, по сути дела, обычные файлы, имеющие определенную структуру. Каталог представляет собой набор записей определенного формата, называемых *элементами каталога*. Первым элементом каталога всегда является запись о файле с именем "`.`". Этот элемент является ссылкой "на самого себя", указывая на соответствующую самому себе запись в родительском

каталоге. Вторым элементом каталога всегда является запись о файле с именем ". .". Этот элемент ссылается на родительский каталог, указывая на его первый элемент. Таким образом, первые два элемента каталога всегда существуют и имеют известное содержимое. Если каталог является корневым для данного физического раздела, то оба эти элемента ссылаются на собственный каталог.

Примечание

На всякий случай напомню, что штатные средства просмотра файловой системы по умолчанию не отображают файлы (а каталог, как мы уже выяснили, это просто тип файла), имена которых начинаются с точки (.). Такие файлы называют "скрытыми" и обычно в них содержится системная и конфигурационная информация, а "спрятаны" они для повышения "дуракоустойчивости" операционной системы.

Каждый элемент каталога связывает имя некоторого файла со служебной информацией о нем, включающей ссылку на место физического хранения данных. Содержимое элемента каталога можно представить так:

- 16 байт для имени файла;
- размер файла;
- информация о физическом размещении содержимого файла на диске;
- метки времени;
- атрибуты доступа;
- счетчик ссылок на физические данные;
- тип файла;
- статус ("закрыт" или "открыт").

Вы, конечно же, заметили, что для имени файла в элементе каталога отведено всего 16 байт. Что же делать, если требуется задать для файла более длинное имя? В файловой системе QNX (о которой мы еще поговорим) есть служебный файл `/.inodes` (Information Nodes — "информационные узлы", эти "узлы" — дальние родственники индексных узлов UNIX). Если длина имени какого-нибудь файла превысит 16 символов (т. е. 16 байт),

то в файле /.inodes будет создана запись для этого файла, в которую переместится вся информация о файле, кроме имени. В элементе каталога, относящемся к данному файлу, останутся имя файла (теперь его длина может достигать 48 символов) и, разумеется, ссылка на запись в файле /.inodes.

С широким распространением платформы Java 2 Micro Edition (Sun Microsystems) возникла необходимость поддержки во встраиваемых системах еще более длинных имен файлов. Разработчики QSS добавили в версии 6.2.1 очень простое решение: "лишняя" часть имен файлов помещается в файл .longfilenames, аналогичный файлу .inodes. Это позволяет задавать имена, состоящие из 505 символов. Если файловую систему с такими длиннющими именами смонтировать в версии QNX Neutrino, которая еще не знала о существовании подобных имен, то "урезание" будет выполняться тем же способом, каким DOS урезает длинные имена Windows.

Примечание

При выполнении команды `ls -l` каталог будет обозначен символом `d`.

Жесткие ссылки

Хорошо, ну а если для одних и тех же физических данных файла создать еще один элемент каталога? Да хоть дюжину! Такие дополнительные элементы каталога (т. е. имена) называют *жесткими ссылками* (или "жесткими связями"). При создании жесткой ссылки, во-первых, информация о физическом размещении данных выносится в файл /.inodes, во-вторых, счетчик ссылок (атрибут файла) увеличивается на единицу. При удалении одной из жестких ссылок реально будет удален только соответствующий элемент каталога, а счетчик ссылок на inode-запись будет уменьшен на единицу. Как только счетчик достигнет значения "ноль", и inode-запись, и физические данные файла будут уничтожены. Справедливости ради заметим, что для уничтожения файла есть еще одно обязательное условие — статус файла должен иметь значение "закрыт".

Вы можете спросить, удаляется ли inode-запись, если счетчик стал равен единице или если длина имени файла уменьшилась, скажем, до десяти символов? Нет, не удаляется. Созданная inode-запись сохраняется, пока существует файл.

Примечание

Нельзя создавать жесткие связи для каталогов, кроме уже существующих — "." и "..".

Примечание

При выполнении команды `ls -l` жесткая ссылка обозначается так же, как файл, на который она ссылается, при этом счетчик ссылок будет иметь значение больше 1.

Символические ссылки

Еще один весьма полезный тип файлов — *символические ссылки* (в UNIX-системах их обычно называют "мягкими" ссылками). Это, по сути дела, текстовый файл, содержащий имя другого файла или каталога, к которому перенаправляются все запросы ввода/вывода.

У символической ссылки есть важное достоинство — она может указывать на файл или каталог, находящийся на другом физическом носителе (например, в другом разделе диска или на другом узле сети).

Возможность создания символических ссылок для каталогов создает опасность бесконечных циклов. Поэтому число переходов по символическим ссылкам ограничено значением переменной `SYMLOOP_MAX`, определенным в заголовочном файле `<limits.h>`.

Примечание

При выполнении команды `ls -l` символическая ссылка обозначается символом `l`, при этом к имени файла добавляется стрелка с именем того файла, на который сделана ссылка.

Именованные программные каналы (FIFO)

Именованные программные каналы (FIFO) предназначены для организации взаимодействия между двумя или более процессами: один процесс пишет в программный канал, другой читает из программного канала.

Скажем прямо, FIFO — далеко не самый быстрый способ межзадачного взаимодействия. Но он не лишен достоинств: во-первых, данные, записанные в FIFO, сохраняются при отключении питания, во-вторых, взаимодействующие процессы не должны передавать друг другу никакие дескрипторы или идентификаторы (но им должно быть известно имя FIFO-файла).

Примечание

При выполнении команды `ls -l` FIFO-файл будет обозначен символом `f`.

Блок-ориентированные специальные файлы

Блок-ориентированные специальные файлы (очень часто их называют "блочными устройствами") — файлы, предназначенные для того, чтобы скрыть от приложений физические характеристики аппаратуры. Слово "блочный" говорит о том, что обмен данными осуществляется блоками по несколько байт (например, при работе с жестким диском обычный размер блока — 512 байт). В QNX блок-ориентированные специальные файлы создаются не на диске, а в оперативной памяти. Когда же они создаются? При старте соответствующих драйверов. Например, драйвер EIDE создает блок-ориентированные файлы: `/dev/hd0` — для доступа к первому жесткому диску, `/dev/hd1` — для доступа ко второму жесткому диску и т. д.

Примечание

При выполнении команды `ls -l` блочное устройство будет обозначено символом `b`.

Байт-ориентированные специальные файлы

Байт-ориентированные специальные файлы (очень часто их называют "символьными устройствами") — это файлы, аналогичные блочным устройствам, с той разницей, что символьные устройства обеспечивают интерфейс к аппаратуре, осуществляющей посимвольный ввод/вывод. К такой аппаратуре относятся, например, последовательный порт, сетевая карта и т. п. Так же как блочные устройства, байт-ориентированные специальные файлы создаются драйверами при запуске. Например, драйвер Ethernet-карты создает файл `/dev/en0`.

Примечание

При выполнении команды `ls -l` символьное устройство будет обозначено символом **c**.

Named Special Device — именованные специальные устройства

Этот тип специфичен для QNX. Дело в том, что благодаря своей универсальности байт- и блок-ориентированные специальные файлы могут использоваться приложениями не только для обмена данными с драйверами устройств, но и для взаимодействия с другими программами. В этих случаях приложения, создающие специальные файлы, будут являться как бы программными устройствами. Поскольку не всегда данные при обмене удобно представлять в виде символов и блоков, потребовалось ввести специальный дополнительный тип файлов — Named Special Device. Разумеется, обмен посредством этого типа файлов требует знания формата данных от всех участников обмена. Например, для взаимодействия компонентов графической системы используется специальный файл `/dev/photon`.

Примечание

При выполнении команды `ls -l` такие именованные специальные устройства будут обозначены символом **n**.

Разграничение доступа к файлам

Доступом к файлам управляют с помощью проверки прав доступа и изменения атрибутов файла.

Проверка прав доступа

Каким образом регулируется возможность приложений выполнять операции чтения/записи с файлами? Каждая выполняющаяся программа (т. е. процесс) имеет четыре идентификатора:

- идентификатор владельца (UID);
- идентификатор группы (GID);
- эффективный идентификатор владельца (EUID);
- эффективный идентификатор группы (EGID).

Про эти атрибуты мы еще поговорим в *гл. 5*, посвященной процессам. Сейчас достаточно знать то, что для проверки прав доступа к файлу используются только эффективные идентификаторы.

Нам уже известно, что каждый файл имеет атрибуты доступа для трех "личностей" (см. рис. 4.1): владельца (**User**), группы (**Group**) и "остальных" (**Other**). Для каждой из этих "личностей" можно задавать права на чтение, запись и исполнение.

Итак, в разграничении доступа к файлам участвуют EUID и EGID процесса и файловые атрибуты доступа. Когда процесс пытается открыть для записи какой-либо файл, QNX сначала смотрит, совпадает ли EUID процесса с идентификатором владельца файла. Если совпадает, то проверяется, есть ли у владельца право на запись — если есть, то файл открывается с разрешением на запись, если нет, — процесс получит уведомление `Permission Denied` (Доступ запрещен). Конечно, EUID может не совпасть с идентификатором владельца файла. Тогда проверяется, совпадает ли EGID процесса с файловым идентификатором группы. Если никакие идентификаторы не совпали, то к данному процессу применяются атрибуты доступа, установленные для "остальных".

Примечание

Для процессов с EUID = 0 (т. е. для пользователя `root`) доступ к файлам предоставляется без процедуры проверки прав. Разумеется, это не означает, что пользователь `root` может запустить на исполнение неисполняемый файл ☺.

В диалоговом окне **File Information** (см. рис. 4.1) есть два флажка (**Set User ID** и **Set Group ID**), соответствующих дополнительным атрибутам SUID и SGID (эти атрибуты имеют смысл только для исполняемых файлов). В выходной информации команды **`ls -l`** эти атрибуты отображаются путем замены **`x`** на **`S`** в атрибутах владельца (для SUID) и в атрибутах группы (для SGID). Если же снять доступ по исполнению при сохранении флага SUID или/и SGID, то соответствующая заглавная буква **`S`** будет заменена строчной буквой **`s`**.

Назначение атрибутов SUID и SGID заключается в следующем. При запуске процесса его идентификаторы UID, GID, EUID, EGID устанавливаются равными соответствующим идентификаторам родительского процесса. Если же для исполняемого файла установлен атрибут SUID, то идентификаторы UID и EUID нового процесса будут установлены равными идентификатору владельца файла. Атрибут SGID делает то же с групповыми идентификаторами процесса. То есть если существует исполняемый файл `myfile`, принадлежащий пользователю `user1`, с атрибутами, разрешающими исполнение файла всем пользователям, то у процесса, созданного при запуске утилиты `myfile` пользователем `user2`, значения UID и EUID будут равны не `user2`, как следовало бы ожидать, а `user1`.

Нетрудно догадаться, что такие флаги весьма небезобидны с точки зрения защиты информации. Но иногда без них не обойтись, яркий пример — утилита **`passwd`** *(см. гл. 6, разд. "Добавление и удаление пользователей и их групп").*

Изменение атрибутов файла

Изменять атрибуты файла может либо его владелец, либо, конечно, суперпользователь `root`. Обратите внимание: для того чтобы так "хозяйничать", необходимо иметь право записи в ка-

талог, содержащий этот файл. Для выполнения изменений следует воспользоваться файловым менеджером (см. рис. 4.1) или утилитами командной строки **chown** (для изменения владельца и группы), **chgrp** (для изменения группы) или **chmod** (для изменения атрибутов доступа).

Файловая система QNX4

Теперь самое время разобрать то, что мы именовали "собственный раздел QNX". На самом деле такой тип раздела жесткого диска называется *файловой системой QNX4*. Почему 4? Потому что структура и организация данных в файловой системе QNX Neutrino не отличается от QNX4.*xx* (хотя программная реализация работы с этими структурами данных, разумеется, отличается).

Файловая система QNX4 соответствует стандарту POSIX, поэтому в документации QNX она часто называется *файловой системой POSIX*. Она поддерживает многопоточную обработку запросов с клиент-управляемым приоритетом.

Новый раздел QNX создается утилитой **fdisk**. Назначение этой утилиты то же, что у одноименных утилит других операционных систем.

Примечание

Для изменения физической структуры диска нужно обладать правами `root` или, как минимум, иметь доступ по записи к соответствующим файлам устройств.

Если на жестком диске предполагается создавать разделы не только QNX, то рекомендуется создать раздел QNX в последнюю очередь (хотя, пожалуй, средства редактирования физической структуры дисков в Linux достаточно гибки, так что можно не беспокоиться за загрузочные секции, созданные QNX).

Для использования утилиты **fdisk** нужно, чтобы существовал соответствующий блок-ориентированный файл диска, т. е. должен быть запущен драйвер диска. Например, драйвер интерфейса EIDE запускается так:

```
devb-eide &
```

Теперь можно создать раздел QNX:

```
fdisk /dev/hd0 add qnx half
```

В результате в дополнение к существующему разделу диска будет создан новый, использующий половину свободного пространства. Чтобы увидеть результат, надо перезапустить драйвер интерфейса EIDE:

```
slay devb-eide
devb-eide &
```

На этот раз драйвер создаст не только блок-ориентированный специальный файл для доступа ко всему диску (`/dev/hd0`), но и отдельный блок-ориентированный файл для доступа к созданному разделу QNX, например `/dev/hd0t77`. Вообще, в QNX принято такое соглашение об именовании разделов жесткого диска:

- первое число (после `hd`) — это порядковый номер жесткого диска;
- второе число (в нашем примере — `77`) означает тип раздела диска (табл. 4.1).

Таблица 4.1. *Соглашения о наименовании*

Тип	Файловая система
11	DOS 32-bit FAT; разделы свыше 2047 Гбайт
77	QNX-раздел
78	QNX-раздел (дополнительный)
79	QNX-раздел (дополнительный)
131	Linux (Ext2)

Если на диске есть два раздела FAT32, то имена их блок-ориентированных файлов будут примерно такими: `/dev/hd0t11` и `/dev/hd0t11.1`.

Созданный раздел QNX необходимо инициализировать с помощью утилиты **`dinit`**:

```
dinit -h /dev/hd0t77
```

В результате всех проделанных операций будет создан раздел QNX4, содержащий следующие ключевые компоненты:

- блок загрузчика;
- корневой блок;
- битовая матрица (или битовая карта);
- корневой каталог.

Блок загрузчика

Блок загрузчика, или загрузочный блок, — это первый физический блок в разделе диска. Этот блок содержит так называемый IPL (Initial Program Loader) — начальный загрузчик. Назначение IPL мы рассмотрим подробнее в *гл. 12*. Сейчас нам достаточно знать, что IPL — это код, который загружает образ QNX. В случае, если диск не разбит на разделы, блок загрузчика будет первым физическим блоком на диске.

Корневой блок

Корневой блок имеет структуру обычного, содержащего записи следующих файлов:

- корневой (root) каталог данного раздела QNX4. Жесткие связи `"."` и `".."` этого каталога ссылаются на него же;
- файл `/.inodes` (его назначение мы обсуждали в начале этой главы);
- файл `/.boot` содержит загружаемый образ операционной системы QNX (как раз его и загружает IPL);
- файл `/.altboot` содержит резервный загружаемый образ операционной системы QNX. Если этот файл не пуст, то в начале загрузки система предложит загрузить `/.altboot` в качестве опции (по умолчанию будет загружен образ, хранящийся в файле `/.boot` — *см. гл. 12*).

Битовая матрица

Битовая матрица, или битовая карта (bitmap), — файл, содержащий столько битов, сколько блоков в разделе диска. Каждому

физическому блоку раздела соответствует один бит. Если бит имеет значение 1, значит, соответствующий ему блок занят. Битовая матрица используется для выделения физических блоков на диске и содержится в файле `/.bitmap`.

Операция по изменению информации на диске выполняется в форме транзакции, поэтому даже при катастрофических сбоях (например, при отключении питания) файловая система QNX4 остается цельной. В худшем случае некоторые блоки могут быть выделены (т. е. отмечены в битовой матрице как 1), но не использованы. Вернуть эти блоки можно, запустив утилиту **chkfsys** (см. далее).

Корневой каталог

Корневой каталог раздела ведет себя как обычный каталог, за двумя исключениями:

- жесткие связи "." и ".." корневого каталога раздела являются ссылками на этот же корневой каталог;
- корневой каталог всегда содержит записи файлов `/.bitmap`, `/.inodes`, `/.boot` и `/.altboot`.

Примечание

Для того чтобы программа **diskboot** могла использовать раздел для поиска базового образа файловой системы QNX, необходимо наличие файла `/.diskroot` (об этом мы поговорим в *гл. 12*), а для поддержки длинных имен используется файл `.longfilenames`.

Монтирование файловых систем

Файловая система в QNX имеет иерархическую древовидную структуру. Вершиной иерархии является *корневой* каталог, обозначаемый символом "/". Поскольку ОС QNX предназначена для работы на таких ЭВМ, в которых может не быть ни жестких, ни гибких магнитных дисков, структура файловой системы не должна зависеть от физического размещения файлов. Для решения этой задачи в QNX реализован механизм *пространства путевых имен*. Подробно этот

механизм обсуждается в *гл. 5*, посвященной архитектуре QNX. Пока же нам достаточно представлять, что фрагменты единого дерева файловой системы могут размещаться на разных физических носителях, например, в разделах QNX4, находящихся на разных жестких дисках. Нам уже известно, что драйверы устройств при старте создают байт- и блок-ориентированные файлы, служащие для "сырого" доступа к устройствам. Некоторые такие файлы могут "содержать" файловые системы какого-либо типа, например файл `/dev/hd0t77` "содержит" раздел QNX4. Как же получить доступ к данным? Для этого, во-первых, нужен программный модуль, "знающий" структуру данного типа раздела диска. Такой модуль в QNX называется *администратором файловой системы* (его мы обсудим в *гл. 7*). Во-вторых, необходимо, чтобы администратор файловой системы присоединил иерархию файлов, содержащуюся в разделе диска, к единой иерархии файлов ОС (т. е. к пространству путевых имен). Эта операция называется *монтированием* файловой системы. Каталог, к записям которого будет добавлено содержимое корневого каталога монтируемого раздела, называется *точкой монтирования*. Для монтирования и демонтирования файловых систем предназначены утилиты **`mount`** и **`umount`** (обе утилиты может запускать только пользователь `root`). Следующая команда монтирует содержимое раздела, представленного блок-ориентированным файлом `/dev/hd0t77`, в каталог `/fs/QNX4`:

```
mount /dev/hd0t77 /fs/QNX4
```

А такая команда отмонтирует раздел от пространства путевых имен:

```
umount /fs/QNX4
```

Точку монтирования можно перегружать, т. е. монтировать в одну точку несколько разделов.

Диагностика файловой системы

После инициализации раздела утилитой **`dinit`** рекомендуется выполнить проверку физической целостности раздела утилитой **`dcheck`**:

```
dinit -h /dev/hd0t77
dcheck -m /dev/hd0t77
```

Утилита **dcheck** проверяет целостность каждого блока. Сбойные и предотказные блоки будут удалены из битовой матрицы (`/.bitmap`) и отмечены в специальном файле `/.bad_blks`.

Проверка *логической* целостности раздела QNX4 выполняется утилитой **chkfsys**. Эта утилита анализирует каждый файл, обнаруживая и корректируя нарушения структуры метаданных. Кроме того, утилита **chkfsys** возвращает в систему неиспользуемые блоки, помеченные как используемые (так называемые "потерянные блоки"). Для этого **chkfsys** в ходе проверки файлов создает собственную битовую матрицу. Если полученная утилитой битовая матрица будет отличаться от файла `/.bitmap`, то оператору будет предложено сохранить ее в файле `/.bitmap`.

Утилиту **chkfsys** можно запустить с опцией **-z** ***имя_файла***. Тогда в файл *имя_файла* (а он должен размещаться *за пределами* проверяемого раздела) будет помещен список файлов, содержащих сбойные блоки. Эти файлы можно удалить с помощью утилиты **zap**. Так как **zap** просто удаляет соответствующую файловую запись из каталога, можно впоследствии выполнить обратную операцию **zap -u**.

Для проверки логической целостности раздела FAT32 предназначена утилита **chkdosfsys**. По своим функциям она аналогична Windows-утилите **scandisk**.

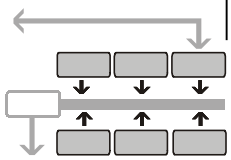
Для контроля использования дискового пространства используют утилиты:

- **df** — POSIX-утилита, показывающая использование дискового пространства смонтированных разделов;
- **du** — POSIX-утилита, показывающая размер файлов и каталогов.

Обе эти утилиты выводят информацию в блоках, поэтому удобно запускать их с флагом **-k**, тогда результат будет выводиться в килобайтах.

Для поиска файлов и каталогов используют:

- в командной строке — утилиту **find**;
- в среде Photon — пункт меню **Launch \ Utilities \ Find**.



Глава 5

Процессы и потоки

В этой главе будут рассмотрены вопросы:

- архитектура QNX;
- механизмы микроядра;
- управление процессами;
- дополнительные способы взаимодействия между потоками.

Сведения, изложенные в этой главе, очень неглубоки — следствие желания не забивать голову читателю обилием технических подробностей. Те, кому нужна подробная информация, могут найти ее как в документации, так и в очень толковых книгах Роба Кртена.

Архитектура QNX

Архитектура — это то, чем QNX, несмотря на большое внешнее сходство, отличается от операционных систем семейства UNIX. Именно архитектура делает QNX операционной системой *жесткого* реального времени.

Центральным понятием в QNX является *микроядро*. Грубо говоря, микроядро (как раз его-то и зовут Neutrino) почти ничего само не делает, а является своего рода коммутирующим элементом, к которому с помощью дополнительных программных модулей добавляется та или иная функциональность. Кроме микроядра в ОСРВ QNX есть еще один важный компонент — *администратор процессов*. Микроядро Neutrino скомпоновано с администратором процессов в единый модуль `procnto` — главный (и единственный безусловно необходимый) компонент QNX. Если нам надо, чтобы система реально делала какую-то

работу, мы должны запустить процесс, выполняющий эту работу. Программы, реализующие сервисные функции, называют *администраторами ресурсов*. Есть администраторы ресурсов, обеспечивающие доступ к дискам, сети и т. д. Все эти программы связаны воедино микроядром и слаженно взаимодействуют с помощью механизма сообщений.

Следует заметить, что существуют различные дополнительные варианты модуля `procnto` (не говоря о версиях для разных процессоров):

- `procnto-smp` — вариант модуля `procnto` с поддержкой симметричной многопроцессорности;
- `procnto-instr` — вариант модуля `procnto`, оборудованный средствами трассировки событий;
- `procnto-smp-instr` — сами догадайтесь, для чего ☺.

На самом деле функциональность ОС может расширяться не только с помощью процессов, но и с помощью динамически присоединяемых библиотек (Dynamic Link Library, DLL). Правильнее будет сказать, что как функциональность ОС расширяется с помощью процессов, так функциональность процессов расширяется с помощью DLL. Более того, администратор ресурсов может быть реализован или как программа, или как DLL.

Таким образом, в общем случае QNX состоит:

- из микроядра Neutrino;
- администратора процессов;
- администраторов ресурсов;
- прикладных программ.

Что же такое процесс? *Процесс* (process) — это выполняющаяся программа. Процесс (или "задача") включает код и данные программы, а также различную дополнительную информацию — переменные системного окружения и т. п.

Кроме процесса важным понятием является "поток управления" (thread, он же просто "поток"; раньше его иногда называли "нитью"). *Поток управления* — это фрагмент процесса, содержащий непрерывную последовательность команд, которые могут выполняться параллельно с другими потоками того же и других процессов.

Процесс является, по сути дела, контейнером потоков и содержит минимум один поток.

Для чего вообще нужны потоки? Они весьма полезны в ряде случаев, поэтому поддержка потоков — обязательное свойство POSIX-совместимых ОС. Обычно потоки используют:

- для распараллеливания задачи на многопроцессорных ЭВМ;
- для более эффективного использования процессора (например, когда один поток ожидает пользовательский ввод, другой может выполнять расчеты);
- для облегчения совместного использования данных (все потоки процесса имеют свободный доступ к данным процесса).

Механизмы микроядра

Итак, микроядро Neutrino — главный и обязательный компонент ОСРВ QNX. Оно выполняет следующие функции:

- создание и уничтожение потоков;
- диспетчеризация потоков;
- синхронизация потоков;
- механизмы IPC (Inter Process Communication);
- поддержка механизма обработки прерываний;
- поддержка часов, таймеров и таймаутов.

Больше Neutrino не делает ничего. Вся остальная функциональность QNX обеспечивается администраторами ресурсов. Обратите внимание на то, что микроядро Neutrino работает только с потоками и ничего не знает о процессах. За процессы отвечает *администратор процессов*.

На этапе выполнения поток может находиться в одном из трех состояний: исполнение на процессоре (RUNNING), ожидание процессора, или готовность к исполнению (READY) и блокировка в ожидании освобождения некоторого ресурса (название блокированного состояния зависит от того, в ожидании какого ресурса заблокирован поток). Есть еще состояние DEAD (в UNIX его называют ZOMBIE) — когда физически поток уничтожен, но администратор процессов еще сохраняет некоторые

структуры данных с информацией о нем, чтобы передать родительскому потоку код завершения.

Диспетчеризация потоков

QNX — многозадачная ОС. Это значит, что в системе может существовать достаточно много процессов (и потоков). Причем несколько из них могут одновременно оказаться в состоянии готовности к исполнению. Как определить, какому из этих потоков предоставить свободный процессор? Для этого микроядро использует приоритет, назначенный каждому потоку. Всего в QNX Neutrino версии 6.3 имеется 256 уровней приоритета (раньше в Neutrino было 64 уровня приоритета). Самый низкий приоритет — 0, самый высокий — 255. Нулевой приоритет имеет специальный поток администратора процессов под названием idle (что в переводе с английского означает "ленивец"), на русском техническом жаргоне ее называют "холодильником". Этот поток всегда находится в состоянии готовности к исполнению READY. Не вздумайте задать своему потоку нулевой приоритет — он никогда не получит процессор!

Диспетчеризация выполняется микроядром в трех случаях:

- исполняющийся на процессоре поток перешел в блокированное состояние;
- поток с более высоким, чем у исполняющегося потока, приоритетом перешел в состояние готовности, т. е. происходит вытеснение потока (это свойство ОС называют *вытесняющей многозадачностью*);
- исполняющийся поток сам передает право исполнения на процессоре другому потоку (вызывает функцию *sched_yield()*).

Все это замечательно, но как быть в том случае, когда несколько готовых к исполнению процессов имеют равные приоритеты? Чтобы предотвратить эту (типичную, надо сказать) ситуацию, используют *дисциплины диспетчеризации*. Микроядро Neutrino позволяет задавать для каждого потока одну из следующих дисциплин диспетчеризации:

- FIFO (First In First Out — "первый вошел — первый вышел"). Если потоку назначена дисциплина диспетчеризации FIFO, то другой поток с таким же приоритетом получит управление,

только если исполняющийся поток заблокируется или сам уступит право исполнения;

- "карусельная" (Round Robin) диспетчеризация — поток исполняется в течение *кванта времени* и передает управление следующему потоку с таким же приоритетом. В QNX 6.3 квант времени (timeslice) по умолчанию равен 4 мс (для процессоров с частотой выше 40 МГц);
- спорадическая диспетчеризация — предназначена для установления лимита использования потоком процессора в течение определенного периода времени. Этот механизм заменил имевшуюся в прежних версиях QNX адаптивную диспетчеризацию и введен в порядке эксперимента — пользуйтесь им осторожно.

При спорадической дисциплине потоку задается несколько параметров:

- нормальный приоритет (N);
- нижний приоритет (L);
- начальный бюджет (C);
- период восстановления (T);
- максимальное количество пропусков восстановления.

Когда поток переходит в состояние READY, его приоритет имеет значение N в течение интервала времени C, после чего приоритет потока снижается до значения L. Но время пребывания потока в состоянии с приоритетом L ограничено. Через интервал времени T (считая вместе с C) приоритет потока снова станет равным N. Поскольку поток при приоритете L может вообще не получить управление, задается ограничение максимального количества пропусков восстановления при достижении которого потоку будет принудительно возвращен приоритет N. Таким образом, спорадическая диспетчеризация гарантирует, что поток "не съест" больше C / T процессорного времени (если не он один имеет приоритет, равный N).

Примечание

Параметры спорадической диспетчеризации в настоящее время задаются только перед запуском потока и не могут быть модифицированы при исполнении потока.

В дополнение к дисциплинам диспетчеризации, Neutrino поддерживает механизм клиент-управляемых приоритетов. Это нужно для серверного потока, обрабатывающего запросы потоков-клиентов. Идея заключается в том, что приоритет сервера устанавливается равным максимальному из приоритетов клиентов, заблокированных в ожидании освобождения сервера. Действительно, было бы не очень здорово, если бы поток с приоритетом 100 ждал, пока сервер закончит обработку предыдущего запроса с приоритетом, например, 5.

Синхронизация потоков

Для синхронизации потоков в QNX применяют несколько способов. Причем только два из них (mutex и condvar) реализованы в микроядре, остальные — надстройки над этими двумя:

- взаимоисключающая блокировка (**Mut**ual **ex**clusion lock — mutex, "мутекс") — этот механизм обеспечивает исключительный доступ потоков к разделяемым данным. Только один поток в один момент времени может владеть мутексом. Если другие потоки попытаются захватить мутекс, они становятся мутекс-заблокированными;

  Примечание

  Если в состоянии мутекс-блокировки находится поток с приоритетом выше, чем у потока-владельца блокировки, то значение эффективного приоритета этого потока автоматически повышается до уровня высокоприоритетного заблокированного потока.

- условная переменная (condition variable, или condvar) — предназначена для блокирования потока до тех пор, пока соблюдается некоторое условие. Это условие может быть сложным. Условные переменные всегда используются с мутекс-блокировкой для определения момента снятия мутекс-блокировки;
- барьер — устанавливает точку для нескольких взаимодействующих потоков, на которой они должны остановиться и дождаться "отставших" потоков. Как только все потоки из контролируемой группы достигли барьера, они разблокируются и могут продолжить исполнение;

- ждущая блокировка — упрощенная форма совместного использования условной переменной с мутексом. В отличие от прямого применения mutex + condvar имеет некоторые ограничения;
- блокировка чтения/записи (rwlock) — простая и удобная в использовании "надстройка" над условными переменными и мутексами;
- семафор — это, можно сказать, мутекс со счетчиком. Вернее, мутекс является семафором со счетчиком, равным единице. Семафор могут захватить несколько потоков (их число равно значению счетчика), все остальные потоки, пытающиеся захватить семафор, будут заблокированы. Семафоры бывают именованные и неименованные. Именованный (более медленный) вариант семафора можно использовать для синхронизации потоков, выполняющихся в разных процессах и даже на разных узлах сети.

Большинство этих механизмов работает только в пределах одного процесса, но это ограничение преодолевается путем разделения памяти.

Кроме перечисленных способов синхронизацию можно осуществлять с помощью FIFO-диспетчеризации, QNX-сообщений и атомарных операций.

Механизмы IPC

В микроядре Neutrino реализована поддержка нескольких механизмов IPC:

- синхронные сообщения QNX — наряду с архитектурой микроядра являются фундаментальным принципом QNX. Это самый быстрый способ обмена данными произвольного размера в QNX, при этом микроядру безразлично, между какими потоками происходит обмен QNX-сообщениями — внутри процесса, между разными процессами или даже между процессами, работающими на разных узлах локальной вычислительной сети;
- Pulses (по-русски этот тип сообщений называют "импульсами") — это фиксированные сообщения, имеющие размер 40 бит (8-битный код импульса и 32 бита данных), не блокирующие отправителя;
- сигналы POSIX (как простые, так и реального времени).

По просьбе заказчиков фирма QSS в порядке эксперимента ввела такое новшество, как *асинхронные сообщения*. Этот механизм, как и механизм импульсов, использует буфер для хранения сообщений. Функции асинхронных сообщений имеют такие же названия, как функции обычных сообщений, но с префиксом *asyncmsg_* (например, *asyncmsg_MsgSend()*).

Помимо этих механизмов ОСРВ QNX поддерживает несколько дополнительных способов IPC (о них мы скажем подробнее чуть позже):

- очереди сообщений POSIX (реализованы в администраторе очередей `mqueue`);
- разделяемая память (реализована в администраторе процессов);
- именованные каналы (реализованы в администраторе файловой системы QNX4);
- неименованные каналы (реализованы в администраторе каналов `pipe`).

Добавим пару слов о сигналах. *Сигналы* являются не блокирующим отправителя способом IPC. Сигналы имеют структуру, подобную импульсу (1 байт кода + 4 байта данных), и тоже могут ставиться в очередь. Для отправки процессу сигнала администратор может использовать две утилиты: стандартную UNIX-утилиту `kill` и более гибкую QNX-утилиту `slay`. По умолчанию обе эти утилиты посылают сигнал SIGINTR, при получении которого процесс обычно уничтожается. Строго говоря, процесс может определить три варианта поведения при получении сигнала:

- *игнорировать сигнал* — для этого следует задать соответствующее значение такому атрибуту потока, как сигнальная маска (поэтому обычно говорят не "игнорировать", а "маскировать" сигнал). Надо сказать, что три сигнала не могут быть проигнорированы: SIGSTOP, SIGCONT и SIGTERM;
- *использовать обработчик по умолчанию* — т. е. ничего не предпринимать. Обычно обработчиком по умолчанию для сигналов является уничтожение процесса;
- *зарегистрировать собственный обработчик* — т. е. собственную функцию, которая будет вызвана при получении сигнала.

Поток может вызвать функцию ожидания того или иного сигнала (сигналов). При этом поток станет SIGNAL-блокированным.

Кстати, об утилите `slay`. Она имеет чрезвычайно полезную опцию `-P`. Для чего — не скажу, загляните в электронную документацию, поставляемую в составе дистрибутива QNX Momentics ☺.

Спецификация POSIX определяет порядок обработки сигналов только для процессов. Поэтому при работе с многопоточными процессами в QNX необходимо учитывать следующие правила.

- Действие сигнала распространяется на весь процесс.
- Сигнальная маска распространяется только на поток.
- Неигнорируемый сигнал, предназначенный для какого-либо потока, доставляется только этому потоку.
- Неигнорируемый сигнал, предназначенный для процесса, передается первому не SIGNAL-блокированному потоку.

Сигналы и сообщения типа "импульс" позволяют реализовать в QNX событийно-управляемую модель поведения системы. Под *событиями* понимаются QNX-импульсы, прерывания, сигналы и различные типы неблокирущих сообщений. Обычно события исходят от трех следующих источников:

- вызов функции *MsgDeliverEvent()*;
- обработчик прерывания;
- сработавший таймер.

Администратор процессов QNX

До сих пор мы говорили, обсуждая различные механизмы QNX, о взаимодействии потоков, синхронизации потоков и т. п., намеренно стараясь избегать при этом слова "процесс". И делали мы это по понятной причине — микроядро Neutrino понятия не имеет ни о каких процессах. Оно знает только потоки и знает, как с ними обращаться. Но полнофункциональную POSIX-систему невозможно представить без процессов, работающих в изолированных адресных пространствах. Так вот: поддержку процессов в QNX обеспечивает *администратор процессов*. А поскольку управление процессами и памятью является очень важной и критичной функцией операционной системы, администратор

процессов скомпонован с микроядром Neutrino в единый программный модуль `procnto`.

Итак, рассмотрим функции, которые выполняет администратор процессов:

- управление процессами;
- управление механизмами защиты памяти;
- поддержка механизма разделяемой памяти и IPC на ее основе;
- управление пространством путевых имен.

Управление процессами

Вспомним, что процесс является контейнером потоков. Собственно, всю реальную работу делают именно потоки, поэтому любой процесс состоит не меньше чем из одного потока.

Процесс имеет ряд атрибутов:

- идентификатор процесса (process ID, PID);
- идентификатор родительского процесса (parent process ID, PPID);
- реальные идентификаторы владельца и группы (UID и GID);
- эффективные идентификаторы владельца и группы (EUID и EGID);
- текущий рабочий каталог;
- управляющий терминал;
- маска создания файлов (umask);
- приоритет;
- дисциплина диспетчеризации.

Жизненный цикл процесса включает четыре этапа:

- создание — процесс может быть создан только другим (родительским) процессом, при этом администратор процессов создает у себя необходимые структуры данных;
- загрузка кода и данных процесса в ОЗУ;
- выполнение потоков;
- завершение.

Завершение процесса происходит в две стадии. Первую стадию выполняет *поток завершителя* (termination thread) администратора процессов. При этом освобождаются ресурсы, связанные с процессом (страницы ОЗУ, открытые файловые дескрипторы и т. п). Поток завершителя выполняется с идентификатором уничтожаемого процесса.

Вторая стадия завершения процесса происходит внутри администратора процессов, при этом код возврата завершаемого процесса передается процессу-родителю. Здесь, как говорится, возможны варианты:

- процесс-родитель был заблокирован в ожидании кода завершения дочернего процесса. В этом случае код возврата сразу будет передан родителю, родитель разблокируется и дочерний процесс завершится;
- процесс-родитель отказался от получения кода завершения дочернего процесса, т. е. завершающийся процесс имел флаг `SPAWN_NOZOMBIE`. В этом случае дочерний процесс будет немедленно завершен;
- процесс-родитель не отказывался от получения кода возврата дочернего процесса, но и не вызвал функцию получения этого кода. В этом случае дочерний процесс блокируется до тех пор, пока родитель не прочтет код завершения, т. е. завершающийся процесс становится DEAD-блокированным или "зомби".

Управление механизмами защиты памяти

Администратор процессов QNX обеспечивает поддержку полной защиты памяти (так называемую "виртуальную память") процессов.

Каждому процессу предоставляется 4 Гбайта адресного пространства, из них код и данные процесса могут занимать пространство от 0 до 3,5 Гбайт. Диапазон адресов от 3,5 до 4 Гбайт принадлежит модулю **procnto** (эти цифры относятся к ЭВМ на базе процессора *x*86, в реализациях QNX для других аппаратных платформ эти значения могут быть другими).

Для отображения виртуальных адресов на физическую память используются аппаратные *блоки управления памятью* (MMU — Memory Management Unit).

Управление пространством путевых имен

Пространство путевых имен является одной из особенностей ОС QNX. Дело в том, что управление ресурсами ввода/вывода не встроено в микроядро, а реализуется посредством дополнительных процессов — администраторов ресурсов. Например, записью файлов на диск управляет администратор файловой системы, отправкой данных по сети — администратор сети. Как же администраторы ресурсов в ОС QNX интегрируют свои услуги? Для этого администратор процессов предоставляет механизм пространства путевых имен.

Пространство путевых имен представляет собой дерево каталогов и файлов, в вершине которого находится корневой каталог "/" (`root`). При запуске каждый администратор ресурсов регистрирует у администратора процессов свою зону ответственности, или "префикс" (можно сказать, ветку дерева). Сам модуль **`procnto`** при загрузке регистрирует в пространстве путевых имен несколько префиксов:

- `/` — корень (`root`) файловой системы, к которому монтируются все остальные префиксы;
- `/proc/` — каталог, в который отображается информация о запущенных процессах, представленных их идентификаторами (PID);
- `/proc/boot/` — каталог, в который в виде "плоской" файловой системы отображаются файлы, входящие в состав загрузочного образа QNX;
- `/dev/zero` — устройство, которое при чтении из него всегда возвращает ноль. Используется, например, для того, чтобы заполнить нолями страницы памяти;
- `/dev/mem` — устройство, представляющее всю физическую память.

Префикс, к которому администратор ресурсов запрашивает присоединение своего поддерева, называется *точкой монтирования*. Точки монтирования могут перегружаться, т. е. к одной точке могут подключать свои поддеревья несколько администраторов ресурсов. Кроме того, администратор процессов позволяет создавать

символические префиксы, т. е. регистрировать в пространстве путевых имен ссылки на существующий файл:

```
ln -Ps существующий_файл имя_ссылки
```

Механизм пространства путевых имен позволяет администратору процессов определять, какой из администраторов ресурсов должен выполнить поступивший запрос на ввод/вывод данных. Алгоритм будет такой.

1. Процесс выполняет системный вызов *open()* для того, чтобы открыть файл. При этом указывается полное (путевое) имя файла. Библиотечная функция *open()* шлет администратору процессов QNX-сообщение определенной структуры, спрашивая что-то вроде: "Какой из администраторов ресурсов отвечает за этот файл?"
2. Администратор процессов сопоставляет путевое имя файла с деревом префиксов и возвращает функции *open()* сообщение-ответ, содержащее идентификатор администратора ресурсов.
3. Функция *open()* напрямую обращается к администратору ресурсов с запросом на открытие файла.
4. Администратор ресурса создает у себя структуру данных, называемую *блоком управления открытым контекстом* (OCB — Open Control Block) и возвращает функции *open()* файловый дескриптор.

Теперь прикладной процесс может запрашивать операции чтения/записи, используя полученный файловый дескриптор. Администратор ресурсов, в свою очередь, использует OCB для того, чтобы различать запросы разных процессов. OCB включает:

- файловый дескриптор, уникальный внутри одного процесса;
- идентификатор процесса, уникальный для узла сети;
- информацию о файле (например, информацию о текущем файловом указателе).

Новый OCB создается при каждом вызове функции *open()*.

Разделяемая память и динамически присоединяемые библиотеки

Механизм разделяемой памяти реализован в администраторе процессов и предназначен для быстрого обмена большими объемами данных между процессами. Суть этого механизма заклю-

чается в том, что процесс запрашивает администратор процессов о выделении некоторого региона памяти. Затем этот регион может отображаться разными процессами на их собственное адресное пространство. Поскольку несколько процессов могут получить доступ к одному участку памяти, в целях сохранения целостности данных необходимо синхронизировать операции чтения/записи с разделяемой памятью.

Иногда программный код, выполняющий стандартные операции (яркий пример — пересылка сообщений), целесообразно выделить в отдельный программный модуль, доступный для одновременного использования несколькими процессами. Такой модуль называют *разделяемым объектом* или *динамически присоединяемой библиотекой*, а операцию "подключения" процесса к разделяемому объекту — *динамической компоновкой*. При использовании разделяемого объекта в состав приложения включается не программный код, а информация о динамической библиотеке.

Каким же образом выполняется динамическая компоновка? При создании процесса администратор процессов копирует в физическую память загружаемые сегменты программы и расшифровывает заголовок программы. Если в заголовке указано, что программе необходима динамическая библиотека, то администратор процессов загружает разделяемую библиотеку, содержащую динамический компоновщик, и передает этому динамическому компоновщику управление.

Динамический компоновщик выполняет поиск и загрузку в память необходимой динамической библиотеки. Эта библиотека добавляется во внутренний список всех загруженных библиотек, сопровождаемый динамическим компоновщиком.

Динамический компоновщик осуществляет поиск запрошенной разделяемой библиотеки в следующем порядке.

1. Сначала выполняет проверку, была ли загружена в память разделяемая библиотека, просмотрев список загруженных библиотек. Если библиотека еще не загружена, то компоновщик начинает поиск.
2. Если указано абсолютное путевое имя (т. е. начинающееся с символа "слэш" — "/"), то загружается именно указанная библиотека. Если библиотека с указанным именем не существует, то дальнейший поиск не производится.

3. Если указано не абсолютное путевое имя, то динамический компоновщик осуществляет поиск библиотеки в каталогах, перечисленных в переменной окружения `LD_LIBRARY_PATH`, однако только в том случае, если программа не имеет маркировки `setuid`.
4. Если разделяемая библиотека по-прежнему не найдена, и если динамическая секция исполняемого кода содержит тег `DT_RPATH`, то поиск производится и по пути, определенному в теге `DT_RPATH`.
5. Если разделяемая библиотека по-прежнему не найдена то компоновщик выполняет поиск в каталогах, определенных в переменной окружения `LD_LIBRARY_PATH` модуля **`procnto`**. Если эта переменная не была определена, то используется значение по умолчанию — каталог, содержащий загрузочный образ ОС.

Кроме того, процесс может загружать разделяемую библиотеку во время исполнения, используя функцию *dlopen()*, и, когда больше не нуждается в ней, выгружать ее с помощью функции *dlclose()*. Поиск, загрузку и выгрузку динамической библиотеки и в этом случае выполняет динамический компоновщик (он содержится в разделяемом объекте **`libqnx.so`**).

Получение информации о процессах

События создания и уничтожения процессов и потоков фиксируются с помощью пакета трассировки SAT (System Analysis Toolkit). Однако часто бывает нужно быстро получить информацию о текущем состоянии процессов и потоков. Для этого в состав QNX входит несколько утилит:

- **`ps`** — основная POSIX-утилита для мониторинга процессов. Она включена в QNX как для совместимости POSIX, так и для удобства администраторов, недавно работающих в QNX;
- **`sin`** — весьма информативная QNX-утилита мониторинга процессов. С помощью **`sin`** можно, задав соответствующую опцию, получить информацию о процессах на другом узле сети Qnet. По умолчанию **`sin`** выдает для каждого процесса: PID, размер кода, размер стека и использование процессора.

С помощью аргументов-команд можно получить дополнительную информацию:

- **args** — показать аргументы процессов;
- **cpu** — показать использование ЦПУ;
- **env** — показать переменные окружения процессов;
- **fds** — показать открытые файловые дескрипторы;
- **flags** — показать флаги процессов;
- **info** — показать общую информацию о системе;
- **memory** — показать память, используемую процессами;
- **net** — показать информацию об узлах сети;
- **registers** — показать состояние регистров;
- **signals** — показать сигнальные маски;
- **threads** — показать информацию по потокам;
- **timers** — показать таймеры, установленные процессами;
- **users** — показать реальные и эффективные идентификаторы владельцев и групп процессов.

Для выполнения команд достаточно ввести первые два символа, например команда **sin flags** равнозначна команде **sin fl**. У утилиты **sin** есть вариант с графическим интерфейсом — утилита **vsin** (рис. 5.1).

pidin — эта утилита появилась в QNX только с 6-й версии и предназначена для получения детальной информации о потоках.

Кроме этих утилит, эффективное средство мониторинга процессов есть в составе QNX IDE — перспектива QNX System Information. Чтобы воспользоваться этим средством, необходимо запустить целевой агент **qconn** на узле, подлежащем мониторингу. Если необходимо выполнить мониторинг локального узла, значит, **qconn** нужно запустить на локальной ЭВМ. Окно перспективы QNX System Information показано на рис. 5.2.

В левой части отображается список текущих процессов, а содержимое правой части зависит от выбранной вкладки (на рис. 5.2 показана вкладка **System Summary**).

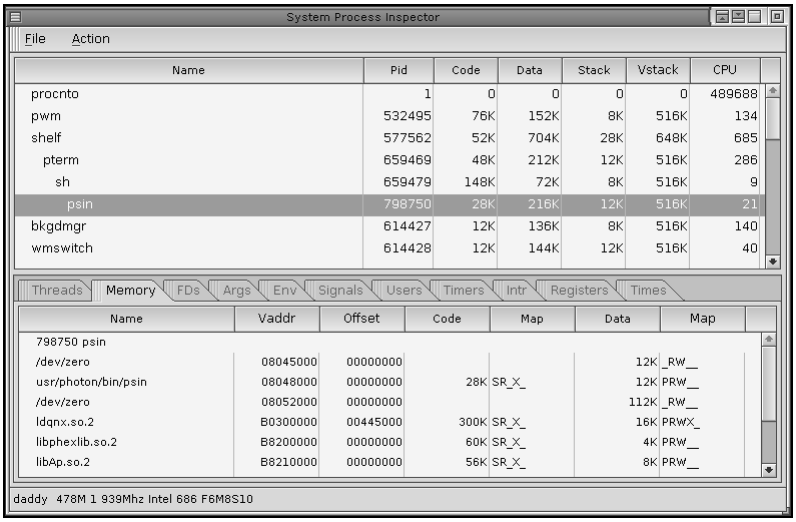


Рис. 5.1. Окно утилиты **vsin**

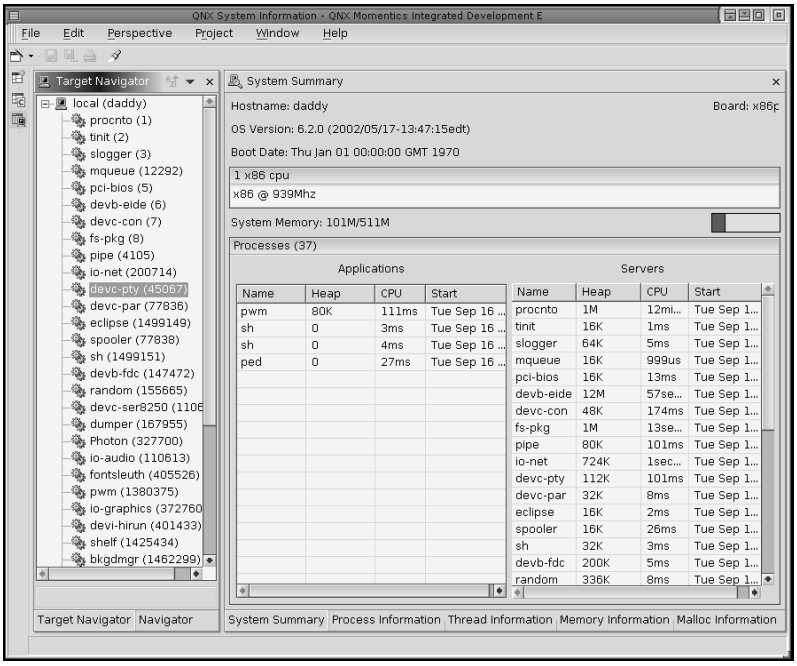


Рис. 5.2. Окно перспективы QNX System Information, вкладка **System Summary**

"Посмертная" диагностика процессов

В основе "посмертной" диагностики процессов лежит использование программы **dumper**, сохраняющей на диске образ "аварийного" процесса — так называемый *core-файл*. Эту программу можно использовать и для немедленного получения образа какого-либо работающего процесса. Если **dumper** запущен системой, то по умолчанию он сохраняет core-образы в каталоге `/var/dumps`. Образы сохраняются с именами в формате *имя_процесса*`.core`.

Полученный образ процесса можно анализировать двумя способами: либо утилитой **coreinfo**, сразу выдающей информацию о core-образе, либо с помощью GNU-отладчика **gdb**:

```
gdb имя_программы имя_программы.core
```

Многие поставщики программного обеспечения требуют, чтобы при обращении в их службы технической поддержки клиенты предоставляли соответствующие core-образы.

Дополнительные способы IPC

Как уже упоминалось, вне микроядра Neutrino с помощью дополнительных администраторов ресурсов реализовано несколько способов МЗВ:

- очереди сообщений POSIX (реализованы в администраторе очередей **mqueue**);
- именованные каналы (реализованы в администраторе файловой системы POSIX);
- неименованные каналы (реализованы в администраторе каналов **pipe**).

Очереди сообщений POSIX

Очереди сообщений POSIX реализованы в QNX с помощью администратора очередей **mqueue**. Администратор **mqueue** регистрирует в пространстве путевых имен префикс `/dev/mqueue`, имеющий тип "каталог". Очереди сообщений POSIX — это именованные объекты, поэтому данный механизм можно использовать для обмена данными между процессами как в рамках одной ЭВМ, так и между процессами, работающими на разных узлах сети.

Сообщения POSIX не копируются непосредственно из адресного пространства одного процесса в адресное пространство другого, как это делают сообщения QNX, а используют промежуточное адресное пространство администратора **mqueue**. Поэтому сообщения POSIX работают относительно медленно.

Очередь сообщений можно посмотреть командой:

```
ls -lR /dev/mqueue
```

Размеры файлов означают количество сообщений в очередях.

Именованные и неименованные каналы

Каналы, или *программные каналы*, — это один из традиционных способов МЗВ в UNIX. Назначение канала — обеспечить однонаправленную передачу данных от одного процесса к другому. При этом вывод одной программы соединяется с вводом другой. В системном администрировании неименованные каналы используются для создания так называемых *конвейеров* — цепочек последовательно выполняющихся команд, когда результат одной команды является входными данными для другой, например:

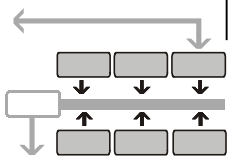
```
cat /etc/services | grep telnet
```

Здесь показано, что канал создается с помощью символа "|". В результате выполнения этого конвейера на экран будут выведены только те строки файла `/etc/services`, которые содержат слово "telnet".

Различия между именованными и неименованными каналами:

- именованные каналы представляют собой особый тип файла, хранящегося в файловой системе (т. е. один процесс пишет данные в этот файл, а другой — читает), поэтому именованные каналы работают медленнее, но могут использоваться для МЗВ между любыми процессами в сети;
- неименованные каналы реализованы с помощью администратора каналов **pipe**, который и выполняет буферизацию данных;
- неименованные каналы могут использоваться для МЗВ только между процессами, связанными отношениями "родительский" — "дочерний".

Несмотря на то, что неименованный канал работает быстрее именованного, он уступает по скорости QNX-сообщениям.



Глава 6

Разграничение доступа к данным в ОС QNX

По сути дела, данные могут находиться в двух "местах" — храниться в виде файлов на носителе (обычно — на жестком диске) или размещаться в ОЗУ, будучи частью какого-либо процесса. С точки зрения ЭВМ, данными оперируют не пользователи, а выполняющиеся процессы. Таким образом, задача разграничения доступа сводится к двум подзадачам: обеспечить механизм, позволяющий пользователям запускать только определенные программы, и обеспечить механизм, позволяющий процессам работать только с определенными файлами. Оба этих механизма реализованы с помощью файловых атрибутов доступа *(см. гл. 4)*. В *гл. 5* мы говорили, что каждому процессу при создании присваивается четыре идентификатора: реальный и эффективный идентификаторы пользователя (UID и EUID) и реальный и эффективный идентификаторы группы (GID и EGID). Как же происходит привязка этих идентификаторов к конкретному человеку? Для этого используется механизм регистрации пользователей. В этой главе будут рассмотрены вопросы:

- регистрация пользователя;
- добавление и удаление пользователей и их групп.

Регистрация пользователя

Вспомним, что пользователь регистрируется в системе посредством утилиты `login` или `phlogin`. Получив имя и пароль, утилита (пускай это будет `login`), просматривает базу данных пользователей

`/etc/passwd` в поисках указанного имени. Файл `/etc/passwd` состоит из строк следующего формата:

`username:haspw:userid:group:comment:homedir:shell`

`username` — имя пользователя, используемое для входа в систему; `haspw` — если поле не пустое, то в файле `/etc/shadow` хранится пароль пользователя; `userid` — идентификатор пользователя (для `root` равен 0);

`group` — идентификатор группы;

`comment` — любая строка, не содержащая символ `":"`;

`homedir` — домашний каталог пользователя, т. е. каталог, в котором пользователь может произвольно создавать и удалять файлы;

`shell` — командный интерпретатор, который вызывает утилита **`login`** при успешном входе пользователя в систему.

Если введенное имя пользователя существует в базе данных, то проверяется значение поля `haspw`. Поле может быть пустым, что означает отсутствие пароля у пользователя, или не пустым. Если поле не пустое, то **`login`** просматривает базу данных паролей `/etc/shadow`, разыскивая строку, соответствующую имени пользователя. Файл `/etc/shadow` состоит из строк следующего формата:

`username:password:lastch:minch:maxch:warn:inact:expire:reserved`

`username` — имя пользователя;

`password` — зашифрованный пароль пользователя;

`lastch` — время последней модификации;

`minch` — минимальное количество дней для модификации;

`maxch` — максимальное количество дней для модификации;

`warn` — количество дней для предупреждения;

`inact` — максимальное количество дней между входами в систему;

`expire` — дата истечения доступа;

`reserved` — поле зарезервировано для последующего использования.

Если имя и пароль введены правильно, то **`login`** запускает командный интерпретатор, указанный в файле `/etc/passwd`, с идентификаторами UID и EUID, равными идентификатору

пользователя, указанному в файле `/etc/passwd`, и идентификаторами GID и EGID, равными идентификатору группы, указанному в файле `/etc/passwd`.

Все остальные процессы, запускаемые из этого интерпретатора, включая Photon, наследуют эти идентификаторы.

Следует добавить, что существует файл `/etc/group`, строки которого имеют формат:

```
groupname:reserved:group:member
```

`groupname` — имя группы;

`reserved` — зарезервировано для дальнейшего использования;

`group` — числовой идентификатор группы;

`member` — список имен пользователей, принадлежащих к данной группе.

В список имен пользователей, принадлежащих к группе, можно добавить любого существующего пользователя.

Надо заметить, что утилита **`login`** (но не **`phlogin`**!) создает в домашнем каталоге пользователя файл `.lastlogin`, содержащий дату и время последнего входа в систему.

Добавление и удаление пользователей и их групп

В этом разделе описаны:

- ❒ смена пароля и добавление пользователей и групп;
- ❒ удаление пользователей и групп;
- ❒ изменение атрибутов процесса.

Смена пароля и добавление пользователей и групп

Пользователь может изменить свой пароль, выполнив команду **`passwd`**. Утилита запросит прежний пароль и дважды попросит ввести новый пароль.

Если команду **passwd** выполнит пользователь root, при этом указав в качестве аргумента команды имя пользователя, то возможно два варианта:

- если пользователь с таким именем уже существует, то пользователь root получит предложение изменить его пароль;
- если пользователя с таким именем нет, то утилита предложит системному администратору ввести регистрационные данные пользователя для заполнения файла /etc/passwd. Если при вводе данных администратор задаст идентификатор группы, которой нет в файле /etc/group, то утилита выведет сообщение, чтобы пользователь root не забыл отредактировать файл /etc/group.

Добавление новых групп выполняется путем простого редактирования файла /etc/group.

При добавлении пользователей и изменении паролей старые версии файлов /etc/passwd и /etc/shadow сохраняются, соответственно, в файлах /etc/opasswd и /etc/oshadow.

Кстати, утилита **passwd** представляет из себя замечательный образец использования файлового атрибута Set User ID. Дело в том, что процесс **passwd** редактирует файл, записывать в который имеет право только пользователь root. То есть процесс **passwd** должен всегда иметь UID и EUID, равные 0. Это и достигается использованием атрибута SUID. Разумеется, предоставлять файлам атрибуты SUID и SGID нужно очень аккуратно, чтобы не сделать брешь в защите информации.

Удаление пользователей и групп

Для того чтобы удалить пользователя из системы, системный администратор должен отредактировать три файла, удалив из них строки, соответствующие пользователю: /etc/passwd, /etc/shadow, /etc/group.

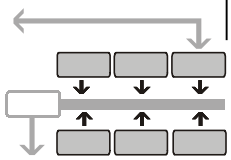
Для удаления группы достаточно отредактировать файл /etc/group, но обязательно убедитесь, что ни для кого из существующих пользователей эта группа не является первичной (короче говоря, проверьте, чтобы в файле /etc/passwd эта группа не упоминалась).

Изменение атрибутов процесса

Для временного изменения UID и EUID предназначена утилита **su**, а для временного изменения GID и EGID — утилита **newgrp**. Для того чтобы можно было изменить идентификатор группы, необходимо, чтобы имя пользователя было в списке членов этой группы в файле `/etc/group`.

В природе еще встречается свободно распространяемый пакет с утилитой **sudo** — она представляет собой **su** с расширенной функциональностью.

Для того чтобы регистрировать использование утилиты **su** пользователями, необходимо создать файл журнала с именем, указанным в конфигурационном файле `/etc/default/su`. По умолчанию там определен файл `/usr/adm/sulog`. Это наследство QNX4, где не было каталога `/var` для хранения текущей изменяемой информации, как это принято в UNIX, а использовался каталог `/usr`. Поэтому лично я предпочитаю указать утилите **su** вести журнал в файле `/var/log/sulog`. Не забудьте, что журнальный файл должен иметь атрибуты доступа по чтению и записи только для своего владельца — пользователя `root`.



Глава 7

Управление ресурсами ЭВМ в ОС QNX

В этой главе рассматриваются вопросы:

- администраторы ресурсов;
- файловые системы в QNX;
- символьные устройства ввода/вывода.

Управление ресурсами ЭВМ — одна из главных функций любой операционной системы. К основным ресурсам, которыми управляет ОС QNX, относятся:

- файловые системы;
- символьные устройства ввода/вывода (последовательные и параллельные порты, сетевые карты и т. д.);
- виртуальные устройства ("нуль"-устройство, псевдотерминалы, генератор случайных чисел и т. п.).

В QNX поддержка ресурсов не встроена в микроядро и организована с помощью специальных программ и динамически присоединяемых библиотек, получивших название "администраторы ресурсов". Взаимодействие между администраторами ресурсов и другими программами реализовано через четко определенный, хорошо документированный интерфейс файлового ввода/вывода. Хотя, конечно, на самом деле весь обмен построен на базовом механизме QNX-сообщений микроядра Neutrino.

Администраторы ресурсов

Администратор ресурсов — это прикладная серверная программа, принимающая QNX-сообщения от других программ и, при

необходимости, взаимодействующая с аппаратурой. Для связи между программой-клиентом и администратором ресурсов используется механизм пространства имен. Администратор ресурсов работает так.

1. Выполняется инициализация интерфейса сообщений, при этом создается канал, по которому клиенты могут посылать свои сообщения администратору ресурсов.
2. Регистрируется путевое имя (т. е. зона ответственности) в пространстве имен администратора процессов.
3. Запускается бесконечный цикл по приему сообщений от клиентов.
4. С помощью операторов `switch/case` выполняется переключение на нужный обработчик для каждого типа сообщения.

По такой схеме работают все администраторы ресурсов.

Файловые системы в QNX

В этом разделе рассматриваются:

- классификация файловых систем в QNX;
- реализация поддержки файловых систем.

Классификация файловых систем в QNX

ОС QNX обеспечивает поддержку различных файловых систем с помощью соответствующих администраторов ресурсов, каждый из которых при запуске регистрирует у администратора процессов зону ответственности в виде путевого имени. Такая реализация позволяет запускать и останавливать любую комбинацию файловых систем динамически.

Файловые системы, поддерживаемые в QNX, можно классифицировать так.

- Образная файловая система (image filesystem) — простая файловая система "только для чтения", состоящая из модуля **`procnto`** и других файлов, включенных в загрузочный образ QNX. Этот тип файловой системы поддерживается непосредственно администратором процессов и достаточен для многих

встроенных систем. Если же требуется обеспечить поддержку других файловых систем, то модули их поддержки добавляются в образ и могут запускаться по мере необходимости.

- RAM — плоская "файловая система", которую автоматически поддерживает администратор процессов. Файловая система RAM основана на использовании ОЗУ и позволяет выполнять операции чтения/записи из каталога `/dev/shmem`. Этот тип файловой системы нашел применение в очень маленьких встроенных системах, где не требуется хранить данные на энергонезависимом носителе и достаточно ограниченных функциональных возможностей (нет поддержки каталогов, жестких и мягких ссылок).
- Блочные файловые системы — традиционные файловые системы, обеспечивающие поддержку блок-ориентированных устройств типа жестких дисков и дисководов CD-ROM. К ним относятся файловые системы QNX4, DOS, Ext2 и CD-ROM.
  - Файловая система QNX4 (**`fs-qnx4.so`**) — высокопроизводительная файловая система, сохранившая формат и структуру дисков ОС QNX4, однако усовершенствованная для повышения надежности, производительности и совместимости со стандартом POSIX.
  - Файловая система DOS (**`fs-dos.so`**) обеспечивает прозрачный доступ к локальным разделам FAT (12, 16, 32), при этом файловая система конвертирует POSIX-примитивы работы с диском в соответствующие DOS-команды. Если эквивалентную операцию выполнить нельзя (например, создать символьную ссылку), то возвращается ошибка.
  - Файловая система CD-ROM (**`fs-cd.so`**) обеспечивает прозрачный доступ к файлам на компакт-дисках формата ISO 9660 и его расширений (Rock Ridge, Joliet, Kodak Photo CD и аудио).
  - Файловая система Ext2 (**`fs-ext2.so`**) обеспечивает прозрачный доступ из среды QNX к Linux-разделам жесткого диска версии как 0, так и 1.
- Flash — не блок-ориентированные файловые системы, разрабатываемые специально для устройств флэш-памяти.

- Network — файловые системы, обеспечивающие доступ к файловым системам на других ЭВМ. К ним относятся файловые системы NFS и CIFS (SMB).
  - Файловая система NFS (Network File System) обеспечивает клиентской рабочей станции прозрачный доступ через сеть к файлам независимо от операционных систем, используемых файл-серверами. NFS использует механизм удаленного вызова процедур (RPC, Remote Procedure Call) и работает поверх TCP/IP.
  - Файловая система CIFS (Common Internet File System) обеспечивает клиентским станциям прозрачный доступ к сетям Windows, а также к UNIX-системам с запущенным сервером SMB (Server Message Block)[1]. Работает поверх TCP/IP.
- Virtual — особые файловые системы, обеспечивающие специфические функциональные возможности при работе с другими файловыми системами.
  - Пакетная файловая система — обеспечивает привычное для пользователя представление файлов и каталогов, хранящихся в каталогах, именуемых пакетами. Реализована с помощью администратора ресурсов fs-pkg. Эта файловая система будет подробно рассмотрена в специальном разделе данной главы.
  - Inflator — обеспечивает динамическое разжатие при открытии файлов, сжатых утилитой `deflate` и содержащихся в заданном каталоге. Реализована с помощью администратора ресурсов `inflator`.

Реализация поддержки файловых систем

Поскольку у некоторых файловых систем, работающих в ОС QNX, много общих черт, то для максимизации повторного использования программного кода файловые системы проектируют как комбинацию драйверов и разделяемых библиотек. Такое решение позволяет существенно сократить количество дополнительной

[1] Поддержка протокола SMB реализована в пакете Samba.

памяти, требуемой при добавлении файловой системы в QNX, поскольку добавляется только код, непосредственно реализующий протокол обмена с данной файловой системой.

Например, если администратор конфигурирования аппаратуры **enum-devices** обнаружил интерфейс EIDE, то запускается драйвер **devb-eide**. Этому драйверу для работы необходимо загрузить модуль поддержки блок-ориентированного ввода/вывода **io-blk.so**, который создает в каталоге `/dev` несколько блок-ориентированных файлов устройств. По умолчанию они обозначаются `hd`*n* (для жесткого диска) и `cd`*n* (для CD-ROM), где *n* соответствует физическому номеру устройства. Кроме того, для каждого раздела жестких дисков создается свой блок-ориентированный специальный файл с именем `hd`*n*`t`*m*, где *n* — номер устройства, а *m* — тип раздела. Например, для раздела FAT32 на первом жестком диске будет создан файл `/dev/hd0t11`. Если разделов одного типа несколько, то они нумеруются дополнительно с разделением номеров точкой, например, `/dev/hd0t11.1`.

Модуль **io-blk.so** обеспечивает для всех блочных файловых систем буферный кэш, в который помещаются данные при выполнении записи на диск. Это позволяет значительно сократить число операций чтения/записи с физическим диском, т. е. повышает производительность работы файловых систем. Однако критичные с точки зрения надежности функционирования блоки файловой системы (например, информация о структуре диска) записываются на диск немедленно и синхронно, минуя обычный механизм записи.

Для доступа к жестким дискам, компакт-дискам и оптическим дискам драйверу необходимо подгрузить соответствующие модули поддержки общих методов доступа, соответственно, **cam-disk.so**, **cam-cdrom.so** и/или **cam-optical.so**.

Для поддержки собственно блочных файловых систем модуль **io-blk.so** загружает необходимые администраторы файловых систем, также реализованные в виде динамически присоединяемых библиотек. Поддержка блочных файловых систем реализована в модулях администраторов ресурсов:

- **fs-qnx4.so** — файловая система QNX4;
- **fs-ext2.so** — файловая система Ext2;

- **fs-dos.so** — файловая система FAT32;
- **fs-cd.so** — файловая система ISO9660.

Администраторы файловых систем монтируют свои разделы в определенные точки файловой системы. Раздел QNX4, выбранный в процессе загрузки как первичный, монтируется в корень файловой системы — /. Остальные разделы по умолчанию монтируются в каталог /fs. Например, компакт-диск монтируется в точку /fs/cd, а файловая система FAT32 — в точку /fs/hd0-dos.

Для того чтобы программа **diskboot** могла использовать раздел для поиска базового образа файловой системы QNX, необходимо наличие файла /.diskroot *(см. гл. 12, разд. "Процесс начальной загрузки").*

Операция по изменению информации на диске выполняется в форме транзакции, поэтому даже при катастрофических сбоях (например, при отключении питания) файловая система QNX4 остается цельной. В худшем случае некоторые блоки могут быть выделены, но не использованы, вернуть эти блоки можно, запустив утилиту **chkfsys**.

Пакетная файловая система

Ну вот, наконец-то мы можем сказать правду — почти все, что показывает файловый менеджер или команда **ls**, на самом деле неправда. Нет на диске таких каталогов: /usr, /bin, /etc и многих других. Нет и никогда не было. Дело в том, что все средства просмотра файловой системы показывают дерево префиксов администратора процессов. А значительная часть всех префиксов зарегистрирована администратором пакетной файловой системы **fs-pkg**.

Инсталляционные пакеты и их репозитарии

Создавая ОС QNX Neutrino, разработчики думали и о том, каким образом обеспечить простой и эффективный механизм поставки и инсталляции программного обеспечения для QNX. В настоящее время очень распространенным средством доставки ПО от разработчика к пользователю стал Интернет. Поэтому

было принято решение о том, что файлы, входящие в состав программного продукта для QNX, следует помещать в архив формата TGZ, т. е. все файлы компонуются в один утилитой **tar**, и этот файл сжимается утилитой **gzip**. Такой архив и есть QNX-пакет — файл с расширением `qpk`.

Чтобы избавить пользователя от необходимости скачивать весь пакет, если требуется узнать информацию о производителе, версии, совместимости с другим программным обеспечением и т. п., такого рода информация помещается в отдельный от пакета файл манифеста с расширением `qpm`. Одну или несколько пар файлов `.qpk` и `.qpm` можно с помощью тех же утилит **tar** и **gzip** объединить в репозитарий `.qpr`. В этом случае информация о содержимом репозитария помещается в отдельный файл-манифест `.qrm`. Репозитарий с его манифестом можно размещать на любом носителе — CD-ROM, веб-сайте и т. п. Для установки пакетов в операционной системе QNX используется уже знакомый вам инсталлятор QNX Software Installer — программа **qnxinstall**.

Рассмотренный нами тип репозитария — это репозитарий для хранения готовых к инсталляции пакетов программного обеспечения. В составе ОС QNX есть необходимые инструменты для построения собственных инсталляционных пакетов.

Репозитарии инсталлированного программного обеспечения

В самой ОС QNX есть два репозитария:

- `/pkgs/base` — для базовой системы QNX;
- `/pkgs/repository` — для дополнительного программного обеспечения.

Эти каталоги и то, что в них хранится, — реальные объекты файловой системы на диске, все честно (или почти честно — каталог `/pkgs/base` является "отображением" содержимого файла `/boot/fs/qnxbase.qfs`).

Содержимое репозитариев имеет определенную структуру, позволяющую избежать конфликтов между программным обеспечением разных производителей, версиями, целевыми и хост-

платформами. Путевое имя каталога, содержащего тот или иной пакет, имеет стандартизованный формат:

`/pkgs/repository/`*`производитель/продукт/каталог_пакета`*

Внутри каждого пакета обязательно есть XML-файл `MANIFEST`, который полностью описывает пакет. Платформо-независимые файлы (например, текстовые файлы) размещаются в каталогах, воспроизводящих дерево файловой системы (корнем считается каталог пакета). Платформо-зависимые файлы хранятся аналогично, но файлы, относящиеся к одной аппаратной архитектуре, располагаются в отдельном каталоге (`ppcbe`, `x86`, `shle` и т. д.), уже внутри которого воспроизводится обычное дерево каталогов (рис. 7.1).

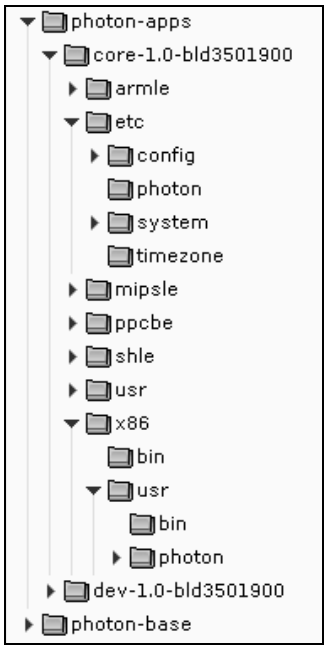


Рис. 7.1. Дерево каталогов

Такой подход, с одной стороны, позволяет экономить дисковое пространство — платформо-независимый код не дублируется. С другой стороны, на одной ЭВМ можно без конфликтов хранить файловые системы для разных аппаратных платформ, что

при сетевой прозрачности QNX обеспечивает широкие возможности для машин с ограниченными ресурсами.

Администратор пакетной файловой системы **fs-pkg** отображает содержимое файловой системы в таком виде, как мы привыкли это видеть, скрывая пакетную структуру данных. Некоторые каталоги существуют реально — `/var`, `/tmp` и др. — они создаются при первом старте ОС после инсталляции *(см. гл. 12, разд. "Процесс начальной загрузки")*. Информация о конфигурации пакетов хранится в файле `/etc/system/package/packages`.

Идеология пакетов предполагает их неизменность, т. е. пакеты должны быть доступны только для чтения. А что делать, если требуется модифицировать или вообще заменить какой-либо файл? Возможность таких изменений реализована с помощью каталога `/var/pkg/spill`. Именно туда помещаются измененные и добавленные файлы, а также записывается информация об "удалении" файлов.

Деинсталляция и деинициализация пакетов выполняется программой QNX Software Installer.

Символьные устройства ввода/вывода

Символьными устройствами ввода/вывода называют такие устройства, которые передают или принимают последовательность байтов один за другим, в отличие от блок-ориентированных устройств. Имена администраторов символьных устройств ввода/вывода имеют вид **devc-***. Обычно в системе имеются следующие символьные устройства:

- консольные устройства (или текстовые консоли);
- последовательные устройства;
- параллельные устройства;
- псевдотерминалы (ptys).

Для повторного использования кода управления символьными устройствами соответствующие администраторы ресурсов компонуются со статической библиотекой `io-char`. Эта библиотека управляет потоками данных между приложением и драйвером устройства посредством очередей разделяемой памяти. Каждая очередь работает по принципу FIFO.

Режимы ввода при работе устройств:

- поточный или "сырого ввода" (raw) — наиболее производительный режим, однако `io-char` не выполняет никакое редактирование принимаемых данных;
- редактируемый (edited) — режим, при котором `io-char` может выполнять операции редактирования над каждым символом строки. После окончания редактирования строки она становится доступной для обработки прикладным процессом (обычно, после ввода символа возврата каретки `CR`). Такой режим часто именуют *каноническим*.

Консольные устройства

Системные консоли управляются процессами-драйверами **devc-con** или **devc-tcon**. Совокупность клавиатуры и видеокарты с монитором называют физической консолью.

Консольный драйвер **devc-con** позволяет запускать на физической консоли несколько терминальных сеансов посредством виртуальных консолей. При этом **devc-con** организует несколько очередей ввода/вывода к `io-char` через несколько символьных устройств с именами `/dev/con1`, `/dev/con2` и т. д. С точки зрения приложения создается эффект наличия нескольких консолей.

Консольный драйвер **devc-tcon** представляет собой "облегченную" (т. е. поддерживающую только одиночную консоль с "сырым" вводом) версию драйвера **devc-con** для систем с ограниченным объемом памяти.

Последовательные устройства

Последовательные устройства ввода/вывода управляются семейством процессов-драйверов **devc-ser***. Каждый из драйверов может управлять более чем одним физическим устройством и обеспечивать поддержку нескольких символьных устройств.

Параллельные устройства

Параллельные устройства (обычно это принтерные порты) управляются драйвером-процессом **devc-par**. Этот драйвер под-

держивает только вывод, чтение из устройства `/dev/parn` дает результат, аналогичный чтению из `/dev/null`.

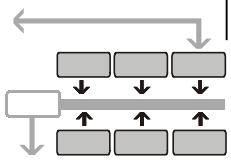
Псевдотерминалы (ptys)

Псевдотерминалы управляются драйверным процессом **`devc-pty`**. С помощью аргумента при запуске драйвера определяется количество псевдотерминалов.

Псевдотерминал состоит из двух частей: основной (master) драйвер и подчиненное (slave) устройство. Подчиненное устройство обеспечивает для прикладных процессов интерфейс, идентичный обычному POSIX-терминалу. Обычный терминал взаимодействует с аппаратным устройством, а подчиненное устройство псевдотерминала вместо этого взаимодействует с основным драйвером псевдотерминала. Основной драйвер может взаимодействовать с другим процессом. Таким образом, псевдотерминал может использоваться для того, чтобы процесс мог взаимодействовать с другим процессом как с символьным устройством.

Псевдотерминалы, как правило, используются для создания интерфейса для таких программ, как эмуляторы терминала **`pterm`** или **`telnet`**.

Глава 8



Сетевая подсистема QNX

В этой главе будут рассмотрены вопросы:

- структура сетевой подсистемы QNX;
- "родная" QNX-сеть — Qnet;
- поддержка TCP/IP в QNX.

ОС QNX — система изначально сетевая, однако сетевые механизмы, как и все остальное, реализованы в виде дополнительных администраторов ресурсов. Хотя некоторая поддержка сети есть в микроядре — способ адресации QNX-сообщений обеспечивает возможность передачи их по сети. Грубо говоря, микроядру Neutrino все равно, является сообщение сетевым или локальным, что дает QNX мощный механизм поддержки сетевых кластеров.

В QNX реализованы средства поддержки двух протоколов — TCP/IP, являющегося промышленным стандартом, и Qnet, "родного" протокола QNX, реализующего концепцию "прозрачной сети". Компьютеры, объединенные в сеть Qnet, фактически представляют собой виртуальную многопроцессорную суперЭВМ.

Структура сетевой подсистемы QNX

В центре реализации сетевой подсистемы QNX Neutrino находится администратор сетевого ввода/вывода **io-net**. Процесс **io-net** при старте регистрирует префикс-каталог `/dev/io-net` и загружает необходимые администраторы сетевых протоколов и аппаратные драйверы. Протоколы, драйверы и другие необходимые компоненты (все они реализованы в виде DLL) загружаются либо в соответствии с аргументами командной строки, заданными

при запуске **io-net**, либо в любое время командой монтирования **mount**.

Следующая команда запускает поддержку сети с драйвером сетевого адаптера NE2000 и с поддержкой протоколов TCP/IP и Qnet:

```
io-net -dne2000 -ptcpip -pqnet
```

Или, например, можно выполнить следующие команды.

1. Запустить администратор сети:
   ```
   io-net &
   ```
2. Загрузить драйвер Ethernet для адаптера NE2000:
   ```
   mount -T io-net devn-ne2000.so
   ```
3. Загрузить администратор протокола Qnet:
   ```
   mount -T io-net npm-qnet.so
   ```
4. Загрузить администратор протокола TCP/IP:
   ```
   mount -T io-net npm-tcpip.so
   ```

Отключить поддержку, например, Qnet и выгрузить DLL **npm-qnet.so** можно командой:

```
umount /dev/io-net/qnet_en
```

Для получения диагностической и статистической информации о работе сетевой карты предназначена утилита **nicinfo** (**N**etwork **I**nterface **C**ard **INFO**rmation). По умолчанию утилита **nicinfo** обращается к устройству `/dev/io-net/en0`. Для адаптеров Ethernet утилита **nicinfo** выдает наименование модели контроллера (у меня — RealTek 8139) и информацию, представленную в табл. 8.1.

Таблица 8.1. *Информация, выдаваемая утилитой* **nicinfo**

Наименование параметра	Пример вывода	Значение
Информация об адаптере		
Physical Node ID	00005C 000218	Физический адрес (MAC-адрес) адаптера
Current Physical Node ID	00005C 000218	MAC-адрес, присвоенный адаптеру (имеет смысл для адаптеров с программируемыми MAC-адресами)

Таблица 8.1 *(продолжение)*

Наименование параметра	Пример вывода	Значение
Информация об адаптере		
Media Rate	100.00 Mb/s full-duplex UTP	Скорость и способ передачи данных
Mtu	1514	Максимальный размер кадра, байты
Lan	0	Логический номер сети (используется, если к ЭВМ подключено несколько сетевых адаптеров)
I/O Port Range	0xC000 -> 0xC0FF	Диапазон портов ввода-вывода
Hardware Interrupt	0x5	Номер прерывания
Promiscuous	Disabled	Включен ли режим приема всех пакетов
Multicast	Enabled	Включен ли режим групповой передачи
Информация по пакетам		
Total Packets Txd OK	1136	Общее число удачно отправленных кадров
Total Packets Txd Bad	0	Общее число кадров, отправленных с ошибками
Total Packets Rxd OK	1934	Общее число удачно принятых кадров
Total Rx Errors	0	Общее число принятых пакетов, содержащих ошибки
Количество байтов в пакетах		
Total Bytes Txd	163 907	Сколько всего байтов отправлено
Total Bytes Rxd	846 826	Сколько всего байтов получено

Таблица 8.1 *(окончание)*

Наименование параметра	Пример вывода	Значение
Информация об ошибках		
Tx Collision Errors	0	Количество коллизий при передаче
Tx Collisions Errors (aborted)	0	Количество коллизий при передаче (с отменой передачи пакета)
Carrier Sense Lost on Tx	0	Количество "потерь несущей" при передаче
FIFO Underruns During Tx	0	Число попыток отправить данные из пустого буфера
Tx defered	0	Количество пакетов, передача которых была отложена
Out of Window Collisions	0	Число поздних коллизий
FIFO Overruns During Rx	0	Количество переполнений приемного буфера
Alignment errors	0	Число ошибок выравнивания
CRC errors	0	Количество ошибок несовпадения контрольной суммы

Для разработки собственных драйверов сетевых карт в состав QNX Momentics входит специальный программный пакет Network Driver Development Kit (Network DDK), включающий исходные тексты нескольких драйверов и детальную инструкцию по написанию аппаратно-зависимого кода.

"Родная" QNX-сеть — Qnet

Как уже было сказано, операционная система QNX обладает мощным средством сетевого взаимодействия, позволяющим

превратить сеть QNX-машин фактически в единый распределенный компьютер. Прозрачность сетевого взаимодействия в сети Qnet основана на способности QNX выполнять передачу сообщений между микроядрами Neutrino через сеть. Каким же образом можно отличить сетевое сообщение от локального и, главное, как идентифицировать узлы сети? Для этой цели служит пространство путевых имен администратора процессов. При загрузке и инициализации администратор **npm-qnet.so** регистрирует символьное устройство `/dev/io-net/qnet_en` и префикс-каталог `/net`, в котором размещаются папки с именами узлов сети. То есть, например, если в сети есть узлы с именами **alpha** и **beta**, каталог `/etc` узла **beta** является каталогом `/net/beta/etc` узла **alpha**.

Таким образом, Qnet обеспечивает прозрачный доступ к файлам всех узлов сети с помощью обычных локальных средств — файлового менеджера, команды **ls** и т. п. Кроме того, Qnet позволяет запускать процессы на любом узле сети. Для этого предназначена утилита **on**.

Утилита **on** является своего рода расширением командного интерпретатора по запуску приложений. Синтаксис утилиты **on**:

`on` *опции команда*

При этом команда будет выполнена в соответствии с предписаниями, заданными посредством опций. Основные опции этой утилиты:

- **-n** ***узел*** — указывает имя узла, на котором должна быть запущена команда. При этом в качестве файловой системы будет использована файловая система узла, с которого запущен процесс;
- **-f** ***узел*** — то же, что и предыдущая опция, но в качестве файловой системы будет использована файловая система узла, *на* котором запущен процесс;
- **-t** ***tty*** — указывает, с каким терминалом целевой ЭВМ должен быть ассоциирован запускаемый процесс;
- **-d** — предписывает отсоединиться от родительского процесса. Если задана эта опция, то утилита **on** завершится, а запущенный процесс будет продолжать выполняться (т. е. он будет запущен с флагом `SPAWN_NOZOMBIE`);

- **-p** ***приоритет [дисциплина]*** — можно задать значение приоритета и, если надо, дисциплину диспетчеризации.

Например, команда:

```
on -d -n host5 -p 30f -t con2 inetd
```

запустит на консоли `/dev/con2` узла с именем **host5** программу **inetd**, находящуюся на нашем узле. Приоритет запущенного процесса будет иметь значение 30 с дисциплиной диспетчеризации FIFO. После запуска программы **inetd** утилита **on** сразу завершит свою работу и интерпретатор выдаст приглашение для ввода очередной команды.

Здесь уместно вспомнить про то, как администратор пакетной файловой системы **fs-pkg** отображает платформо-зависимые файлы. Как вы помните, такие файлы отображаются не только в корневой каталог `/`, но и в платформо-зависимый каталог, например, `/ppcbe`. Это позволяет брать на удаленном узле файлы, предназначенные для выполнения на аппаратуре конкретного узла сети. Например, мы хотим на нашем бездисковом компьютере на базе процессора PowerPC (Big Endian) запустить утилиту **ls**, расположенную на узле с именем **alpha** (и с процессором Pentium III). Для этого нужно выполнить команду:

```
/net/alpha/ppcbe/usr/bin/ls -l /home
```

Этот механизм дает возможность использовать без конфликтов дисковое пространство всех узлов сети Qnet независимо от аппаратуры, на которой работает QNX Neutrino. В сочетании с возможностями утилиты **on** администратор получает "виртуальную суперЭВМ". Но не забывайте про "ложку дегтя" — в сети Qnet не стоит рассчитывать на безопасность информации.

Информацию о доступных узлах сети Qnet можно посмотреть командой **sin net**. В результате получим список доступных узлов Qnet-сети с информацией о каждом из них:

```
$ sin net
myhost   144M 1    631 Intel 686 F6M8S6
hishost  467M 1    939 Intel 686 F6M8S10
```

Как видно из приведенного фрагмента, выдаются: имя узла, размер доступной оперативной памяти, количество процессоров, тактовая частота и модель процессора.

Примечание

Некоторые утилиты QNX имеют сетевую опцию, указывающую, к какому узлу сети применить действие утилиты (например, **sin** или **slay**).

Глобальные имена процессов

В QNX Neutrino существует механизм "локальных имен процессов", унаследованный от QNX4. Этот механизм основан все на той же регистрации префиксов и поддерживается, соответственно, администратором процессов. Желающие изучить его подробнее могут почитать описание библиотечной функции *name_attach()*. Этот механизм уникален для QNX (т. е. не определен POSIX), поэтому компания QSS не рекомендует им пользоваться. Правда, программистам, воспитанным на QNX4, имена процессов нравятся из-за их простоты и удобства. Однако в QNX Neutrino было серьезное ограничение — не поддерживались в чистом виде глобальные имена процессов, имевшиеся в QNX4 (см. описание администратора глобальных имен **nameloc** в документации QNX4). Что значит "в чистом виде"? Дело в том, что глобальные имена процессов можно без особого труда реализовать на прикладном уровне. В версии 6.3 разработчики QNX решили облегчить жизнь прикладным программистам и написали нужный код сами.

Глобальные имена процессов поддерживаются с помощью администратора **gns** (GNS — Global Name Service). Эта программа может быть запущена либо в серверном (опция **-s**), либо в клиентском (опция **-c**) режиме. При этом она регистрирует префиксы `/dev/name/gns_server` или `/dev/name/gns_client` соответственно. Регистрация имени выполняется прикладной программой с помощью вышеупомянутой функции *name_attach()*. Для регистрации глобального префикса в аргументах этой функции задается флаг `NAME_FLAG_ATTACH_GLOBAL`. Префикс регистрируется или в каталоге `/dev/name/global`, или в каталоге `/dev/name/local`.

Технология Jump Gate

Для обеспечения прозрачности в графической оболочке Photon существует механизм, получивший название Jump Gate.

Технология Jamp Gate основана на использовании серверного процесса **phrelay**, который передает клиентским программам информацию о графическом изображении в Photon. Клиентами **phrelay** могут быть: **Phditto**, **Phindows**, **PhinX**. Подключиться к серверу можно либо через последовательный канал, либо через сеть TCP/IP. Для использования процесса **phrelay** в TCP/IP обычно применяется программа **inetd**. В стандартном файле `/etc/inetd.conf` уже есть (в закомментированном виде) нужная запись, поэтому достаточно просто раскомментировать ее:

```
phrelay stream tcp nowait root /usr/bin/phrelay phrelay
```

Проверьте, что файл `/etc/services` содержит строку (вообще говоря, она там *должна* быть):

```
phrelay       4868/tcp
```

Программы-клиенты кэшируют получаемую информацию, поэтому им достаточно получать данные только об изменениях "картинки". Работающая программа **Phditto** показана на рис. 8.1.

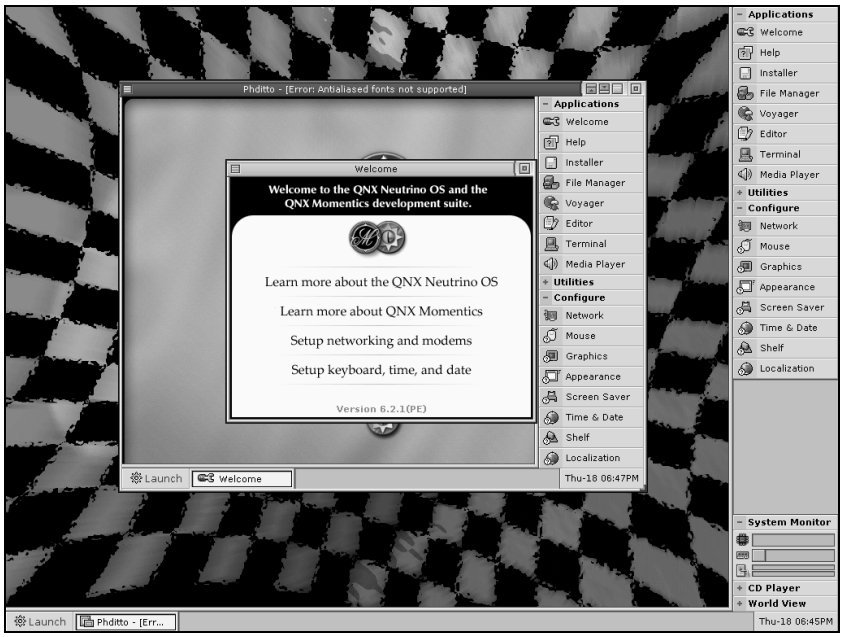


Рис. 8.1. Программа **Phditto** в действии

При удаленном подключении к Photon microGUI по умолчанию создается дополнительный сеанс (session). Чтобы подключиться к существующему сеансу, необходимо указать имя файла нужного сеанса (Named Special Device). Например, подключимся к текущему сеансу Photon узла **host1**:

```
phditto -n /dev/photon host1
```

Для доступа к процессу **phrelay** из Windows предназначена клиентская программа **Phindows** (рис. 8.2).

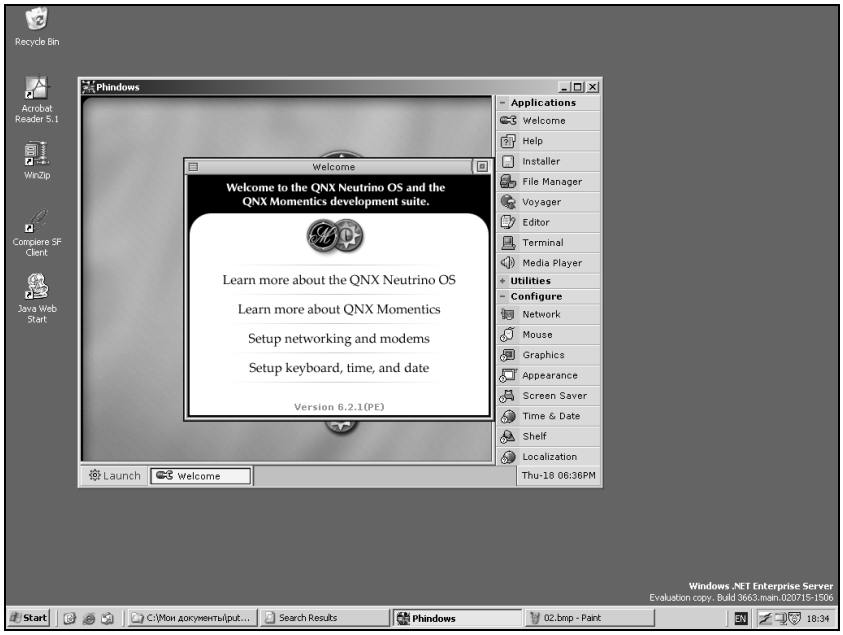


Рис. 8.2. Доступ к процессу **phrelay** из ОС Windows

Для доступа к процессу **phrelay** из ЭВМ, использующей графическую среду X Window System, независимо от аппаратной платформы и операционной системы используется QNX-утилита **PhinX**.

Примечание

Для использования утилиты **PhinX** необходимо на ЭВМ-клиенте разрешить подключение к X-серверу извне локальной командой **xhost**:

```
xhost +имя_машины_qnx
```

Затем нужно подключиться к ЭВМ с QNX через какой-либо протокол удаленного доступа, например:

```
telnet имя_машины_qnx
```

После этого следует установить на удаленном QNX-хосте переменную системного окружения `DISPLAY` в необходимое значение:

```
export DISPLAY=имя_клиентской_машины:0.0
```

и запустить утилиту **phinx**.

Сервер **phrelay** может подключить клиента как к существующему сеансу Photon (тогда клиент получит полный доступ к запущенным приложениям), так и к отдельному, специально созданному сеансу (тогда пользователи будут работать независимо в разных окружениях). Запретить или разрешить подключение к своему сеансу Photon пользователь может, установив или сбросив флажок в окне утилиты **phrelaycfg** (элемент меню **Remote Access**).

Поддержка TCP/IP в QNX

Поддержка стека протоколов TCP/IP обеспечивается с помощью трех модулей, которые могут загружаться администратором сетевого ввода/вывода **io-net**:

- `/lib/dll/npm-ttcpip.so` — облегченный стек TCP/IP для систем с ограниченными ресурсами, реализует часть функциональности полных реализаций TCP/IP стека;
- `/lib/dll/npm-tcpip-v4.so` — стандартная реализация стека протоколов NetBSD v1.5;
- `/lib/dll/npm-tcpip-v6.so` — профессиональный стек протоколов TCP/IP, KAME-расширение стека NetBSD v1.5.

Как вы, наверное, догадались, **npm-tcpip.so** — это просто ссылка на один из указанных модулей.

Для обеспечения безопасности в модуле профессионального стека TCP/IP используется протокол IPSec, который защищает IP-пакет от несанкционированного изменения посредством присоединения криптографической контрольной суммы, вычисленной односторонней хеш-функцией, и/или шифрует

содержимое пакета криптографическим алгоритмом с закрытым ключом.

Для пакетной фильтрации (firewall) и трансляции сетевых адресов (NAT — Network Address Translation) используется перенесенный в QNX вариант пакета IP Filter 3.4.27. Основу пакета IP Filter составляет модуль **ipfilter.so**, загружаемый администратором **io-net**. В состав IP Filter также входит ряд утилит:

- **ipf** — предназначена для изменения списка правил фильтрации;
- **ipfs** — сохраняет и восстанавливает информацию для NAT и таблицы состояний;
- **ipfstat** — возвращает статистику пакетного фильтра;
- **ipmon** — монитор для сохраненных в журнале пакетов;
- **ipnat** — пользовательский интерфейс для NAT.

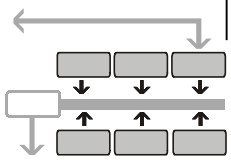
Конфигурация сети TCP/IP хранится в файле `/etc/net.cfg`. Для того чтобы изменения, внесенные в этот файл, вступили в силу, необходимо запустить администратор конфигурирования TCP/IP — утилиту **netmanager**. Обычно для изменения настроек TCP/IP используют графическую утилиту **phlip**, которая автоматически запускает **netmanager** при сохранении изменений (мы пользовались ею при начальном конфигурировании QNX после инсталляции). Например, для запуска **io-net** с поддержкой TCP/IP вручную можно поступить, скажем, так:

```
io-net -d rtl -p tcpip
netmanager
```

Для получения информации о TCP/IP в QNX содержится полный набор общепринятых UNIX-утилит. Вот основные из них:

- **arp** — выполнение ARP-запросов;
- **ifconfig** — настройка параметров IP-интерфейса;
- **if_up** — проверка доступности IP-интерфейса;
- **netstat** — отображение состояния и статистики IP-сети;
- **route** — настройка статической маршрутизации;
- **nslookup** — опрос службы доменных имен DNS;

- **showmount** — получение информации о состоянии сервера NFS;
- **ping** — эхо-запрос узла IP-сети пакетами ECHO_REQUEST протокола ICMP;
- **rpcinfo** — посылка RPC-запросов RPC-серверу и отображение полученной информации об RPC;
- **traceroute** — трассировка маршрута IP-пакетов.



Глава 9

Графический интерфейс пользователя Photon μGUI

В этой главе будут рассмотрены вопросы:

- общие сведения;
- утилиты конфигурирования.

Общие сведения

По аналогии с операционной системой QNX графическая среда Photon μGUI (или microGUI) представляет собой графическое микроядро с семейством процессов, расширяющих функциональность графического микроядра. Все эти процессы взаимодействуют посредством стандартного QNX-механизма передачи сообщений, которые, как известно, легко путешествуют по сети — отсюда сетевые возможности "Фотона".

Само *графическое микроядро*, или *сервер* Photon представляет собой небольшой процесс **`Photon`**. За управление графическим выводом отвечает администратор графического вывода **`io-graphics`**. Администратор графического вывода загружает динамические библиотеки, например интерпретатор графического потока `gri-photon.so`, и видеодрайвер `devg-tnt.so`. Информацию по прорисовке шрифтов **`io-graphics`** получает от сервера шрифтов, имеющего реализации как в виде библиотеки `phfont.so`, так и виде приложения (есть несколько приложений, отличающихся функциональностью).

За ввод информации от мыши, клавиатуры и других устройств отвечает драйвер ввода **`devi-hirun`**.

Утилита **fontsleuth** указывает серверу шрифтов **phfont**, где находятся шрифты.

Фон рабочего стола "рисует" утилита **bkgdmgr**.

За переключение между окнами приложений с помощью комбинации клавиш <Alt> + <Tab> отвечает утилита **wmswitch**, за "хранение экрана" — **saver**.

Для того чтобы обеспечить полнофункциональную графическую среду, позволяющую пользователям манипулировать окнами приложений (изменением их размеров, перемещением, сворачиванием и т. п.), используется *оконный менеджер* **pwm**. Рабочий стол с меню быстрого запуска приложений реализован с помощью процесса *администратора рабочего стола* **shelf**, он же поддерживает панель задач.

Из сказанного ясно, что функциональность графического интерфейса легко конфигурировать — просто нужно выбрать только необходимые компоненты. В среде разработки для запуска Photon'a используется готовый командный сценарий `/usr/bin/ph`. Можно использовать его как образец для запуска графической среды на целевых системах. Этот сценарий в обычной настольной системе запускается одним из двух способов:

- вручную из командной строки в любое время после регистрации в системе;
- из сценария `/etc/rc.d/rc.local` утилитой **tinit**, если не существует файл `/etc/system/config/nophoton`.

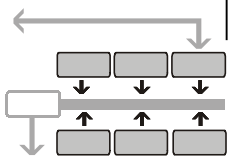
В сценарии `ph` заданы различные детали поведения системы. Например, если существует переменная окружения `LOGNAME` (а это означает, что ее либо нарочно инициализировали ранее в командных сценариях, либо пользователь уже прошел регистрацию с помощью утилиты **login**), то Photon стартует сразу. В противном случае Photon запустит утилиту **phlogin** для регистрации пользователя в системе. Можно запретить пользователю выход из среды Photon в командную строку, задав значение 1 для переменной `PHEXIT_DISABLE`.

Утилиты конфигурирования

Компоненты среды Photon можно конфигурировать с помощью нескольких утилит, указанных в табл. 9.1.

Таблица 9.1. *Утилиты конфигурирования среды Photon*

Что нужно конфигурировать	**Имя утилиты**	**Название элемента меню в Shelf**
Оконный менеджер `pwm` и администратор фона `bkgdmgr`	`pwmopts`	**Appearance**
Шрифты	`fontadmin`	**Fonts**
Администратор графического вывода `io-graphics`	`phgrafx`	**Graphics**
Мышь (на самом деле — драйвер ввода `devi-hirun`)	`input-cfg`	**Mouse**
Разрешение/запрет удаленного подключения к существующему сеансу среды Photon	`phrelaycfg`	**Remote Access**
Хранитель экрана `saver`	`savecfg`	**Screen Saver**



Глава 10

Печать в ОС QNX

В составе дистрибутива есть две системы печати: традиционная (**lpd**) и "родная" (**spooler**).

Традиционная система устарела и поддерживается по историческим причинам; она широко известна в UNIX-подобных ОС и, как следствие, описана во многих книгах. Она основана на сервере печати (спулер) **lpd**, конфигурация которого задается в файле `/etc/printcap`, и клиентских утилитах **lpr**, **lprq**, **lprrm**, **lprc**.

Про родную систему печати мы поговорим подробнее, рассмотрев следующие темы:

- печать в среде QNX;
- печать в разнородной среде.

Печать в среде QNX

Основу "родной" системы печати QNX составляет серверный процесс-администратор **spooler**. Назначение спулера — обеспечивать бесконфликтный доступ нескольких пользователей (или программ) к контролируемому им устройству. Этот процесс автоматически запускается администратором нумерации устройств **enum-devices**. По умолчанию **spooler** контролирует доступ к параллельному порту `/dev/par1`. Если к параллельному порту подключить принтер, то **spooler** автоматически распознает его и путем зондирующих запросов получает параметры принтера.

В каталоге `/etc/printers` содержится несколько файлов конфигурации спулера для разных типов принтеров. При запуске **spooler** выполняет несколько операций.

1. Определяет тип принтера и выбирает соответствующий этому типу файл конфигурации.

2. Сравнивает свойства, указанные в файле конфигурации, с параметрами принтера, полученными при его сканировании.
3. Регистрирует в пространстве имен префикс-каталог `/dev/printers/`*`имя_принтера`*`/`. В этом каталоге создаются каталог для спулинга `spool/`, файлы устройств `phs`, `raw` и файл, соответствующий типу принтера (в случае HP-принтеров — `pcl`).
4. Создает каталог `/var/spool/printers/`*`имя_принтера.имя_хоста`*`/`. В этот каталог отображается содержимое каталога `/dev/printers/`*`имя_принтера`*`/spool`.
5. В каталог спулинга записывается файл с результирующими настройками принтера.

Когда пользователь выдает задание на печать, соответствующий файл помещается в каталог спулинга и **`spooler`** вызывает необходимые фильтры для обработки файла. Результатом обработки является файл в формате, понятном принтеру. Конечный файл посылается в устройство печати (`/dev/par1`).

В составе дистрибутива QNX поставляется несколько фильтров для наиболее популярных моделей принтеров. Кроме того, в состав QNX входит Printer DDK (Driver Development Kit), представляющий собой подробно комментированный пример исходного кода фильтра с инструкцией для разработчиков.

Для управления заданиями можно воспользоваться утилитой **`prjobs`** из состава Photon (меню **Print Manager**).

Печать в разнородной среде

Здесь мы рассмотрим:

- печать из QNX на Windows-сервер печати;
- печать с QNX-клиента на QNX-сервер печати;
- печать из Windows на QNX-сервер печати.

Печать из QNX на Windows-сервер печати

Для того чтобы печатать из QNX на принтер, подключенный к Windows-серверу печати, нам потребуется, во-первых, программа, которая может посылать задания на Windows-сервер

печати по SMB-протоколу, во-вторых, нам нужно заставить процесс **spooler** поверить, что к нашей машине подключен принтер. И наконец, в-третьих, необходимо "отобрать" задание у спулера и передать его программе, посылающей задания Windows-серверу. Для решения поставленных задач понадобится выполнить несколько шагов.

Шаг 1. Нужно установить утилиту **smbclient** (она входит в пакет Samba, поставляемый вместе с диском QNX Momentics PE на дополнительном компакт-диске с ПО "третьих" производителей). Именно эта утилита будет посылать файл (разумеется, в готовом для печати формате) на Windows-сервер. Поскольку в SWD Software используются преимущественно принтеры Hewlett-Packard, у меня шаблон команды печати выглядит так:

```
smbclient //server/LJ2200D "пароль" -U name -c "print имя_файла.pcl"
```

где `server` — имя Windows-сервера печати; `LJ2200D` — сетевое имя принтера; *пароль* — пароль в Windows-сети; *имя_файла* — имя в Windows-сети.

Шаг 2. Нужно установить фильтр, вызывающий утилиту **smbclient**. Этот фильтр должен просто принять в качестве аргумента имя файла для печати и вызвать команду, описанную в предыдущем пункте. Я написал вот такой фильтр (вы можете сделать это гораздо лучше и красивее):

```
#include <stdlib.h>
#include <stdio.h>

int main(int argv, char **argc)
{
       char command[1024];
       sprintf(command, "smbclient //win_server/LJ2200D \
              \"win_password\" -U win_name \
             -c \"print %s\"\n", argc[1]);
       system (command);
       return EXIT_SUCCESS;
}
```

Не забудьте поместить фильтр (давайте назовем его **myfilter**) в каталог `/usr/bin` — именно там **spooler** ищет фильтры.

Шаг 3. Нужно модифицировать подходящий файл конфигурации из каталога `/etc/printers/` для использования фильтра, созданного на шаге 2. Я правил, как можно догадаться, файл `pcl.cfg`. В этом файле есть строка, подключающая фильтр:

```
Filter = phs:$d:phs-to-pcl -m$m
```

Эта строка указывает спулеру, что при появлении в каталоге спулинга PHS-файла прежде, чем отправить его на устройство `$d` (значение переменной `$d` задается опцией **`-d`** при запуске процесса **`spooler`** — по умолчанию это `/dev/par1`), необходимо вызвать фильтр **`phs-to-pcl`**. По сути дела выполняется команда:

```
/usr/bin/phs-to-pcl
      /dev/printers/имя_принтера/spool/имя_файла.phs \
      -mмодель > /dev/par1
```

Фильтр **`phs-to-pcl`** выдает на стандартный вывод PCL-файл, который и отправляется на локальный принтер.

Итак, добавим в файл конфигурации строки, задающие правила фильтрации:

```
Filter = phs:pcl:phs-to-pcl -m$m
Filter = pcl:$d:myfilter
```

Первая строка предписывает спулеру при появлении PHS-файла выполнить такую операцию:

```
/usr/bin/phs-to-pcl
    /dev/printers/имя_принтера/spool/имя_файла.phs \
    -mмодель > /dev/printers/имя_принтера/spool/имя_файла.pcl
```

Вторая строка указывает спулеру при появлении PCL-файла в каталоге спулинга выполнить команду:

```
/usr/bin/myfilter
    /dev/printers/имя_принтера/spool/имя_файла.pcl > \
    /dev/null
```

Фильтр же самым бесцеремонным образом отправляет PCL-файл на Windows-сервер. Почему я указал в качестве целевого устройства `/dev/null`? Об этом — на следующем шаге.

Шаг 4. Нужно перезапустить администратор печати **`spooler`** с новыми параметрами. Для этого сначала остановим уже работающий процесс **`spooler`** (если таковой уже имеется):

```
slay spooler
```

Затем запустим его так, как нам нужно:

```
spooler -d /dev/null -c /etc/printers/pcl.cfg -n
      HP_LaserJet_2200
```

Что делает эта команда? Во-первых, она запускает спулер на ноль-устройстве. Это позволит нам сделать систему печати "тихой и безвредной". Во-вторых, спулеру принудительно указывается файл конфигурации, поэтому зондирование оборудования выполнять не требуется (да и что, собственно, зондировать — принтера у нас реально нет). Наконец, в-третьих, задается имя принтера. Под этим именем мы и увидим наш замечательный принтер в диалоговом окне печати.

Вот и все. Хотелось бы на всякий случай дать пару советов.

- Если у вас черно-белый принтер, не забудьте указать этот факт в настройках принтера в диалоговом окне печати. А чтобы не делать этого всякий раз (настройка не сохраняется), отредактируйте-ка лучше файл конфигурации так, чтобы переменная `InkType` везде была равна `1:"B&W"`:

  ```
  InkType = 1:"B&W"
  ```

- Лучше, чтобы принтер был установлен на доменном контроллере. Иначе утилита **`smbclient`** не всегда может аутентифицироваться при подключении к Windows-серверу.

Печать с QNX-клиента на QNX-сервер печати

Для того чтобы печатать с QNX-клиента на QNX-сервер, необходимо выполнить работу, аналогичную описанной в предыдущем разделе. Только дополнительный фильтр должен не вызывать утилиту **`smbclient`**, а просто копировать файл через Qnet в спулинговый каталог сервера печати. Обратите внимание, каталог `/dev/printers/HP_LaserJet_`*`XXX`*`/spool` не доступен через Qnet. Необходимо будет копировать в каталог `/var/spool/printers/HP_LaserJet_`*`XXX`*`.myname/`. При записи файла в этот каталог спулинга администратор печати **`spooler`** не активизируется автоматически, поэтому файл будет отправлен на принтер тогда, когда спулер выполнит очередную проверку содержимого каталога и обнаружит там файл (это происходит с периодом 2—3 минуты — точно время не замерял). Либо это произойдет

раньше, когда спулер будет активизирован другим заданием, поступившим в каталог `/dev/printers/HP_LaserJet_XXX/spool`, или когда будет запущена утилита **prjobs**.

Печать из Windows на QNX-сервер печати

Для того чтобы предоставить возможность Windows-клиентам печатать на нашем QNX-сервере печати, потребуется уже знакомый пакет Samba. В состав пакета Samba входит два сервера:

- **smbd** (для предоставления доступа к SMB-ресурсам);
- **nmbd** (сервер имен протокола NetBIOS).

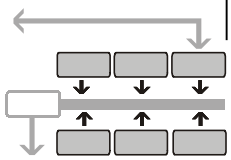
Оба сервера используют общий файл конфигурации `smb.conf`. Он может выглядеть так:

```
[global]
      workgroup = SWD.RU
      netbios name = MYNAME
      security = share
      public = yes

[LaserPrinter]
      comment = HP LaserJet 2200
      path = /dev/printers/HP_LaserJet_2200/spool
      print ok = Yes
      printer driver = HP LaserJet 2000
      print command = mv %s
         /dev/printers/HP_LaserJet_2200/spool/samba.pcl
```

После запуска **smbd** и **nmbd** в сетевом окружении Windows-машин в домене `SWD.RU` появится ЭВМ с именем `MYNAME`, к которой подключен принтер с сетевым именем `LaserPrinter`. Теперь можно на Windows-машине подключить сетевой принтер стандартным способом.

Что же происходит на самом деле? Сервер **smbd** получает от Windows-клиента файл, отформатированный для печати, и просто помещает его в каталог спулинга. Дальнейшую работу выполняет штатная система печати QNX.



Глава 11

Командный интерпретатор

В этой главе будут рассмотрены вопросы:

- знакомство с shell;
- потоки ввода/вывода и конвейеры;
- командные сценарии;
- интерпретаторы для систем с ограниченными ресурсами.

Знакомство с shell

После ввода правильных имени и пароля утилита **login** запускает командный интерпретатор (login shell). Что такое командный интерпретатор? Это программа, которая выдает на экран приглашение (для "простого" пользователя — **$**, для суперпользователя — **#**), принимает команды и запускает их.

Существует ряд реализаций командных интерпретаторов:

- Bourne Shell (**sh**) — интерпретатор, стандартизованный в рамках POSIX 1003.2;
- Korn Shell (**ksh**) — расширение **sh**;
- Bourne Again Shell (**bash**) — значительно измененный **sh**;
- C-Shell (**csh**) — интерпретатор с Си-подобным синтаксисом сценариев;
- другие.

В ОС Linux, например, интерпретатором по умолчанию является **bash**, в QNX — **ksh**. Поскольку **ksh** полностью реализует функциональные возможности **sh**, ради экономии места, с одной стороны, и по традиции — с другой, в QNX файл `/bin/sh` —

символическая ссылка на `/bin/ksh`. В одной системе можно одновременно использовать любые интерпретаторы.

Если пользователь работает в графической оболочке Photon, то командная строка доступна ему с помощью графического псевдотерминала — программы **pterm**.

Поскольку shell является интерпретатором команд, то команда — это базовое понятие. Команды обычно имеют следующий синтаксис:

`<`*имя команды*`>` `[-`*опция* `...] [`*аргумент* `...]`

Например:

```
ls  -ls  /usr/bin
```

где:

`ls` — имя программы-утилиты (утилита **ls** показывает содержимое каталога);

`-ls` — опции: `l` — формат вывода (long); `s` — размер файлов в блоках;

`/usr/bin` — каталог, содержимое которого следует показать.

Эта команда выдаст на экран в длинном формате содержимое каталога `/usr/bin`, добавив при этом информацию о размере каждого файла в блоках. Такой синтаксис является общепринятым, но не обязательным. Для того чтобы точно знать синтаксис конкретной команды, следует обратиться к документации. Пользователи UNIX-подобных систем часто пользуются электронной документацией, доступной через утилиту **man**. В QNX исторически используется другая утилита — **use**. Например, для получения информации о синтаксисе утилиты **ls** нужно выполнить команду

```
use ls
```

Документация **man**-страниц хранится в виде *отдельного* файла, а **use**-сообщение *встроено* внутрь программы, в данном случае — внутрь утилиты **ls**.

Характерна история развития утилиты **tar** (Tape ARchiv, TAR). Чтобы с помощью этой утилиты "распаковать" (извлечь содержимое архива) `archive.tar`, выполняется, например, команда:

```
tar xvf archive.tar
```

где, как вы уже догадались:

`tar` — имя утилиты;

`xvf` — опции (`x` — извлечь; `v` — показывать на экране ход "расTARивания"; `f` — "расTARивать" указанный файл, а не ленту в стримере);

`archive.tar` — имя архивного файла.

Обратите внимание, что перед опциями нет дефиса (-). Этот нюанс постоянно сбивал с толку пользователей, поэтому в современных реализациях утилиты **`tar`** поддерживается два варианта задания опций: с дефисом и без него.

Существует несколько сотен команд, но обычно пользователь применяет лишь несколько десятков. Наиболее часто используемые команды приведены в табл. 11.1.

Таблица 11.1. *Примеры команд*

Что нужно сделать?	**Команда (без аргументов)**
Определить текущий каталог	`pwd`
Просмотреть содержимое каталога	`ls`
Перейти в другой каталог	`cd`
Создать пустой файл	`touch`
Создать каталог	`mkdir`
Копировать файл	`cp`
Копировать каталог	`cp -R`
Удалить файл	`rm`
Удалить каталог	`rmdir` `rm -R`
Просмотреть файл	`cat`
Переименовать файл	`mv`

Команды могут быть как отдельными программами (утилитами), так и "встроенными" командами shell. К встроенным командам относятся **`cd`**, **`pwd`**, **`set`**, **`echo`**, **`alias`** и т. д.

Можно задать несколько команд в одной командной строке, разделив их точкой с запятой (;). Перевод строки (нажатие клавиши <Enter>) воспринимается как конец команды. Если команда длинная и ее необходимо перенести на следующую строку, то перед переводом строки ставят обратный слэш (\). Чтобы запустить команду в фоновом (неинтерактивном) режиме, на конце ставиться амперсанд (&). Например, запустим утилиту **ped** — штатный текстовый редактор Photon из окна псевдотерминала:

```
ped &
```

Интерпретатор сообщит номер фонового процесса и выдаст приглашение ввести новую команду, не дожидаясь завершения фоновой программы.

Если вы забыли добавить амперсанд, введя только

```
ped
```

то можно остановить интерактивный процесс комбинацией клавиш <Ctrl> + <Z>, а затем перевести его в фоновый режим командой **bg**. Перевод последней программы, запущенной в фоновом режиме, в интерактивный режим выполняется командой **fg**.

Если необходимо, чтобы фоновый процесс завершился прежде, чем будут выполнены какие-либо действия, используется команда **wait**. Если в качестве аргумента для **wait** указать PID конкретного процесса, то она будет ждать завершения именно его, а если параметр не задавать, то **wait** будет ожидать завершения всех фоновых процессов, дочерних для данного shell.

Интерактивный процесс можно принудительно завершить, используя комбинацию клавиш <Ctrl> + <C>, а фоновый процесс — с помощью POSIX-утилиты **kill** или QNX-утилиты **slay**. Для утилиты **kill** требуется указывать PID уничтожаемого процесса в качестве аргумента, а для утилиты **slay** в качестве аргумента указывается имя уничтожаемого процесса. Идентификатор процесса (PID) в свою очередь может быть получен либо с помощью POSIX-утилиты **ps**, либо с помощью QNX-утилит **sin** и **pidin**.

Двойной амперсанд && предписывает интерпретатору shell выполнять последующую команду при условии нормального завершения предыдущей, иначе — игнорировать, например:

```
mkdir /tmp/myfolder && cp file1.txt /tmp/myfolder/
```

В этом случае сначала будет предпринята попытка создать каталог `/tmp/myfolder`, а затем (если удалось создать каталог) в него будет скопирован файл `file1.txt`.

Символ `||`, напротив, предписывает интерпретатору shell выполнять следующую команду при ненормальном завершении предыдущей, иначе — игнорировать. Например:

```
mkdir /tmp/myfolder || rm /tmp/*
```

В этом случае сначала будет предпринята попытка создать каталог `/tmp/myfolder`, и, если каталог создать не удалось, — будет удалено содержимое каталога для временных файлов `/tmp`.

Потоки ввода/вывода и конвейеры

Каждая "нормальная" программа получает три стандартных потока ввода/вывода. Каждый из этих потоков представлен в виде файлового дескриптора:

- 0 — стандартный ввод (stdin);
- 1 — стандартный вывод (stdout);
- 2 — стандартный поток диагностических сообщений (stderr), обычно его называют "стандартным потоком ошибок".

Поток 0 по умолчанию связан с клавиатурой, а потоки 1 и 2 — с экраном терминала. Убедиться в этом можно, выполнив команду просмотра дескрипторов открытых файлов:

```
sin fd
```

Стандартные потоки ввода, вывода и ошибок могут быть перенаправлены. Перенаправление стандартного вывода выполняется с помощью символа `>`. Например:

```
ls /usr > /tmp/file1
```

Команда **`ls`** сформирует список содержимого каталога `/usr` и вместо того, чтобы выдать результат на экран, поместит его в файл `/tmp/file1`. Если файла с таким именем не было, то он будет создан. Если же файл `/tmp/file1` уже существовал, то его прежнее содержимое будет уничтожено.

Выполним другую команду:

```
pwd > /tmp/file1
```

Команда **pwd** сформирует полное имя текущего каталога и поместит его в файл /tmp/file1, уничтожив прежнее содержимое файла. А если нам нужно добавить новую информацию к прежней? Тогда используется директива >>, т. е. предыдущая команда будет выглядеть так:

```
pwd >> /tmp/file1
```

В этом случае вывод утилиты **pwd** будет добавлен в конец файла /tmp/file1.

Для перенаправления стандартного ввода используется символ <. Например:

```
wc -c < /tmp/file1
```

Команда **wc** подсчитает и выдаст на экран число символов в файле /tmp/file1.

Строго говоря, при перенаправлении следует указывать номер перенаправляемого потока, однако shell по умолчанию знает, что ">" — это "1>", а "<" — это "0<". Если же необходимо оперировать с другими потоками (например, с stderr), то значение дескриптора должно задаваться явно. Проиллюстрируем:

```
ls /tmp11
```

Если каталог /tmp11 не существует, то команда выдаст сообщение об ошибке ls: No such file or directory (/tmp11). Допустим, мы хотим все диагностические сообщения команды "сбрасывать" в файл /tmp/errlog. Тогда команда должна выглядеть так:

```
ls /tmp11 2> /tmp/errlog
```

Если же сообщения об ошибках нас вообще не интересуют и мы не хотим "засорять" ими экран, то сделаем так:

```
ls /tmp11 2>/dev/null
```

Потоки можно объединять, используя конструкцию >&. Для иллюстрации выполним команду, пишущую в стандартные потоки вывода и ошибок. Для этого попытаемся вывести на экран существующий файл (/tmp/file1) и несуществующий файл (/tmp/file2), стандартный вывод направим в файл /tmp/log:

```
cat /tmp/file1 /tmp/file2 > /tmp/log
```

Теперь модифицируем команду так, чтобы поток ошибок направлялся туда же, куда и поток вывода:

```
cat /tmp/file1 /tmp/file2 > /tmp/log 2>&1
```

В приведенной команде поток 2 добавляется к потоку 1.

Допускается произвольно комбинировать операции перенаправления потоков в одной команде:

```
wc -c </tmp/file1 >/tmp/file2
```

В данном случае утилита **wc** осуществит ввод из файла `/tmp/file1`, а результат запишет в файл `/tmp/file2`.

Вывод одной программы можно объединять с вводом другой. Конструкция, обеспечивающая такое объединение, обозначается символом | и называется *конвейером*:

```
ls -l /usr/bin | wc -l
```

В данном примере утилита **ls** сформирует вывод, состоящий из информации о содержимом каталога `/usr/bin`, результат будет передан на "вход" утилиты **wc**, которая выдаст количество строк в выводе **ls**.

Если программа "умеет" работать и со стандартным вводом, и со стандартным выводом, то ее называют *фильтром* (утилита **wc** — типичный фильтр). В конвейере может использоваться несколько команд, а значит, все команды конвейера, кроме первой и последней, должны быть фильтрами. Например:

```
ls /usr/bin | sort | wc -l > /tmp/file2
```

Уместно добавить, что существует утилита **tee**, позволяющая дублировать поток вывода. Один из потоков вывода, например, можно записать в файл, а другой вывести на экран:

```
ls — l /usr | tee /tmp/lslog
```

Говоря точнее, утилита **tee** свой стандартный ввод направляет на свой стандартный вывод, дублируя его в указанные файлы (в примере задан только один файл — `/tmp/lslog`).

В заключение разговора о потоках хочу напомнить, что устройства в QNX представлены в виде специальных файлов, а значит, механизм перенаправления потоков можно использовать для "сырых" операций чтения/записи с устройствами. Под словом "сырые" я в данном случае имею в виду такие операции, при выполнении которых пользователь сам отвечает за то, что формат передаваемых им данных будет понятен драйверу, а формат вывода драйвера будет понятен ему самому.

Например, выведем сообщение "Hello folks!" в окно псевдотерминала **pterm**, запущенного первым в текущем сеансе Photon, введя в любом другом окне псевдотерминала команду:

```
echo Hello folks! > /dev/ttyp0
```

Или обнулим содержимое файла `/tmp/file1`:

```
cat >/tmp/file1 < /dev/null
```

Можно передавать результат выполнения одной команды в качестве аргумента другой команде, для этого команду-"аргумент" следует заключить в одиночные кавычки (`).

Примечание

Клавиша этого символа обычно находится под клавишей <Esc> в левой верхней части клавиатуры — не путайте с клавишей апостроф ('), расположенной рядом с клавишей <Enter>.

Проиллюстрируем:

```
less `which ph`
```

Команда **which** ищет указанный файл по очереди во всех каталогах, перечисленных в переменной окружения `PATH` (т. е. там же, где ищет команду **shell**), как только находит — выдает на экран путевое имя найденного файла и прекращает поиск. Команда **less** позволяет "прокручивать" длинные текстовые файлы, дозируя их фрагментами, равными доступному пространству экрана (прокручивание выполняется нажатием клавиши <пробел>, выход — <q>). То есть указанная команда позволяет просмотреть текст сценария **ph**, не задумываясь о том, где он физически находится.

Таким образом, перенаправление, слияние, объединение потоков ввода/вывода предоставляет администратору мощный инструмент управления системой. Этот механизм подобен технологиям компонентного программирования, получившим широкое распространение в последние годы. При этом компонентами являются любые программы, как входящие в дистрибутив, так и написанные самим пользователем. Shell же позволяет сочетать их произвольным образом.

Рассмотрим такой пример: пусть нам необходимо получить список файлов каталога `/bin`, которые являются символическими

ссылками, причем для каждого файла в этом списке должна быть указана контрольная сумма. Можно написать программу, которая будет делать эту работу, а можно использовать штатные утилиты QNX. Поиск файлов по заданным критериям выполняет утилита **find**:

```
find /bin -type l
```

Контрольное суммирование выполняет утилита **cksum**. Она принимает список объектов в качестве аргумента, поэтому передадим ей в качестве такого аргумента результат выполнения утилиты **find**:

```
cksum `find /bin -type l`
```

И, наконец, форматирование потока вывода выполняет утилита **awk**. От **awk** нам нужна возможность оставить в выводе утилиты **cksum** только первую и третью колонки, разделенные символом табуляции, причем сначала выведем третью колонку (имя файла), а затем — первую (контрольная сумма). Результат сохраним в файле `result.txt`. Итак:

```
cksum `find /bin -type l` | awk '{print $3 "\t" $1}' >
      /tmp/result.txt
```

Всего-то дел! А теперь попробуйте то же самое сделать графическими средствами любой операционной системы — и вы поймете, почему командная строка так популярна у пользователей UNIX-подобных систем.

Командные сценарии

Прекрасно — теперь мы дошли, пожалуй, до основной сферы применения командных интерпретаторов. Дело в том, что удобно записать команды в текстовый файл, а затем указать интерпретатору shell "выполнить" этот файл. Такой файл называют *командным сценарием* (shell script — отсюда жаргонное "скрипт" или "шэльный скрипт"). По сути дела, интерпретатор реализует язык программирования высокого уровня. На этом языке администраторы могут осуществлять управление вычислительной системой (что, в общем-то, они и делают).

Запуск сценариев на исполнение

Коль скоро shell — это еще и язык программирования, уместно начать его освоение со знаменитой программы "Hello world!". Итак, с помощью любого текстового редактора создадим файл, скажем, `hello.sh`. В него поместим одну строку:

```
echo Hello world!
```

Примечание

Обратите внимание, что последним символом в командном сценарии должен быть символ перевода строки. Ведь, чтобы интерпретатор shell выполнил команду, введенную в командной строке, нужно нажать клавишу <Enter>, не так ли? Так что когда у вас не будет работать какой-нибудь сценарий, не поленитесь проверить его на наличие символа перевода строки в конце файла — начинающим shell-программистам помогает в 50% случаев ☺.

Хорошо, но как будем запускать сценарий? Я знаю по крайней мере четыре способа для этого:

- запустить интерпретатор, указав ему имя сценария в качестве аргумента или связав стандартный поток ввода интерпретатора с файлом сценария. Например:

  ```
  sh hello.sh
  ```

- запустить интерпретатор, связав стандартный поток ввода интерпретатора с файлом сценария. Например:

  ```
  sh < hello.sh
  ```

- выполнить сценарий в текущем экземпляре shell командой "точка" (.). Например:

  ```
  . hello.sh
  ```

- установить для сценария файловый атрибут "исполняемый" и запустить как обычную утилиту. Например:

  ```
  chmod  755 cmd
  hello.sh
  ```

Самым распространенным является третий способ. Только не забудьте, что при третьем способе файл должен или находиться в одном из каталогов, перечисленных в переменной окружения `PATH`, либо для запуска надо указывать полное путевое имя сценария.

Второй способ тоже достаточно важен. Например, встраиваемая система управления базами данных реального времени Empress может использовать большое количество переменных окружения, особенно необходимых при разработке приложений. Производитель поставляет командные сценарии, задающие значения этих переменных. Если запустить такой сценарий первым или третьим способом, то произойдет вот что: запустится новый экземпляр shell, выполнит команды из сценария и благополучно завершится, не изменив ничего в системе. А если применить команду "точка", то в нашем рабочем экземпляре shell будут установлены нужные переменные и мы будем наслаждаться инструментами Empress.

Все это замечательно, но вам уже известно, что в природе есть много командных интерпретаторов. Язык программирования каждого из них имеет свой синтаксис и свои особенности реализации. Как, например, при третьем способе запуска будет определяться интерпретатор, необходимый для выполнения сценария? Для этой цели в самой первой строке сценария помещают символы `#!` и, без пробела, имя интерпретатора, для которого написан сценарий, например — `#!/bin/bash`.

Примечание

Такая конструкция может быть только в первой строке. Во всех остальных случаях первый же символ `#` считается началом комментария, заканчивающегося символом конца строки.

Модифицируем наш файл `hello.sh`:

```
#!/bin/ksh
echo Hello world!  # This is comment
# This is comment too
```

Переменные и параметры сценариев

Ярким и типичным примером переменных, используемых интерпретатором shell, являются переменные системного окружения. Аналогичным способом мы можем задавать собственные переменные для наших сценариев или модернизировать существующие. Имя переменной shell выбирается так же, как это принято в большинстве языков программирования, т. е. это может

быть последовательность букв, цифр и символов подчеркивания, начинающаяся с буквы или символа подчеркивания. Обычно имена переменных для shell задают из последовательности заглавных букв в сочетании с символами подчеркивания, но это просто необязательная традиция.

Переменные в shell задаются сразу со значением (можно с пустым). При объявлении переменной и она, и ее значение должны быть записаны *без пробелов* относительно символа равенства (=).

Примечание

Пробелы возле символа = — типичная ошибка начинающих shell-программистов.

Например, создадим переменную `MY_VAR` со значением `aaaa`:

```
MY_VAR=aaaa
```

или переменную `MY_VAR1` без значения:

```
MY_VAR1=
```

В отличие от обычных языков программирования, значение shell-переменной — всегда строка *символов*. То есть команда:

```
MY_VAR2=123
```

создает переменную `MY_VAR2`, значением которой является строка ASCII-символов "1", "2" и "3", но никак *не число* 123!

Если для переменной задается значение, содержащее пробелы, то нужно заключить его в кавычки:

```
MY_VAR3="QNX Neutrino RTOS v.6.3.0"
```

Чтобы получить доступ к значению переменной, надо перед ее именем поставить символ `$`. Например, чтобы с помощью команды **`echo`** вывести на экран содержимое переменной `MY_VAR3`, надо выполнить такую команду:

```
echo $MY_VAR3
```

Помните, мы говорили, что результат выполнения одной команды можно передать в качестве аргумента другой команде? Точно так же результат выполнения команды можно присвоить переменной в качестве значения, например:

```
DATE=`date`
```

При этом сначала выполнится команда **`date`**, а результат ее выполнения вместо стандартного вывода приписывается в качестве значения переменной `DATE`.

Если нужно, чтобы имя переменной не сливалось с другими символами, то имя переменной заключают в фигурные скобки. Так команда:

```
echo ${MY_VAR3}beta1
```

выдаст сообщение:

```
QNX Neutrino RTOS v.6.3.0beta1
```

Если необходимо, чтобы сценарий запрашивал ввод значений переменных, используется команда **read**. Она обеспечивает работу со сценарием в диалоговом режиме, например:

```
read A
```

По этой команде сценарий остановится, ожидая ввода. От пользователя требуется ввести какое-либо значение и нажать клавишу <Enter>. Введенное значение будет присвоено переменной `A`, и сценарий продолжит выполнение. Таким способом можно присвоить значения сразу нескольким переменным, например:

```
read A B C
```

Для того чтобы попрактиковаться, напишем сценарий `script1.sh`:

```
#!/bin/ksh
echo -n Input variables A, B, and C:
read A B C
echo A=$A
echo B=$B
echo C=$C
```

Затем сделаем наш сценарий исполняемым и запустим:

```
chmod a+x script1.sh
./script1.sh
```

На запрос сценария `Input variables A, B, and C:` введем строку **`111 222 333`**, не забыв после этого нажать клавишу <Enter> (про то, что для выполнения команды или ввода данных всегда надо нажимать клавишу <Enter>, больше повторять не буду ☺).

На экране мы увидим следующее:

```
Input variables A, B, and C:111 222 333
```

Теперь, эксперимента ради, снова запустите сценарий, но введите строку **`111 222 333 444 555`** и посмотрите, как изменится результат.

Примечание

Для того чтобы можно было использовать служебные символы в составе строковых переменных, применяется экранирование — обратный слэш (\).

Если, скажем, переменной A надо присвоить в качестве значения строку $B, то это делается так:

```
A='$B"
```

Тогда команда **echo A=$A** выдаст на экран: A=$B.

Мы уже обсуждали, что переменные в интерпретаторе shell имеют строковый тип, однако shell предоставляет некоторые средства, позволяющие обрабатывать переменные как целые числа. Для этого предназначена команда **expr**. Напишем сценарий script2.sh:

```
#!/bin/ksh
x=50 y=4
a=`expr $x + $y`; echo a=$a
b=`expr $y - $x`; echo b=$b
c=`expr $x / $y`; echo c=$c
d=`expr $x "*"  $y`; echo d=$d
e=`expr $x % $y`; echo e=$e
```

Запустим этот сценарий. На экране получим:

```
a=54
b=46
c=12
d=200
e=2
```

Заметьте, что в результате деления получаем только целую часть результата. Остаток от деления получается с помощью операции %. Заметьте также, что символ умножения (звездочка — *) взят в кавычки (т. е. "экранирован"), т. к. * — служебный символ shell, обозначающий произвольную строку.

Примечание

Не забудьте, что переменные и знаки операций *обязательно разделяются пробелами*. Отсутствие таких пробелов — очень распространенная ошибка в shell-программировании.

Все переменные доступны только в том процессе **ksh**, в котором они были объявлены (вспомните назначение команды "точка"). Для того чтобы к переменным можно было обратиться из процессов, дочерних по отношению к данному **ksh**, переменные следует "экспортировать". Это можно сделать двумя способами. Первый — создать переменную, а затем в любое время экспортировать ее:

```
VAR1=myvar
...
export VAR1
```

Второй способ — экспортировать переменную сразу при создании:

```
export VAR1=myvar
```

Посмотреть, какие из переменных данного интерпретатора экспортированы, можно, выполнив команду **export** без аргументов.

Примечание

Все переменные интерпретатора shell (экспортированные и не экспортированные) выводятся на экран командой **set** без аргументов.

Внутренние переменные shell

Интерпретатор поддерживает некоторые переменные, которым он присваивает значения самостоятельно (табл. 11.2).

Таблица 11.2. *Внутренние переменные интерпретатора*

Переменная	Значение
?	Код завершения последней команды (0 — нормальное завершение)
$	PID текущего процесса **ksh**
!	PID последнего процесса, запущенного в фоновом режиме
#	Число параметров, переданных в shell
*	Перечень параметров shell как одна строка

Таблица 11.2 *(окончание)*

Переменная	Значение
@	Перечень параметров shell как совокупность слов
-	Флаги, передаваемые в shell

Различия между $* и $@ заключаются в том, что если $* имеет значение, например "111 222 333", то $@ будет иметь значение "111" "222" "333".

Параметры сценариев

Как и любая программа, сценарий может принимать аргументы. Ранее, говоря о внутренних (встроенных) переменных shell, я упомянул о переменных, содержащих число и перечень параметров. Как же прочитать эти параметры? Для этой цели shell поддерживает специальные переменные, имена которых состоят из одной цифры в диапазоне от 0 до 9. При этом переменная 0 содержит имя самого сценария, переменная 1 — первый параметр, переменная 2 — второй параметр и т. д. Общее число параметров, как вы помните, содержится в переменной #.

Замечательно, но если вдруг сценарию нужно передать, скажем, 30 аргументов? Передать-то, понятно, передадим — дело немудреное, но как получить к ним доступ? А для этого есть встроенная команда **shift**. Чтобы понять принцип ее использования, вспомните, как вы просматриваете длинный документ Word: в один момент можно видеть только фрагмент текста, соответствующий рабочему пространству окна Word; чтобы посмотреть другие части документа, нужно "перемещать" рабочее пространство окна вверх или вниз. Аналогично, переменные 1—9 являются этим самым окном для просмотра аргументов. Перемещается это окно с помощью команды **shift** на нужное число позиций. Переменная 0 всегда содержит имя сценария. Для наглядности рассмотрим пример script3.sh:

```
#!/bin/ksh
echo "Parameters =" ${#}
echo ------------------
echo "Param0 =" $0
```

```
echo "Param1 =" $1
echo "Param2 =" $2
echo "Param3 =" $3
echo ------------------
shift 2
echo "Param0 =" $0
echo "Param1 =" $1
echo "Param2 =" $2
echo "Param3 =" $3
```

Запустим его, задав несколько аргументов, и посмотрим, что получилось:

```
chmod a+x script3.sh
./script3.sh  AAA BBB CCC DDD EEE
```

Отладка сценариев

Для запуска сценария в подобии отладочного режима предназначена команда **set**. Есть два режима, представляющих интерес с точки зрения отладки.

- Устанавливается командой **set –v**. Указывает интерпретатору выводить все строки, которые он считывает. Режим отменяется командой **set +v**.
- Этот режим при отладке используется чаще. Устанавливается командой **set -x**, указывающей интерпретатору выводить на экран команды перед их выполнением. Режим отменяется командой **set +x**.

Операторы языка программирования Korn Shell

Как же можно написать полноценную программу, не используя циклы и проверку условий? Korn Shell поддерживает ряд специальных операторов (команд), основные из которых:

```
test
if
case
for
while
until
```

Оператор *test*, или *[ ]*

Оператор `test` проверяет выполнение условия. Я не встречал этот оператор в сценариях сам по себе, зато в сочетании с оператором `if` он очень распространен. Результат выполнения оператора имеет значение "истина" или "ложь". Это значит, что если есть необходимость использовать оператор `test` в интерактивном режиме, то его результат будет содержаться в переменной `?` (0 — "истина", 1 — "ложь"). У оператора два равнозначных формата, причем в равной степени в сценариях используются оба:

```
test условие
```

или

```
[ условие ]
```

Примечание

Между квадратными скобками и условием *обязательно должны быть пробелы*. Отсутствие этих пробелов — очень распространенная ошибка.

Условия могут быть нескольких, если можно так выразиться, типов:

- проверки файлов;
- сравнения строк;
- сравнения целых чисел.

Некоторые условия проверки файлов:

- `test -f myfile.txt` — "истина", если `myfile.txt` — обычный файл;
- `test -d mydir` — "истина", если `mydir` — каталог;
- `test -c myfile` —"истина", если `myfile` — символьное устройство;
- `test -r myfile.txt` — "истина", если `myfile.txt` доступен по чтению;
- `test -w myfile.txt` — "истина", если `myfile.txt` доступен по записи;
- `test -s myfile.txt` — "истина", если `myfile.txt` не пустой.

Некоторые условия сравнения строк:

- `test aaa = bbb` — "истина", если `aaa` совпадает с `bbb`;
- `test aaa != bbb` — "истина", если `aaa` не совпадает с `bbb`;
- `test -n $A` — "истина", если переменная `A` задана (не пустая);
- `test -z $A` — "истина", если `A` — пустая (т. е. не имеет значения).

Обратите внимание, что оператор `test` возвращает значение "истина", если *условие* — непустая строка.

POSIX определяет два стандартных "условия", которые можно использовать вместо оператора `test`, — `true` и `false`. Чему равны эти "условия", догадайтесь сами ☺.

Некоторые условия сравнения целых чисел:

- `test x -eq y` — "истина", если `x=y`;
- `test x -ne y` — "истина", если `x!=y`;
- `test x -gt y` — "истина", если `x>y`;
- `test x -ge y` — "истина", если `x>=y`;
- `test x -lt y` — "истина", если `x<y`;
- `test x -le y` — "истина", если `x<=y`.

В одном операторе `test` можно задать несколько условий, комбинируя их с помощью логических операций:

- `!` — НЕ
- `-o` — ИЛИ
- `-a` — И

Оператор *if*

В общем случае этот оператор условия имеет такой синтаксис:

```
if  условие
 then  список_операторов
 [elif  условие
       then  список_операторов]
       [else  список_операторов]
 fi
```

где:

условие — любой оператор, возвращающий значение `true` или `false` (обычно используется оператор `test`);

список_операторов — операторы, разделенные точкой с запятой (`;`).

Обратите внимание на оператор `elif` — это сокращение от "else if". В квадратные скобки взяты необязательные элементы конструкции. Заметим, что конструкция оператора `if` заканчивается служебным словом `fi` ("if" наоборот).

Проиллюстрируем сказанное простейшим примером — сценарием `script4.sh`:

```
#!/bin/ksh
if [ -f /tmp/myfile ]
      then cksum /tmp/myfile
      else ls /etc > /tmp/myfile; cksum /tmp/myfile
fi
```

Оператор *case*

Оператор выбора `case` имеет такой синтаксис:

```
case  строка  in
      шаблон)  список_операторов;;
      шаблон)  список_операторов;;
           ...
esac
```

где:

строка — переменная или последовательность символов, которая последовательно сравнивается с шаблоном *шаблон*. Как только *строка* и *шаблон* совпадут, выполняется соответствующий *список_операторов*. Как, вероятно, вы уже поняли, `in` и `isac` — это служебные слова, причем завершает всю конструкцию `esec` ("case" наоборот).

Примечание

Список операторов для каждого шаблона заканчивается двойной точкой с запятой `;;` (в аналогичной конструкции языка Си второй точке с запятой соответствует служебное слово `break`).

Проиллюстрируем сказанное простейшим примером — сценарием script5.sh:

```
#!/bin/ksh
echo -n "Input command: "
read Command
case $Command in
      a) echo Hello!;;
      b) ls -l /usr/bin;;
      c) cd /usr; pwd;;
      *) echo "Wrong command! Usage a, b, or c";;
esac
```

Как вы понимаете, шаблону * строка будет соответствовать всегда (в языке Си для этой цели используется служебное слово default).

Оператор *for*

Оператор цикла for имеет такой формат:

```
for имя [in список_значений]
      do
          список_операторов
      done
```

Оператор работает так. На каждой итерации цикла переменной имя последовательно присваиваются значения из списка *список_значений* и выполняются команды из списка *список_операторов*. *Список_операторов* заключен между служебными словами do и done.

Приведем хрестоматийный прием использования оператора for — сценарий script6.sh:

```
#!/bin/ksh
LIST=`ls /etc`
for i in $LIST
    do
      if [ ! -d $i ]
      then cksum /etc/$i
      fi
    done
```

В результате выполнения этого сценария на экран будет выведена информация с размером и контрольными суммами содержимого каталога `/etc` (без учета содержащихся в нем подкаталогов).

Если в операторе `for` нет конструкции `in` *`список_значений`*, то *`список_значений`* будет равен внутренней переменной `@` (см. ранее).

Оператор `for` не подходит для организации бесконечных циклов, т. к. он работает, перебирая элементы вполне конечного списка.

Операторы *while* и *until*

В отличие от оператора `for`, оператор `while` позволяет организовывать бесконечные циклы, т. к. следующая итерация инициируется всегда, если только условие истинно:

```
while условие
do
список_операторов
done
```

В этом операторе можно задавать счетчики циклов, параметры выхода из цикла (служебное слово `break`) и т. д. Говоря кратко, оператор `while` соответствует традиционным представлениям о нем. Если *`список_операторов`* оператора `while` выполняется тогда, когда *`условие`* истинно, то в операторе `until` напротив — когда *`условие`* станет истинным, цикл завершится:

```
until условие
      do
          список_операторов
      done
```

Разумеется, цикл можно принудительно прервать командой `break`.

Пустой оператор

Иногда по правилам синтаксиса в сценарии на каком-то месте должен обязательно стоять оператор, а никаких действий предпринимать не хочется. В таком случае используется пустой оператор "двоеточие" (`:`).

Функции в сценариях

Как и в любом языке программирования, в интерпретаторе shell можно описывать функции и затем вызывать их из любого места сценария. Определение функции выполняется очень просто:

```
имя ()
{
    список_операторов
}
```

В любом месте программы, где нужно вызвать функцию, просто указывается ее имя как команда. Функции можно передавать аргументы так же, как аргументы передаются сценарию. Мало того, доступ к аргументам выполняется в функции так же, как в сценарии, т. е. через переменные `0—9`. Другими словами, внутри функции переменные `0—9` рассматриваются как аргументы функции, а в остальных местах сценария — как аргументы сценария. Разумеется, при вызове функции новый процесс не создается. Если функция должна вернуть код завершения, то используется оператор `return`.

Обработка сигналов

Для обработки сигналов в shell предусмотрен оператор `trap`. Его синтаксис:

```
trap "список_операторов" список_сигналов
```

По сути дела, *список_операторов* — это своего рода обработчик сигналов *список_сигналов*.

Например, создадим обработчик для сигнала SIGUSR1 (с номером 16). Пусть при получении этого сигнала сценарий уничтожит содержимое каталога `/tmp`, выведет некоторое сообщение на экран и завершится с кодом 100:

```
#!/bin/ksh
echo My PID = $$
trap "rm /tmp/*; echo Good By!; exit 100" 16
while true
      do
      :
      done
```

Интерпретаторы для систем с ограниченными ресурсами

Использовать интерпретатор **ksh** в системах с ограниченными ресурсами — "дорогое" удовольствие. Поэтому в составе QNX поставляются "урезанные" интерпретаторы:

- **esh** — встраиваемый командный интерпретатор (Embedded Shell);
- **uesh** — малый встраиваемый интерпретатор (Micro-embedded Shell);
- **fesh** — "жирный" встраиваемый интерпретатор (Fat embedded shell).

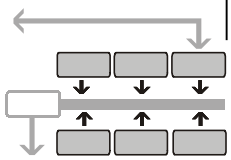
Встраиваемый командный интерпретатор **esh** предоставляет подмножество функциональных возможностей **ksh**. При запуске выполняет простые команды из файла `/etc/esh`. Этот интерпретатор не позволяет писать программы с условными операторами и т. п.

Малый встраиваемый интерпретатор **uesh** в отличие от **esh** не поддерживает:

- интерактивный режим;
- сценарии;
- конвейеры;
- псевдонимы (aliases);
- дополнение имен файлов и команд;
- команду **set**.

"Жирный" встраиваемый интерпретатор **fesh** — небольшое расширение **esh**. Этот вариант shell появился относительно недавно из-за того, что аппаратура современных встраиваемых систем достаточно производительна, чтобы не использовать столь "хилые" интерпретаторы, как **esh** и **uesh**.

Глава 12



Инструменты разработчика QNX

Прежде чем говорить о разработке приложений для ОСРВ QNX, вспомним, что существует два метода: резидентная разработка и кросс-разработка. При *резидентной разработке* весь цикл разработки приложения (написание кода, компиляция, прогон, отладка и т. п.) выполняется на инструментальной машине и это, разумеется, должна быть *x*86-совместимая ЭВМ под управлением непосредственно QNX Neutrino. При *кросс-разработке* прогон и отладка кода могут выполняться только на целевой ЭВМ. Каким методом пользоваться — решайте сами. Замечу лишь, что подавляющее большинство QNX-программистов используют резидентную разработку.

Комплект разработчика QNX Momentics содержит ряд инструментов:

- системы программирования;
- комплекты разработчика;
- интегрированная среда разработки;
- средства визуального моделирования;
- средства разработки графических интерфейсов пользователя;
- средства управления версиями.

Часть из этих инструментов разработана непосредственно компанией QSS, часть другими компаниями (например, Motorola), часть перенесены из Linux. Портированное из Linux программное обеспечение распространяется на условиях GNU GPL (кстати, достаточно жесткой для конечного пользователя лицензии) и технической поддержкой не обеспечивается. Поэтому наиболее полезные OpenSource-продукты постепенно замещаются коммерческими аналогами.

Системы программирования

Компания QSS традиционно не вела самостоятельную разработку инструментальных средств для своих ОСРВ. Например, в QNX4 штатным инструментом была система программирования C/C++ компании Watcom (сейчас эта торговая марка принадлежит корпорации Sybase).

К системам программирования, поставляемым с QNX, относятся:

- C/C++ (GNU CC) — пока это основной инструмент, используемый для создания приложений под QNX. Для защиты пользователей от "острых углов" лицензии GNU GPLL (general public library license) с коммерческими дистрибутивами QNX поставляются коммерческие библиотеки. Для целевых систем *x*86 планируется использовать продукты производства Intel (например, icc);
- Java (на основе IBM j9) — значение этого инструмента постепенно повышается с ростом популярности платформы Java 2 MicroEdition и проекта Eclipse;
- Pascal (проект OpenPascal.org) — для тех, кому это надо;
- и ряд других (perl, python, tcl/tk, ksh, zsh, algol68 и т. д.). Напомню, что большинство из этих продуктов перенесено из Linux энтузиастами, часто — для демонстрации совместимости QNX с Linux на уровне исходных кодов.

По программированию во всех этих системах есть масса специализированных публикаций как на прилавках книжных магазинов, так и в Интернете. Поэтому мы не станем глубоко погружаться в эту тему, однако разработку приложений на языке C (основном языке системного программирования для QNX), как выявила практика, все-таки следует немного затронуть.

Как нас учили в детстве, для получения приложения или библиотеки следует написать программу на одном из языков программирования (в данном случае — на C или C++). Файлы сначала обрабатываются препроцессором, затем то, что получилось, обрабатывается компилятором C или C++. В результате получаем объектный файл. С этим файлом мы можем поступать поразному: включить его в статическую библиотеку или обработать компоновщиком ("линкером"), получив в результате исполняемый файл или разделяемую библиотеку (по желанию).

Создание приложения

На некоторое время забудем про статические и динамические библиотеки и будем говорить только об исполняемых файлах. Итак, для получения приложения исходный текст обрабатывается таким конвейером: препроцессор — компилятор — компоновщик. Эту цепочку удобнее всего вызывать командой **qcc**. Удобство создается тем, что у этой команды есть конфигурационные файлы, позволяющие ей задавать по умолчанию оптимальную комбинацию опций для всех перечисленных инструментов (см. каталог `/etc/qcc`). Давайте создадим текстовой файл с именем `hello.c` и наполним его банальным текстом на языке C:

```
main()
{
      printf("Hello world!\n");
}
```

Обратите внимание, что я не потрудился написать директиву препроцессора `#include <stdio.h>` — **qcc** сама знает, что без `stdio.h` не обойтись.

Для того чтобы получить программу **hello** для Intel-совместимых целевых систем, следует выполнить команду

```
qcc -Vgcc_ntox86 hello.c -o hello
```

Получить полный список целевых платформ, для которых можно сгенерировать приложение, можно командой

```
qcc -V
```

Например, `gcc_ntoppcbe` — для целевых систем PowerPC BigEndian, а `gcc_ntomipsle` — для MIPS LittleEndian.

Замечу, что отлаживать кросс-платформенное приложение удобнее всего все-таки с QNX IDE, т. к. в этом случае от нас требуется только запустить на целевой системе Target Agent — программу **qconn** — и подключиться к нему из IDE.

Программа **qconn** запускается от имени суперпользователя. Еще одна деталь — для отладки на целевой системе должны быть запущены отладочный агент **pdebug** (**qconn** запускает его автоматически) и драйвер псевдотерминалов **devc-pty** (он нужен для **pdebug**). Все это остается в силе и в случае резидентной разработки,

т. к. для IDE целевая система — это та машина, на которой запущена программа **qconn**.

С QNX Neutrino 6.3 поставляются два GCC-компилятора: версия 2.95.2 (используется по умолчанию) и 3.3.1. Откомпилируем ту же программу компилятором посвежее:

```
qcc -V3.3.1,gcc_ntox86 hello.c -o hello_new
```

Может это и случайность, но размер **hello_new** у меня получился меньше, чем у **hello**.

Построение библиотек

Библиотеки, как вы уже знаете, могут быть статическими и динамическими. *Статическая библиотека* представляет собой обычный архив о-файлов, имеющий формат ar и расширение `a` (например, `libsocket.a`).

Разделяемая библиотека представляет собой файл того же формата, что и приложение, — ELF (Executable and Linking Format), но в отличие от приложения, не имеет так называемой *точки входа* — функции *main()*. Разделяемая библиотека имеет расширение `so` (Shared Object), после которого может стоять номер версии, например `libmy.so.2`.

В качестве иллюстрации к сказанному рассмотрим простейший "проект", состоящий из файлов `defs.h`, `main.c`, `aaa.c`, `bbb.c`, `ccc.c` и `Makefile` (какой же проект может обойтись без файла `Makefile`?). Прошу не обращать внимание на бессмысленность примера — он демонстрирует только технологию построения библиотек.

Заголовочный файл содержит объявления функций `aaa`, `bbb` и `ccc`:

```
#ifndef            _MY_DEFS_
#define            _MY_DEFS_
#ifndef            _EXT
#define            _EXT            extern
#endif            // _EXT
_EXT void aaa();
_EXT void bbb();
```

```
_EXT void ccc();
#endif            // _MY_DEFS_
```

Файл `main.c` содержит определение функции *main()*, назначение которой — вывести на экран сообщение и последовательно вызвать функции `aaa`, `bbb` и `ccc`:

```
#include "defs.h"
int main()
{
      printf("I'm main\n");
      aaa();
      bbb();
      ccc();
      return 1;
}
```

Остальные файлы содержат только определение одноименных функций, печатающих сообщения на экране.

Файл `aaa.c`:

```
#include"defs.h"
void aaa()
{
      printf("I'm aaa\n");
}
```

Файл `bbb.c`:

```
#include"defs.h"
void bbb()
{
   printf("I'm bbb\n");
}
```

Файл `ccc.c`:

```
#include"defs.h"
void ccc()
{
      printf("I'm ccc\n");
}
```

И, наконец, файл `Makefile`. У него три цели[1] — static (пример сборки статической библиотеки), dynamic (пример сборки динамической библиотеки) и clean (уничтожение объектных файлов):

```
OBJS = aaa.o bbb.o ccc.o
.PHONY=clean
static: main.o $(OBJS)
      ar -q libmy.a $(OBJS)
      $(CC) -o $@  -L ./  main.o -Bstatic -lmy
dynamic: main.o
      $(CC) -c aaa.c -shared
      $(CC) -c bbb.c -shared
      $(CC) -c ccc.c -shared
      $(CC) -o libmy.so -shared $(OBJS)
      $(CC) -o $@ main.o -L ./ -Bdynamic -lmy
$(OBJS): defs.h
clean:
      $(RM) *.o
```

Сначала выполним команду

```
make static
```

При этом сначала будет создано три объектных файла `aaa.o`, `bbb.o` и `ccc.o`, из которых получится статическая библиотека `libmy.a`. Затем будет создана программа **`dynamic`**, которой для работы требуется библиотека `libmy.so`.

Чтобы можно было запускать программу **`dynamic`**, необходимо добавить путь к динамической библиотеке в переменную окружения `LD_LIBRARY_PATH` — ее использует динамический компоновщик для поиска разделяемых объектов. Например, добавим в переменную текущий каталог (`.`):

```
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
```

Ну вот, можно запускать приложение:

```
./dynamic
```

Теперь выполните команду **`ls -l`** и обратите внимание на размеры файлов `dynamic` и `libmy.so`. Сравните их с размерами файлов `static` и `libmy.a`.

---

[1] Цель — это логическая конструкция файла `Makefile`.

Теперь приступим к сборке динамической библиотеки. Сначала уберем старые объектные файлы:

```
make clean
```

Затем выполним команду:

```
make dynamic
```

При этом сначала будут созданы:

- три готовых к включению в разделяемую библиотеку объектных файла `aaa.o`, `bbb.o` и `ccc.o`, из которых будет создана разделяемая библиотека `libmy.so`;
- объектный файл `main.o`.

Затем файл `main.o` будет скомпонован с библиотекой `libmy.a`, в результате чего получится приложение **`static`**, которое вы можете запустить. Затем выполните команду **`ls -l`** и обратите внимание на размеры файлов `static` и `libmy.a`.

Добавлю лишь, что мы с вами рассмотрели тип разделяемого объекта, который должен быть обязательно загружен при запуске приложения. Однако можно загружать и выгружать разделяемые объекты по мере надобности с помощью функций *dlopen()* и *dlclose()*. Надо сказать, что в документации QSS эти два типа разделяемых объектов удобства ради называются по-разному (соответственно Shared Library и DLL).

Комплекты разработчика

К комплектам разработчика относятся:

- DDK (Driver Development Kit) — пакеты разработки драйверов. Включает пакеты по типам драйверов: Audio DDK, Network DDK, Graphics DDK и т. д.;
- Embedding Tools — средства встраивания. К ним относятся инструменты построения загружаемых образов QNX Neutrino, образов встраиваемых файловых систем, комбинирования образов, работы с Flash-памятью и т. п.;
- BSP (Board Support Packages) — пакеты для использования QNX на процессорных платах различных производителей. Включают специализированные модули IPL, Startup и, при необходимости, драйвер бортовой Flash-памяти. BSP можно

адаптировать для неподдерживаемых процессорных плат, например, собственной разработки. Для этого в составе QNX Momentics PE есть исходные коды IPL, Startup и драйверов. Несколько подробнее об IPL и Startup рассказано в *гл. 13*;

- HAT (High Availability Toolkit) — комплект обеспечения повышенного коэффициента доступности (про этот пакет поговорим отдельно);
- SAT (System Analysis Toolkit) — пакет трассировки событий микроядра (этому пакету тоже стоит уделить немного внимания отдельно);
- Power management framework — средства, необходимые для управления режимом некоторых типов устройств питания;
- Qnet SDK — комплект разработчика, предназначенный для построения специализированных вариантов протокола Qnet.

Интегрированная среда разработки

При выборе элемента меню **Integrated Development Environment** открывается окно, именуемое "рабочим местом" (workbench). При первом запуске QNX IDE создаст в вашем домашнем каталоге папку с именем `workspace`, с содержимым которой IDE и будет работать. Это содержимое называется "ресурсами", к ним относятся папки и файлы, составляющие проекты.

"Рабочее место" содержит редакторы и представления. *Редактором* (editor) называется элемент (plug-in) IDE, позволяющий просматривать и/или модифицировать ресурсы, — например, редактор C-файлов, редактор Make-файлов. *Представлением* (view) называется элемент IDE, позволяющий манипулировать ресурсом, отображая (представляя) его в каком-то логическом виде. К представлениям относятся, например, различные навигаторы и таблицы свойств элементов проектов.

Набор редакторов и представлений, оптимизированный для выполнения какой-либо специализированной задачи, называется *перспективой* (perspective). При первом старте IDE вы увидите перспективу **Resource**. Кроме нее, есть ряд предопределенных перспектив, например:

- **C/C++ Development** — для проектов на языках C/C++;

- **Java** — для Java-проектов;
- **Debug** — для отладки программ;
- **Plug-in Development** — для разработки новых элементов IDE;
- **QNX Application Profiler** — для профилирования разработанных программ;
- **QNX System Builder** — для формирования встраиваемых образов QNX и загрузки их на целевые ЭВМ;
- **QNX System Profiler** — для визуализации трассы событий ядра, построенной с помощью пакета System Analysis Toolkit;
- **QNX System Information** — для мониторинга процессов, выполняющихся на целевых системах.

По сути дела, перспектива представляет собой файл в XML-формате, описывающий используемые редакторы и представления, а также их размещение на "рабочем месте". Поэтому вы можете разместить любые имеющиеся редакторы и представления так, как вам нравится, и объявить полученную конфигурацию рабочего места как новую перспективу. Свои перспективы можно удалять, стандартные — нельзя.

На рис. 12.1 представлена перспектива **Debug**.

Для "рабочего места" можно задать различные настройки, например:

- **Perform build automatically on resource modification** — выполнять сборку проекта при сохранении изменений в каком-либо ресурсе проекта;
- **Save all modified resources automatically prior to manual build** — сохранять измененные ресурсы при запуске сборки проекта вручную;
- **Link Navigator selection to active editor** — активизировать редактор ресурса при выделении имени ресурса в представлении **Navigator**. И наоборот — при активизации редактора с открытым ресурсом автоматически выделять имя ресурса в навигаторе.

Можно задавать массу различных параметров: способ открытия новых перспектив и проектов, размещение вкладок редакторов и представлений, параметры сравнения разных версий одого файла, параметры хранения истории изменения ресурсов и т. п.

Кроме того, можно настраивать различные аспекты поведения инструментов, например зависимости между проектами, генерировать ли отладочные версии файлов и т. д.

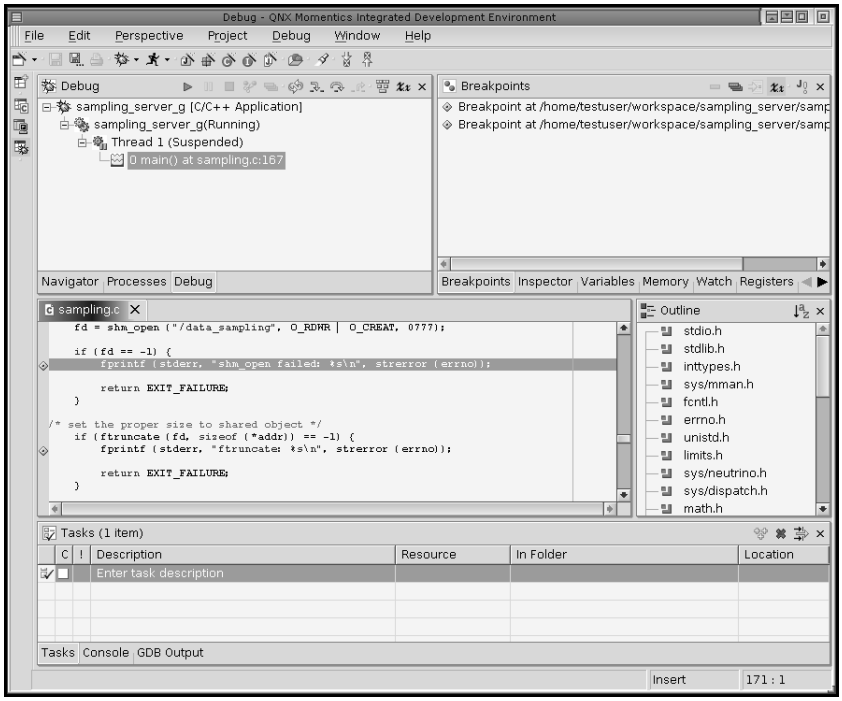


Рис. 12.1. Перспектива **Debug**

Средства визуального моделирования

Мне известно только одно средство визуального моделирования для QNX — Rational Rose RealTime (или просто Rose). Хотя среда разработки Rose существует для нескольких ОС, для целевых систем QNX моделирование поддерживается только в среде Windows. Для поддержки QNX дополнительно к самой среде Rational Rose RealTime необходимо установить продукт Rational Rose Adaptation Layer (это обычный самораспаковывающийся архив). Пользователи профессионального дистрибутива Momentics могут скачать Rational Rose Adaptation Layer из раздела myQNX

на сайте **www.qnx.com**, зарегистрировав свою копию QNX Momentics PE.

Средства разработки графических интерфейсов пользователя

В QNX используют два инструмента визуальной разработки графических интерфейсов.

- Photon Application Builder (PhAB) — штатный инструмент, поставляемый в любом дистрибутиве Momentics, в том числе, в NC. Это очень простое и удобное в использовании средство. Оно не столь богато возможностями в сравнении с аналогичными продуктами для Windows, но для большинства задач его вполне достаточно. Некоторые разработчики создают весьма сложные GUI-приложения.
- Tilcon RealTime Developer (TRTD) — продукт компании Tilcon. Пожалуй, главное достоинство TRTD — богатый набор готовых виджетов.

TRTD состоит из двух частей:

- среда разработки, работающая под Windows;
- графические "ядра" — своего рода виртуальные машины — для разных операционных систем, в том числе для QNX4 и QNX6.

Приложение TRTD создается в среде разработки, а потом может быть запущено в любой операционной системе, для которой есть графическое ядро. Его недостатки традиционны для виртуальных машин: в первую очередь — высокая ресурсоемкость.

Средства управления версиями

Для управления версиями в любом дистрибутиве QNX можно использовать CVS (Concurrent Version control System) — стандартный инструмент Linux-проектов. В QNX IDE имеется достаточно функциональный графический клиент для CVS.

QNX IDE может работать в качестве клиента самой популярной коммерческой системы управления версиями для крупных проектов — Rational ClearCase.

Идея любой системы управления параллельными версиями проста: имеется центральный сервер с архивной копией проекта, называемой *репозитарием*. Каждый программист команды, работающей над проектом, получает экземпляр проекта, называемый *рабочей копией*. После внесения изменений в любой файл проекта разработчик может внести эти изменения в центральный репозитарий. Изменения получают свой номер и можно всегда посмотреть, какие и когда были внесены изменения, можно в любой момент вернуться к любой предыдущей версии, можно от любой версии сделать отдельную ветвь проекта. Другими словами, система управления версиями — чрезвычайно полезный инструмент даже при индивидуальной работе, не говоря о командной.

QNX как CVS-сервер

Если требуется создать CVS-сервер под QNX, то следует выполнить ряд действий:

1. Задать переменную `CVSROOT`, указав в ней имя каталога для репозитария, и переменную `EDITOR`, указав в ней имя вашего любимого редактора (если вы используете **`vi`**, то переменная `EDITOR` не нужна):

   ```
   mkdir /var/mycvs
   export CVSROOT=/var/mycvs
   export EDITOR=ped
   ```

2. Затем нужно создать пустой репозитарий:

   ```
   cvs init
   ```

   При этом в каталоге `/var/mycvs` будет создан служебный каталог репозитария `CVSROOT`.

3. Для импортирования в репозитарий, например, каталога `/home/myname/myproject`, нужно войти в этот каталог и выполнить команду импортирования:

   ```
   cd /home/myname/myproject
   cvs import myproject swd init
   ```

   где `myproject` — имя проекта в репозитарии, `swd` — тег производителя (произвольный), `init` — тег версии (произвольный).

   На терминале запустится редактор, указанный в переменной `EDITOR` (или **`vi`**, если переменная не задана), чтобы вы могли

ввести комментарий к своим действиям (CVS всегда перед тем, как что-нибудь сделать, предлагает ввести комментарий). Действие выполняется после того, как вы сохраните комментарий и закроете редактор.

4. Для удобства создадим пользователя `cvsuser`, принадлежащего группе, скажем, `cvs`. Укажем для `cvsuser` вместо `/bin/sh` — `/dev/null` (лишняя защита от взлома еще никому не мешала). Поскольку каталог `/var/mycvs` уже существует, утилита **`passwd`** не захочет делать его домашним для пользователя `cvsuser` (сообщит, что нельзя разделять между пользователями каталоги и т. п.). Ничего страшного, пусть утилита **`passwd`** создает каталог `/home/cvsuser` — после добавления пользователя мы его уничтожим и вручную откорректируем файл `/etc/passwd` (вместо `/home/cvsuser` напишем `/var/mycvs`).

5. Теперь добавим CVS-пользователей. Регистрировать каждого такого пользователя в системе — непрактично. Проще настроить отображение CVS-пользователей на какого-нибудь (или каких-нибудь) QNX-пользователей, например, `cvsuser`. Для этого в папке `/var/mycvs/CVSROOT` создадим файл `passwd`. Формат файла прост, одна строка соответствует одному CVS-пользователю:

   Имя_пользователя_CVS:*Зашифр_пароль*:*имя_пользователя_QNX*

6. Заведем CVS-пользователей `myself` и `basily` без пароля:

   ```
   myself::cvsuser
   basily::cvsuser
   ```

   Просто, да? Единственная тонкость — пароли. Их нужно задавать в зашифрованном виде, по аналогии с `/etc/shadow`. Алгоритм используется тот же — стандартная функция *crypt()*. Можно написать свою утилиту, использующую функцию *crypt()*, но мне, например, лень. Проще всего использовать учетную запись пользователя `cvsuser` ☺ (все равно это "учебный" пользователь с `/dev/null` вместо login shell). Другими словами, задаете любой пароль для `cvsuser` и копируете его зашифрованное представление из `/etc/shadow` в `CVSROOT/passwd` между двоеточиями. Для каждого CVS-пользователя, как видно из формата файла, можно задать свой пароль.

7. Да, и не забудьте на самом деле сделать пользователя `cvsuser` владельцем каталога `/var/mycvs` и всего содержимого этого каталога:

   ```
   chown -R cvsuser:cvs /var/mycvs
   ```

8. Теперь настроим запуск CVS-сервера. Добавим в файл `/etc/services` строку:

   ```
   cvspserver 2401/tcp
   ```

   А в файл `/etc/inetd.conf` строку:

   ```
   cvspserver stream tcp nowait root /usr/bin/cvs cvs --allow-
   root=/var/mycvs pserver
   ```

   Заметим, что опций `-allow-root` может быть несколько.

9. Теперь можно запускать суперсервер TCP/IP **`inetd`**, который сам запустит **`cvs`** при поступлении запроса. Только для корректной работы **`cvs`** процесс **`inetd`** не должен заполучить переменную окружения `HOME` (здесь нет смысла, это баг), поэтому делаем так:

   ```
   unset HOME
   inetd
   ```

QNX как CVS-клиент

Теперь что касается клиента CVS. Клиент для работы также оперирует переменной `CVSROOT`. Значение переменной задается в таком формате:

`:`*`Метод_доступа`*`:`*`Пользователь`*`@`*`Хост`*`:`*`Репозитарий`*

Например:

```
export CVSROOT=:pserver:myself@192.168.3.184:/var/mycvs
```

При работе в QNX IDE задавать переменную не требуется — эти параметры там вводятся в графическом режиме. Вообще, в IDE с CVS работать, конечно, удобно — все действия можно выполнять щелчком мыши (не надо помнить командно-строковых директив) и результат получаем в графическом виде. Например, так выглядит окно сравнения модифицированного файла с одной из его предыдущих версий (рис. 12.2).

Сразу видно — что удалено, что добавлено, что изменено. Чтобы иметь этот инструмент, вовсе не обязательно покупать

Momentics PE — это штатные возможности базового движка Eclipse, доступного для скачивания на **www.eclipse.org**.

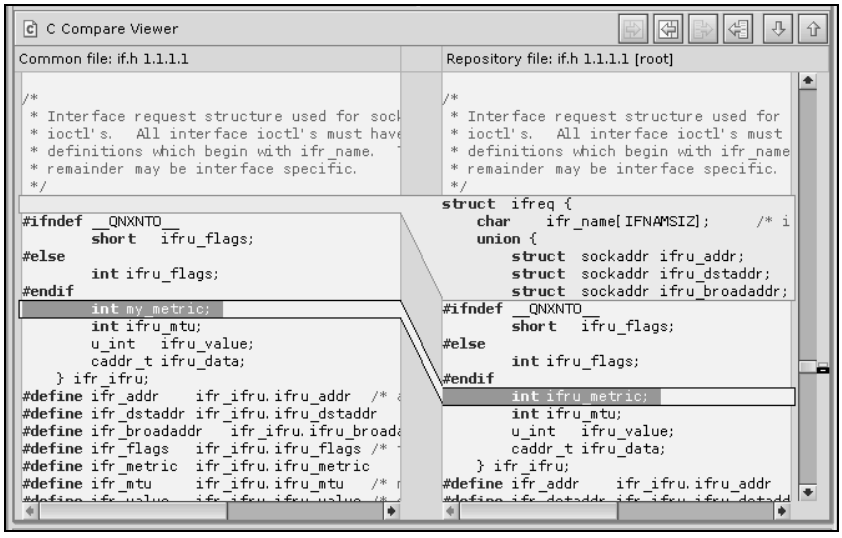
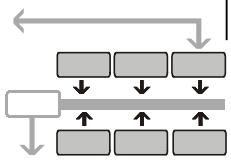


Рис. 12.2. Сравнение версий файла

В Интернете достаточно много документации по CVS. Например, есть русскоязычный сайт с URL — не поверите! — **www.cvs.ru**.



Глава 13

Построение целевых систем QNX

QNX — встраиваемая операционная система. То есть архитектура ОС позволяет разработчикам ПО с помощью специального инструментария формировать специализированные дистрибутивы QNX, "заточенные" для решения конкретной прикладной задачи в рамках ограниченных ресурсов. Как создаются встраиваемые конфигурации QNX — мы и обсудим в этой главе, при этом будут рассмотрены вопросы:

- начальная загрузка системы разработки;
- построение загрузочного образа;
- загрузка образа на целевой ЭВМ.

Процесс начальной загрузки

Чтобы не изобретать велосипед, рассмотрим процесс начальной загрузки на примере той целевой системы производства компании QSS, которая является нашей системой разработки.

После включения питания мультизагрузчик сначала предлагает выбрать, какую ОС следует загрузить. Вернее, он предлагает выбрать раздел диска, с которого следует выполнять загрузку. На моей машине по умолчанию грузится QNX, это выглядит так:

```
Press F1-F4 for select drive or select partition 1,2,3,4? 1
```

(Нажмите клавишу <F1>, <F2>, <F3> или <F4> для выбора дисковода или выберите раздел 1, 2, 3 или 4? 1)

После выбора раздела QNX начинается процесс начальной загрузки ОС. Начальная загрузка QNX протекает поэтапно.

1. Выполняется код начального загрузчика (IPL — Initial Program Loader), который осуществляет минимальное конфигурирование аппаратуры (конфигурирует контроллер памяти, системные часы и т. п.) и затем выполняет поиск на носителе и загрузку в ОЗУ образа ОС и передачу ему управления. IPL бывают:
   - "Warm-start" — для систем с BIOS (на момент старта IPL часть оборудования уже проинициализирована BIOS);
   - "Cold-start" — для систем без BIOS (всю инициализацию оборудования должен выполнять сам IPL).

   Последнее действие IPL — запуск модуля **`startup`**.
2. Выполняется код загрузчика ОС (**`startup`**), входящий в состав загрузочного образа QNX. Загрузчик **`startup`** копирует и (если нужно) разжимает загрузочный образ QNX, определяет состав и конфигурацию аппаратуры, заполняет системную страницу данных ОС и передает управление модулю **`procnto`**, который обязательно входит в состав загрузочного образа QNX.
3. Модуль **`procnto`** выполняет необходимую инициализацию и запускает так называемый *стартовый сценарий*, входящий в состав загрузочного образа.
4. Стартовый сценарий (назовем его `Boot Script`) запускает с необходимыми параметрами программы и сценарии, входящие в состав образа. Без этого система делать ничего не будет — модуль **`procnto`**, как вы помните, лишь обеспечивает выполнение других программ.

Следует заметить, что в QNX обычно используется два загрузочных образа: основной образ помещается в файл `/.boot`, а резервный — в файл `/.altboot`. Вторичный загрузчик **`startup`** предлагает нажать клавишу <Esc> для загрузки резервного образа:

```
Hit Esc for .altboot
```

Если ничего не нажимать, то загружаться будет основной образ `/.boot`. Одним из процессов, входящих в стандартный образ, является **`diskboot`**. Основное назначение этого процесса — поиск базовых образов файловой системы QNX на всех доступных дисках и запуск командного сценария `/etc/system/sysinit`.

Примечание

Раздел диска с файловой системой QNX может быть смонтирован как корневой, только если в нем есть файл `/.diskroot`.

Прежде чем приступить к работе, утилита **diskboot** предлагает нажать клавишу <пробел> для ввода опций загрузки:

```
Press the space bar to input boot options
```

Если ничего не нажимать, загрузка продолжится в автоматическом режиме. Опции загрузки, выставляемые вручную, мы обсудим позже.

Итак, если ничего не нажимать, появится сообщение о сканировании аппаратуры, затем, если утилита **diskboot** нашла больше одного раздела с файлами `/.diskroot`, то будет предложено выбрать, какой из этих разделов будет корневым:

```
You have more then one .diskroot file which wants to mount at /
F1   /dev/hd0t78
F2   /dev/hd0t79
Which one do you whish to mount?
```

На этой ЭВМ в разделе `/dev/hd0t78` установлена ОС QNX Neutrino 6.2.1A, и утилита **biskboot** это определила. Нажимаем клавишу <F2>. Раздел `/dev/hd0t79` смонтируется как `/`, но и раздел `/dev/hd0t78` смонтируется куда-нибудь в каталог `/fs`. Это значит, что лежащий в 78-м разделе файл базовой системы QNX 6.2.1A все равно еще может быть смонтирован в каталог `/pkgs/base` для отображения пакетной файловой системой на `/` вместо файла базовой системы QNX 6.3. Поэтому **diskboot** выведет на экран новое приглашение:

```
You have more then one .diskroot file which wants to mount at
/pkgs/base
F1   /fs/hd0-qnx4-2/boot/fs/qnxbase.qfs
F2   /boot/fs/qnxbase.qfs
Which one do you whish to mount?
```

Если нажать клавишу <F2>, то в каталог `/fpgs/base` будет смонтирован файл `/boot/fs/qnxbase.qfs`. Последним действием утилиты **diskboot** является вызов сценария `/etc/system/sysinit`, запускающего ряд других сценариев из каталога `/etc/rc.d/`.

В конце цепочки стартует утилита **login** или, если при инсталляции было указано запускать графическую среду Photon автоматически, утилита **phlogin**. В любом случае понадобится ввести имя и пароль.

Если в ответ на приглашение:

```
Press the space bar to input boot options
```

нажать клавишу <пробел>, то на экране появится сообщение:

```
F1      Safe modes
F5      Start a debug shell after mounting filesystems
F6      Be Verbose
F7      Mount read-only partitions read/write if possible
F8      Enable a previous package configuration
F9      Target output to debug device defined in startup
code
F10     Force a partition install
F11     Enumerator disables
F12     Driver disables
Enter   Continue boot process
Please select one or more options via functional keys.
Selection?
```

(<F1> — загружаться в "безопасном режиме";

<F5> — запустить командный интерпретатор после подключения файловых систем;

<F6> — установить режим подробного вывода диагностических сообщений;

<F7> — попытаться подключить с возможностью записи разделы, доступные только для чтения;

<F8> — вернуться к предыдущей конфигурации пакетов;

<F9> — выводить диагностическую информацию на устройство, указанное в модуле `startup`;

<F10> — приступить сразу к инсталляции системы;

<F11> — отключить автоматическое распознавание устройств;

<F12> — отключить драйверы;

<Enter> — продолжить загрузку.

Выберите, пожалуйста, одну или более опций, нажав соответствующие функциональные клавиши.

Ваш выбор?)

При нажатии клавиши <F1> утилита **diskboot** предложит несколько вариантов загрузки системы с ограниченной функциональностью.

Опция <F10> доступна только при загрузке с инсталляционного компакт-диска. Без этой опции утилита **diskboot**, кроме инсталляции, предложит просто загрузить систему с одного из обнаруженных разделов QNX (на компакт-диске или в разделе жесткого диска).

При выборе опции <F11> утилита **diskboot** предложит выбрать, автоопределители каких типов устройств отключать, а при выборе <F12> — какие группы драйверов отключать.

Командный сценарий *sysinit*

Последнее действие процесса **diskboot** перед завершением — запуск командного сценария `/etc/system/sysinit`. Задача этого сценария — запустить процессы, обеспечивающие необходимую функциональность ОС. Давайте посмотрим, что же делает этот командный файл.

1. Запускает сервис регистрации системных событий (если он еще не запущен) **slogger**.
2. Запускает администратор неименованных каналов pipe.
3. В случае если это первый запуск системы после инсталляции, то запускает сценарий `/etc/rc.d/rc.setup-once`. Факт первого запуска устанавливается по отсутствию файла `/etc/system/package/packages`.
4. Устанавливает часовой пояс; информация берется из файла `/etc/TIMEZONE`.
5. Запускает командный сценарий `/etc/rc.d/rc.rtc`, если таковой существует (по умолчанию — отсутствует), для настройки часов реального времени.
6. Определяет имя ЭВМ; информация берется из файла `/etc/HOSTNAME`.
7. Запускает командный сценарий `/etc/rc.d/rc.devices` (если таковой существует). Этот скрипт инициирует распознавание аппаратных устройств.

8. Если существует файл `/etc/system/config/useqnet` и запущен администратор сетевого ввода/вывода **io-net**, то загружает администратор сетевого протокола Qnet. Факт работы **io-net** определяется по наличию регистрируемого этим администратором префикса — каталога `/dev/io-net`. Администратор протокола Qnet реализован в виде DLL, расширяющей функциональность администратора **io-net**. Подключение выполняется следующей командой:

   ```
   mount -Tio-net npm-qnet.so
   ```

9. Если существует файл `/.swapfile`, то подключает его в качестве устройства своппинга.
10. Запускает командный сценарий `/etc/rc.d/rc.sysinit` (если таковой существует). Этот скрипт продолжает инициализацию системы.
11. Если не удается запустить `rc.sysinit`, то делает попытку запустить командный интерпретатор Korn Shell в интерактивном режиме. Если стандартный интерпретатор не может запуститься, делается попытка запустить интерпретатор с меньшими требованиями к ресурсам — Fat Embedded Shell (**fesh**).

Таким образом, сценарий `sysinit` перед окончанием своего выполнения вызывает сценарий `rc.sysinit`.

Командный сценарий *rc.setup-once*

Этот командный сценарий вызывается из скрипта `/etc/system/sysinit` только один раз — при первом запуске системы. Он выполняет различную подготовительну работу, например создание ряда каталогов, файла для "ручного" своппинга `/.swapfile`, файла начальной конфигурации пакетной файловой системы `/etc/system/package/packages` и т. д.

Командный сценарий *rc.devices*

Командный файл `/etc/rc.d/rc.devices` вызывается при каждой загрузке QNX из сценария `/etc/system/sysinit`.

Этот сценарий запускает администратор псевдотерминалов **devc-pty**, затем определяет каталоги, содержащие информацию

о поддерживаемых устройствах (для локального узла — `/etc/system/enum`). После этого запускается администратор конфигурирования аппаратуры **`enum-devices`**, сканирующий подключенные устройства.

Командный сценарий *rc.sysinit*

Командный сценарий `/etc/rc.d/rc.sysinit` вызывается при каждой загрузке QNX из сценария `/etc/system/sysinit` и предназначен для выполнения настроек, специфичных для данной ЭВМ, и запуска необходимых сервисов.

Этот сценарий запускает процесс **`dumper`**, сохраняющий "посмертные" core-файлы процессов, завершившихся аварийно. Затем запускает сценарий `/etc/rc.d/rc.local`, если таковой существует. Этот сценарий нужен, если вы хотите добавить свои команды инициализации, не редактируя созданные системой файлы. На моей рабочей станции, например, из сценария `rc.local` запускается процесс **`inetd`**, принимающий запросы к TCP/IP-сервисам.

Последнее действие сценария `rc.sysinit` — запуск программы инициализации терминала **`tinit`**. Эта программа запускает на терминале утилиту входа в систему **`login`** или графическую оболочку Photon с графической утилитой входа в систему **`phlogin`**. Какой вариант использовать, сценарий определяет по наличию или отсутствию файла `/etc/system/config/nophoton`.

Если запустить **`tinit`** не удалось, сценарий пытается последовательно запустить командные интерпретаторы **`ksh`** (**`sh`** — это просто ссылка на **`ksh`**) и **`fesh`**.

Построение загрузочного образа QNX Neutrino

В этом разделе кроме общих сведений рассматриваются:

- секция `boot`;
- файловая система Image;
- сценарий `Boot Script`;
- перспектива QNX System Builder.

Общие сведения

Итак, последнее действие IPL — запуск модуля **startup**. Главный результат работы **startup**-модуля — загружен в память образ и управление передано модулю **procnto**. Что же представляет из себя образ?

QNX-образы бывают двух типов:

- образ ОС (загружаемый, т. е. содержащий **startup**-модуль, или незагружаемый[1]);
- образ встроенной файловой системы (файловой системы Flash).

Образ ОС — это файл, содержащий модуль **procnto** приложения и некоторые данные, необходимые для функционирования целевой системы. Образ ОС можно рассматривать как "маленькую файловую систему", т. к. в образе есть простое дерево каталогов, из которого **procnto** берет информацию о содержащихся в образе файлах, их месте размещения в образе (эта файловая система получила название Image, мы подробно обсудим ее в этой главе).

С точки зрения работающей системы, образ ОС можно рассматривать как файловую систему с доступом по чтению. Содержимое образа можно посмотреть с помощью утилиты **dumpifs**. Посмотрим содержимое образа на нашей инструментальной машине:

```
dumpifs /.boot
```

Как образ ОС, так и образ файловой системы Flash могут содержать сжатые данные — при создании образа эти данные компрессируются (сжимаются), при последующих операциях чтения — декомпрессируются (разжимаются) автоматически. Следует отметить, что использование механизма компрессии/декомпрессии замедляет доступ к данным и создает дополнительную нагрузку на CPU целевой системы. Поэтому использовать компрессию рекомендуется только для систем с действительно ограниченным объемом носителя данных (Flash, ROM).

Образ ОС создается программой **mkifs** (**M**a**K**e **I**mage **F**ile**S**ystem). Данные для своей работы эта утилита берет из командной строки и из *файла построения*, или *build-файла* (buildfile).

---

[1] Незагружаемые образы ОС в настоящее время не поддерживаются.

Для того чтобы создать загрузочный образ, следует написать в текстовом редакторе или сгенерировать с помощью IDE-перспективы QNX System Builder три части build-файла:

- секцию `boot`, определяющую параметры модулей **`startup`** и **`procnto`**;
- сценарий `Boot Script`;
- остальную часть, определяющую содержимое файловой системы Image.

Чтобы не изобретать велосипед, рассмотрим build-файл нашей системы разработки — `/boot/build/qnxbasedma.build`. Основные системные элементы для образа находятся в каталоге `/boot/sys`.

Секция *boot*

Рассмотрим секцию `boot` (комментарии удалены):

```
[virtual=x86,bios +compress] boot = {
   startup-bios -s64k
   PATH=/proc/boot:/bin:/usr/bin
     LD_LIBRARY_PATH=/proc/boot:/lib:/usr/lib:/lib/dll
     procnto-instr
}
```

Означает это следующее:

- `virtual=x86` — собираем образ для *x*86-совместимой целевой системы с полной защитой памяти (для *x*86 других вариантов, собственно, нет);
- `+compress` — использовать сжатие образа. Этот флаг уменьшает размер образа, но увеличивает время начальной загрузки (**`startup`**-модулю придется разжимать образ);
- `startup-bios -s64k` — определяем, какой из **`startup`**-модулей будем использовать, задаем ему опции запуска;
- `PATH=`*`пути_к_программам`* `LD_LIBRARY_PATH=`*`пути_к_DLL`* `procnto-instr` (одной строкой!) — указываем, какой из модулей **`procnto`** включаем в образ. В данном случае опции не указаны, но их можно задавать. Модуль **`procnto`** будет запущен

с указанными значениями переменных окружения PATH и LD_LIBRARY_PATH.

Файловая система Image

За секцией boot следует описание файловой системы Image. Утилита **mkifs** берет указанные файлы из системы разработчика, однако содержимое текстовых файлов можно явно определить прямо в build-файле. Яркий пример — файл сценария Boot Script (про него поговорим отдельно). Итак, посмотрим снова на наш build-файл. В нем перечислены программные модули:

```
libc.so
libcam.so
io-blk.so
cam-disk.so
fs-qnx4.so
fs-dos.so
fs-ext2.so
cam-cdrom.so
fs-cd.so
[data=uip]
seedres
pci-bios
devb-eide
devb-amd
devb-aha2
devb-aha4
devb-aha7
devb-aha8
devb-ncr8
diskboot
slogger
fesh
devc-con
```

Другими словами, в этой части перечисляем все, что собираемся включать в состав образа. Советую не увлекаться и обратить

внимание на то, что в рассматриваемом примере бо́льшая часть модулей образа — драйверы устройств и администраторы файловых систем.

Видно, что перед перечислением исполняемых модулей указан атрибут `[data=uip]`. Дело в том, что поскольку файловая система Image после загрузки находится непосредственно в ОЗУ, при запуске приложений можно поступить двояко: использовать уже загруженный код непосредственно из того места ОЗУ, где он находится (`uip` — Use In Place), либо сделать, так сказать, рабочую копию (`copy`). Причем поведение можно определить отдельно для сегмента кода (`code`) и для сегмента данных (`data`).

Все заданные файлы после загрузки образа на целевой системе по умолчанию помещаются в каталог `/proc/boot`. Если нужно какой-то файл поместить в другое место, то следует указать полное путевое имя этого файла таким, каким оно должно быть на целевой системе.

После перечисления файлов в нашем примере есть такой фрагмент:

```
unlink_list={
/proc/boot/devb-*
}
```

Этот пример наглядно показывает, как можно включать в образ произвольные текстовые файлы. Имя файла — `unlink_list`, между фигурными скобками — содержимое. Этот текстовой файл, как легко догадаться, на целевой системе будет называться `/proc/boot/unlink_list`.

Примечание

Файл `unlink_list` предназначен только для информационных целей. Читатель, конечно, понял, что драйверы в том количестве, в каком они включены в образ, реально не требуются. Поэтому программа **diskboot** (см. о ней далее) удаляет драйверы из ОЗУ после монтирования дисковой файловой системы. Операция удаления жестко задана в программе **diskboot**, а чтобы пользователь знал о том, *что* удалено, разработчики добавили в образ файл `unlink_list`.

Сценарий *Boot Script*

От написания сценариев не освобождает даже QNX IDE, поэтому рассмотрим их чуть подробнее. У `Boot Script` есть полезные свойства:

- он выполняется после того, как администратор процессов завершил свою инициализацию;
- на момент запуска сценария на целевой системе в нашем распоряжении будет файловая система Image со всеми компонентами, которые мы задали;
- из него можно запускать другие сценарии, содержащиеся в образе.

Посмотрим на сценарий из нашего примера:

```
[+script] startup-script = {
  procmgr_symlink ../../proc/boot/libc.so.2
    /usr/lib/ldqnx.so.2
  [pri=10o] PATH=/proc/boot diskboot -b1 -D1 -odevc-con,-n4
}
```

Сценарий называется `startup-script`, но это, строго говоря, неважно — утилита **`mkifs`** узнает его по атрибуту `[+script]`. В среде разработки, как мы видим, сценарий очень прост: сначала создается ссылка `/usr/lib/ldqnx.so.2` на файл `/proc/boot/libc.so.2`, затем запускается "умная" утилита **`diskboot`**, которая доводит загрузку до победного конца.

Примечание

Конечно, никто не запрещает нам загружать целевые системы утилитой **`diskboot`**, но это нерационально. Эта утилита тратит много времени на всевозможные проверки и сканирование и нужна для загрузки ЭВМ с неизвестным набором аппаратуры, поддерживаемой в QNX. По сути дела, в своей сетевой системе вам нужно вручную получить тот же результат, что и у **`diskboot`**. Но это на самом деле просто — спецификация целевой ЭВМ вам известна.

Конечно, возможностей у нас меньше, чем при использовании Korn Shell, т. к. не будем же мы помещать в образ разные утилиты — ОЗУ не безразмерно, — но, тем не менее, есть несколько

встроенных команд, которые всегда можно использовать в стартовом сценарии:

- `display_msg` — выводит указанный текст, например:
  ```
  display_msg Hello!
  ```
- `procmgr_symlink` — создает "виртуальную" символьную ссылку аналогично команде **`ln -P`**, за исключением того, что не требуется утилита **`ln`** (это мы уже видели в примере).
- `reopen` — связывает потоки **`stdin`**, **`stdout`** и **`stderr`** с указанным файлом. Например:
  ```
  reopen /dev/con1
  ```
- `waitfor` — ожидает момента, когда вызов функции *stat()* по указанному путевому имени завершится успешно (т. е. пока не будет зарегистрирован соответствующий префикс). Например:
  ```
  waitfor /dev/pci
  ```

Перед командами сценария можно помещать атрибуты. Есть два типа атрибутов:

- логический атрибут (Boolean) ("+" — `true`, "-" — `false`). Например:
  ```
  [+session] ksh
  ```
  (процесс **`ksh`** будет лидером сеанса);
- атрибут-значение (Value). Например:
  ```
  [pri=27f] ksh
  ```
  (процесс **`ksh`** будет запущен с приоритетом 27 и дисциплиной диспетчеризации FIFO).

Атрибуты можно комбинировать:

```
[+session pri=27f] ksh
```

Примечание

Обратите внимание, что все скомбинированные атрибуты помещаются в одних квадратных скобках.

Если необходимо использовать программы с такими же именами, как у встроенных команд, то для этих программ следует задавать атрибут `[+external]`. Например, запустим гипотетическую утилиту **`reopen`** с опцией **`-f`**:

```
[+external] reopen -f
```

Для наглядности посмотрим на фрагмент настоящего сценария, более содержательного, чем используемый утилитой **diskboot**:

```
display_msg Starting serial driver
devc-ser8250 -e -b115200 &
waitfor /dev/ser1 # don't continue until /dev/ser1 exists

display_msg Starting pseudo-tty driver
devc-pty &

display_msg Setting up consoles
devc-con &
reopen /dev/con2  # set stdin, stdout and stderr to /dev/con2
    [+session pri=27r] PATH=/proc/boot fesh &
reopen /dev/con1  # set stdin, stdout and stderr to /dev/con1
    [+session pri=10r] PATH=/proc/boot fesh &
```

Для выполнения такого сценария в образе должны быть модули **devc-ser8250**, **devc-pty**, **devc-con** и **fesh**.

Перспектива QNX System Builder

Внешний вид перспективы QNX System Builder в последней имеющейся у меня версии IDE показан на рис. 13.1.

Перспектива QNX System Bulder предоставляет такие возможности:

- конфигурирование секции `boot` в графическом виде;
- добавление/удаление модулей в образ в графическом режиме (при этом мы сразу видим, как будет выглядеть файловая система Image на целевой системе). Разделяемая библиотека `libc.so` добавляется автоматически с установлением символической ссылки на нее `/usr/lib/ldqnx.so`;
- мастер оптимизации System Optimizer (для его запуска надо нажать кнопку). Он может выполнять три вида оптимизации:
  - удалить неиспользуемые библиотеки, включенные нами в образ;
  - добавить необходимые библиотеки, которые мы забыли включить в образ;

- удалить из библиотек для сокращения размеров образа те функции, которые не используются (другие инструменты этого не умеют);

❒ информация о каждом элементе образа отображается в окне свойств **Properties** и в окне атрибутов **Item Attributes**.

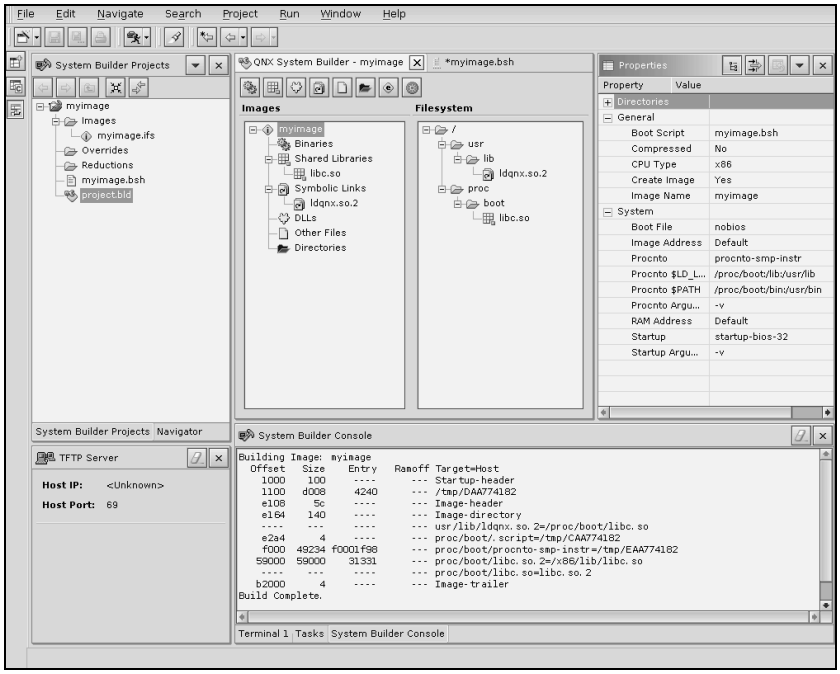


Рис. 13.1. Перспектива QNX System Builder

Сгенерированный IDE образ имеет расширение `ifs` и помещается в каталог *`$HOME:/workspace/имя_проекта/Images`*.

Загрузка образа на целевой ЭВМ

Каким образом можно загрузить образ на целевой ЭВМ? Есть несколько способов.

❒ Скопировать образ в файл `/.boot` или `/.altboot` загружаемого раздела QNX4 на целевой системе. Это самый простой способ (конечно, при наличии диска у целевой ЭВМ).

Поместить образ в ПЗУ целевой системы. Обычно для этой цели используют Flash-память. Такое решение — часто единственно возможное для целевых систем, работающих в жестких условиях эксплуатации (например, при тряске) и требующих очень быстрой загрузки операционной системы.

- Загрузить образ по сети с TFTP-сервера. Для этого на сетевой карте должна быть микросхема ПЗУ, содержащая расширение BIOS, поддерживающее протоколы Internet Boot Protocol и TFTP (Trivial File Transfer Protocol). Эту микросхему ПЗУ называют *bootp ROM*.
- Загрузить образ по сети с узла QNX4. Для этого на сетевой карте должна быть специальная микросхема ПЗУ, содержащая расширение BIOS, поддерживающее протокол QNX Boot Protocol. Эту микросхему ПЗУ называют *QNX boot ROM*.
- Загрузиться через последовательный канал. Для этого на сервере загрузки используется утилита **sendnto**, на целевой системе для получения такого образа требуется специальный IPL.

Загрузка целевой системы с узла QNX4

При включении питания целевой системы запускается код, содержащийся в QNX boot ROM на сетевой карте. Он шлет широковещательный ethernet-запрос, разыскивая образ. Этот запрос содержит MAC-адрес Ethernet-карты целевой системы.

Процесс **netboot** на узле QNX4 (узел QNX4 должен находиться в той же подсети, что и целевая система) просматривает каждый ethernet-кадр на предмет наличия запроса на загрузку от QNX boot ROM.

Как только запрос получен, **netboot** ищет в файле `/etc/config/netmap` по полученному MAC-адресу *логический номер узла* (logical node number). Затем по логическому номеру узла отыскивается нужная запись в файле `/etc/config/netboot`. В соответствии с этой записью **netboot** посылает образ на QNX boot ROM. QNX boot ROM загружает образ в ОЗУ и запускает его.

Примечание

Подробную информацию об утилите **netboot** можно найти в документации ОСРВ QNX4.*xx*.

Загрузка целевой системы с TFTP-сервера

Этот способ удобнее, чем загрузка с QNX4, по нескольким причинам:

- bootp ROM гораздо дешевле QNX boot ROM;
- в качестве сервера загрузки можно использовать ЭВМ с разными операционными системами;
- использование TCP/IP позволяет размещать сервер и целевую систему в разных подсетях.

При включении питания целевой системы запускается код, содержащийся в bootp ROM на сетевой карте. При этом выполнение проходит в две стадии.

1. Сетевая карта шлет DHCP-запрос. Ответ от DHCP-сервера (или BOOTP-сервера — без разницы, но DHCP более гибок) содержит IP-адрес целевой системы, IP-адрес TFTP-сервера и имя образа.
2. Сетевая карта шлет TFTP-запрос. В ответ получает файл образа, который она загружает и передает ему управление.

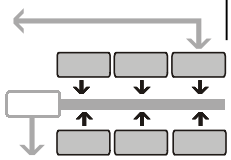
В QNX IDE есть встроенный TFTP-сервер, его главное достоинство — он сам "знает" где лежат образы, сгенерированные IDE (обычные TFTP-серверы умеют работать только с одним каталогом, помещать образы в который следует вручную).

Размещение образа в ПЗУ целевой системы

Есть несколько способов переноса образа QNX Neutrino в ПЗУ целевой системы, зависящих от применяемой аппаратуры.

- *Программирование образа с помощью ППЗУ-программатора* — в случае, если Flash-память не напаяна на процессорную плату (т. е. "не onboard"). При этом может понадобиться операция преобразования образа в формат, понятный программатору, с помощью утилиты **mkrec**.
- *Перенос образа в onboard Flash-память с помощью программатора через специальную шину (например, JTAG)* — возможно, также понадобится утилита **mkrec**.

- *Использование настройки BIOS для программирования образа через устройство ввода/вывода* — этот вариант возможен для некоторых процессорных плат, BIOS Setup которых позволяет войти в сервисный режим программирования onboard-носителей (например, Fastwel CPU686E). Устройством ввода/вывода может быть любое коммуникационное устройство, например, последовательный порт.
- *Программирование образа с помощью драйвера встроенной файловой системы* — этот вариант возможен только тогда, когда имеется драйвер для соответствующей микросхемы Flash. К тому же, скорее всего, потребуется установить связь между целевой и инструментальной системами по сети или другим способом (например, через последовательный порт). В составе QNX Momentics PE поставляется ряд драйверов Flash-устройств, но если подходящего среди них все-таки нет, то его можно разработать (самостоятельно или заказать).



Глава 14

Средства анализа целевых систем

В этой главе будут рассмотрены вопросы:

- трассировка событий ядра;
- журналы событий прикладного уровня;
- получение информации об оборудовании.

QNX — операционная система жесткого реального времени. Это значит, что она в состоянии обеспечить выполнение приложений в условиях критического лимита времени. Однако человеку свойственно ошибаться, и приложения не всегда ведут себя так, как ожидается. Поэтому у разработчика и у администратора обязательно должны быть эффективные инструменты анализа и диагностики.

Трассировка событий ядра

В состав QNX Momentics PE входит пакет System Analysis Toolkit (SAT). SAT позволяет отслеживать:

- вызовы микроядра;
- передачу сообщений;
- обработку прерываний;
- изменения состояний потоков.

Основу этого пакета составляет модуль `procnto-instr` — администратор процессов, с которым скомпоновано микроядро, оборудованное средствами диагностики (на самом деле модулей `instr` несколько — для разных архитектур, систем с SMP).

В операционной системе происходят различные события, все они так или иначе проходят через микроядро. Средства, входящие в инструментальное расширение, сохраняют трассу событий в буферах ядра в виде записей. Для сброса информации из буферов в файл предназначена утилита **`tracelogger`**. Для увеличения скорости этой операции трасса записывается в двоичном формате, поэтому для конвертирования данных в форму, удобную для чтения, следует или воспользоваться утилитой **`traceprinter`**, или написать свою программу обработки журнала трассировки (в составе SAT есть специальная библиотека `libtraceparser.a` — SAT API).

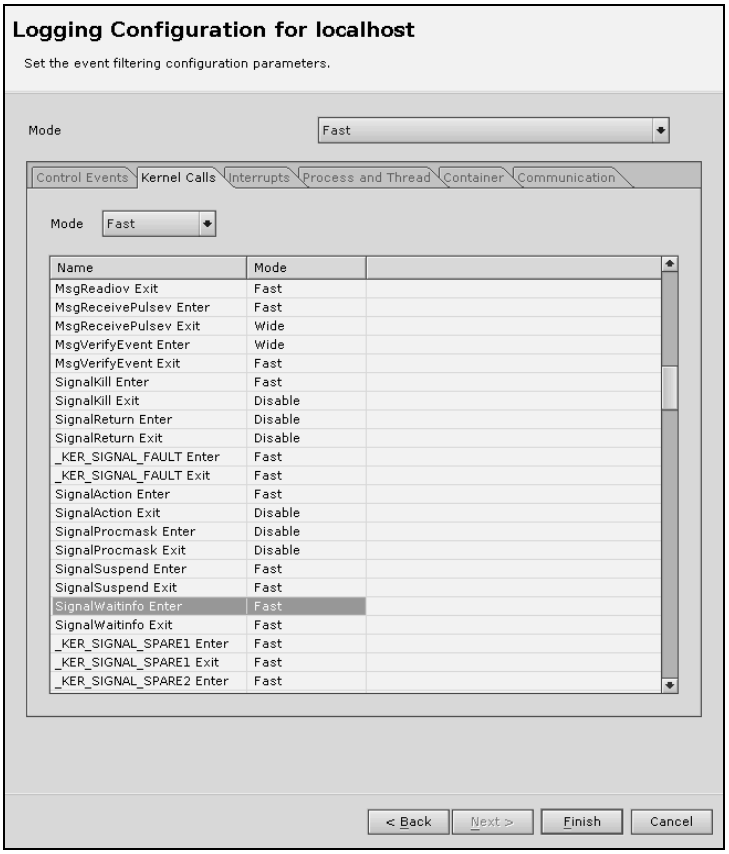


Рис. 14.1. Задание правил фильтрации

По умолчанию утилита **tracelogger** заказывает у администратора процессов 32 буфера. Каждый из буферов состоит из 1024 слотов по 16 байт (т. е. каждый буфер имеет размер 16 K). Событие ядра, для описания которого достаточно одного слота, называется *простым*. Те события, для описания которых недостаточно одного слота, называются *сложными*.

Для трассировки можно задать четыре вида фильтров.

- Режим fast или wide. Fast указывает, что сложные события должны обрезаться до размера простых. Это позволяет сэкономить слоты и увеличить скорость трассировки. Режим fast задан по умолчанию.
- Статический и динамический фильтры. Позволяют задавать правила записи событий в буферы, например, указывать, какие именно события нас интересуют. Динамический фильтр медленнее статического, но более гибок.
- Выходной фильтр. Не влияет на сбор трассы, а только позволяет извлекать нужную информацию из журнала трассировки.

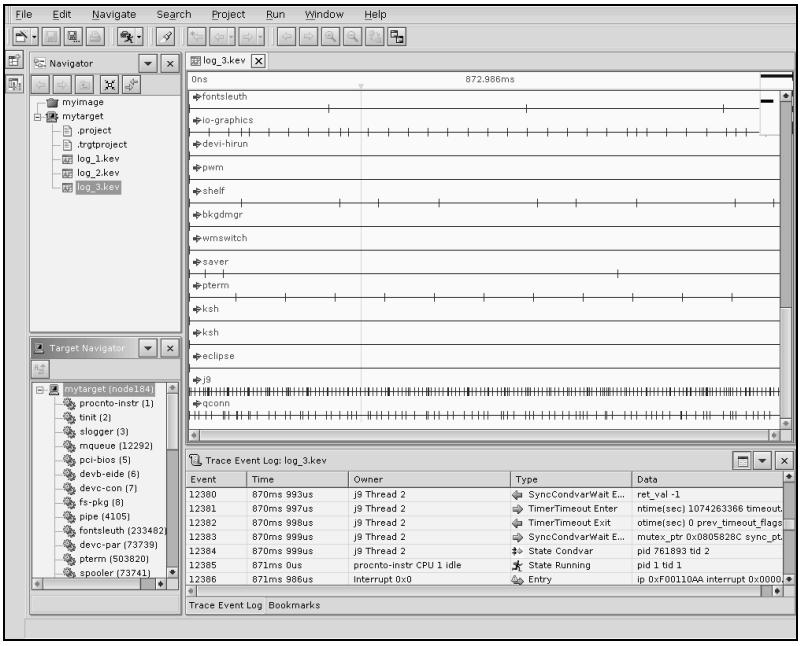


Рис. 14.2. Перспектива QNX System Profiler

Примеры создания фильтров есть в электронной документации по SAT.

С помощью QNX IDE выполнять анализ трассы ядра намного удобнее, т. к. информация о событиях представляется в графическом виде. Например, очень удобно задавать правила фильтрации (рис. 14.1).

Перспектива QNX System Profiler выглядит так, как показано на рис. 14.2.

Журналы событий прикладного уровня

В ОС QNX реализовано две системы ведения журналов событий:

- **syslog**
- **slogger**

Система регистрации событий *syslog*

Система **syslog** традиционно используется в различных UNIX-подобных ОС. В QNX она включена в основном для обеспечения переноса программных продуктов из других операционных систем. Основу системы **syslog** составляет серверный процесс **syslogd**, считывающий при старте конфигурационный файл `/etc/syslog.conf`. Информация для записи в журнал (например, в файл `/var/log/syslog` — имя задается в файле конфигурации) поступает от клиентов, вызывающих для этой цели соответствующие функции. Из командных сценариев информация может посылаться с помощью утилиты **logger**. Кроме журнального файла, **syslogd** может писать информацию непосредственно на экран или передавать для обработки процессу **syslogd**, запущенному на другом хосте сети. Какую же информацию записывает **syslogd** в журнал? Это:

- дата и время возникновения события;
- идентификатор узла сети, на котором произошло событие;
- имя/идентификатор процесса/пользователя, запросившего регистрацию события;
- текстовое сообщение (определяется разработчиком программы-клиента).

Система регистрации событий *slogger*

Штатная QNX-система ведения журналов событий **slogger** проще и эффективнее системы **syslog**. Основу системы составляет серверный процесс **slogger**, регистрирующий по умолчанию префикс `/dev/slog`. Этот файл устройства и является журналом событий. Процессы могут с помощью специальных функций посылать серверу **slogger** информацию для записи в журнал. Информация сохраняется в виде записей определенного формата:

- дата и время возникновения события;
- "уровень серьезности" сообщения (от 0 до 7);
- старший и младший коды события (значения старших кодов определены в файле `/usr/include/sys/slogcodes.h`);
- текстовое сообщение (определяется разработчиком программы-клиента).

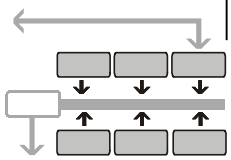
Для просмотра журнала используется утилита **sloginfo**.

Получение информации об оборудовании

Для проверки работы оборудования в QNX есть несколько утилит:

- **nicinfo** — выводит информацию о работе сетевой карты;
- **pin** — выводит информацию об устройствах PC Card;
- **pci** — выводит информацию о всех PCI-устройствах;
- **crttrap** — выполняет автоопределение видеокарты;
- **inputtrap** — выполняет автоопределение устройств ввода (клавиатура, мышь и т. п.).

Глава 15



High Availability Toolkit

В англоязычной документации русскому понятию "коэффициент готовности" соответствует High Availability (HA), что можно перевести как "высокая доступность". Мы будем пользоваться всеми этими терминами как синонимами (уж извините — привычка). Считается, что максимальное значение коэффициента готовности равно 0,99 999 (99,999 %), — это означает, что сервис, предоставляемый системой, недоступен клиентам не более пяти минут в год.

Для обеспечения высокого коэффициента готовности в QNX есть поддержка ряда механизмов:

- полная защита памяти процессов;
- возможность динамической реконфигурации системы без перезагрузки (это свойство особенно ценно при наличии аппаратуры, поддерживающей горячую замену);
- программная поддержка сетевых кластеров (Qnet);
- специализированные инструменты анализа и диагностики (в первую очередь, конечно, SAT);
- специализированный пакет High Availability Toolkit (HAT).

Итак, что за зверь — HAT? Этот пакет состоит из следующих компонентов:

- администратор доступности (High Availability Manager — HAM);
- библиотека функций взаимодействия с HAM — HAM API;
- клиентская библиотека — Client recovery library.

HAM и его API

Главным элементом HAT является администратор доступности — HAM ("живчик"). HAM при старте делает два дела (рис. 15.1):

- регистрирует зону ответственности — каталог `/proc/ham`;
- порождает свою точную копию — процесс Guardian ("страховщик").

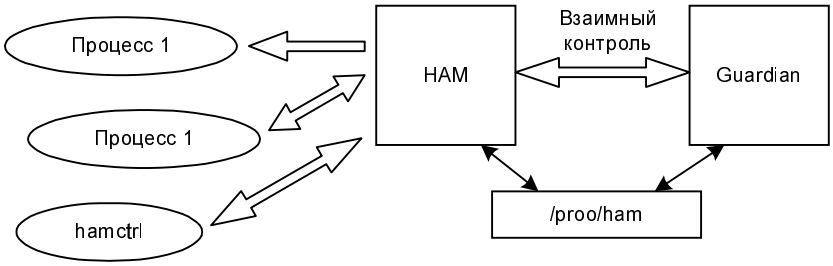


Рис. 15.1. Схема взаимодействия элементов при работе HAM

Корректно завершить процесс **ham** можно с помощью утилиты **hamctrl**, имеющей единственную опцию — **stop**. После того как процесс **ham** запущен, под его надзор можно ставить любые программы и задавать порядок надзора. Для выполнения этих операций используются функции HAM API.

Для того чтобы продолжить обсуждение HAT, введем понятия объекта, условия и действия. *Объект* (Entity) — процесс, контролируемый администратором доступности. Объекты бывают трех типов:

- *self-attached*, или *активный* (в прежних версиях). Это программа, которая знает о существовании процесса **ham** и может с ним взаимодействовать, например, посылать ему *зонд-сигналы* (heart-beats). Такая программа всегда компонуется с библиотекой `libham.a` (HAM API);
- *externally attached*, или *пассивный*. Такой объект ничего не знает о **ham**, его "сдает под наблюдение" другая программа. Обычно для выполнения функции "подключалки" пишут маленькую программку. Как правило, пассивный объект —

это написанная кем-то программа-сервер, для которой надо повысить коэффициент готовности;

- *глобальные*. Это не совсем объекты, скорее это механизм, позволяющий задавать действия в ответ на событие, произошедшее с любым из контролируемых процессов в системе. Глобальный объект нельзя добавить или удалить, он есть всегда. Но для него можно задавать условия и ответные действия при их выполнении.

Условие (Condition) — это, по сути дела, событие, связанное с объектом, которое должен отслеживать процесс **ham**. Определен ряд условий, например:

- Death — объект "умер";
- Detach — объект отключился от **ham**, т. е. отказался от дальнейшего контроля;
- Heartbeat Missed High — объект не посылает зонд-сигналы в течение интервала времени, ограниченного верхним пороговым значением;
- Heartbeat Missed Low — объект не посылает зонд-сигналы в течение интервала времени, ограниченного нижним пороговым значением;
- Restart — объект перезапущен. Это условие имеет значение "истинно" только после успешного перезапуска объекта.

Действие (Action) — это операция, которая должна быть выполнена при наступлении условия. Для условия можно задать, например, следующие действия:

- Restart — перезапустить объект;
- Execute — выполнить произвольную команду (например, запустить процесс);
- Notify Pulse — информировать процесс о наступлении условия путем посылки сообщения-импульса со значением, определенным процессом, желающим получать эти сообщения;
- Notify Signal — информировать процесс о наступившем событии, используя сигналы реального времени со значением, определенным процессом, желающим получать эти сообщения;

- Wait for — ожидать регистрации префикса в пространстве путевых имен (это действие позволяет вставлять задержки в последовательность действий);
- Heartbeat Healthy — выполнить возврат в начальное состояние механизма зонд-сигналов после перезапуска объекта.

Кроме обычных действий предусмотрены "действия при сбое действия" (Action Fail actions). "Сбойные" действия позволяют задать альтернативный алгоритм, когда невозможно выполнить стандартное действие по условию.

Откуда **ham** узнает, к примеру, о смерти процесса? О ненормальном завершении процессов его уведомляет процесс **dumper** (подробнее о **dumper** написано в *гл. 5, разд. " 'Посмертная' диагностика процессов"*).

Итак, после запуска администратора доступности **ham** нам следует зарегистрировать все необходимые компоненты: объекты/условия/действия. При регистрации компонентов в каталоге `/proc/ham` создаются соответствующие префиксы. Давайте предположим, что мы "сдаем под охрану" два объекта: **myprogram** и **hisprogram**. Для **myprogram** зададим условие Death (т. е. процесс **ham** должен реагировать на завершение **myprogram**) и для этого условия — действие Restart (т. е. **ham** должен перезапустить **myprogram**):

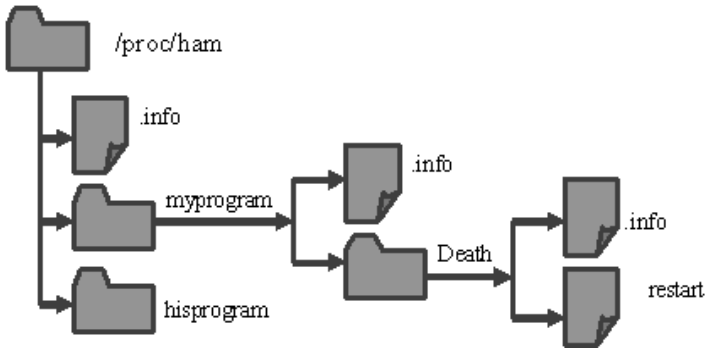


Рис. 15.2. Структура файловой системы /proc/ham

Итак, что же мы видим в каталоге `/proc/ham` (рис. 15.2)? Во-первых, файл `.info`, содержащий информацию о HAM

с Guardian, а также общую информацию о функционировании всей этой кухни. Во-вторых, каталоги, имена которых совпадают с именами контролируемых объектов. Каталог-объект имеет собственный общий описательный файл и каталоги-условия, т. е. каталоги с именами, совпадающими с названиями условий. По аналогии, каталог-объект имеет свой файл `.info`, содержащий сведения о данном объекте. Условия представлены обычными файлами, содержащими описание условия.

Ну что ж, теперь посмотрим, как выполняется регистрация всех этих компонентов. Для этого взглянем на псевдокод активного объекта:

```
НАЧАЛО
      подключиться к НАМ
      присоединиться к НАМ
      задать условия и действия
      НАЧАЛО_ЦИКЛА
            послать зонд-сигнал
            ...
      КОНЕЦ_ЦИКЛА
      отсоединиться от НАМ
      отключиться от НАМ
КОНЕЦ
```

Цикл посылки зонд-сигналов лучше выполнять в отдельном потоке. Рассмотрим псевдокод программы-"подключалки" пассивного объекта:

```
НАЧАЛО
      подключиться к НАМ
      присоединить пассивный объект к НАМ
      задать условия и действия для объекта
      отключиться от НАМ
КОНЕЦ
```

Обратите внимание, что программа для того, чтобы получить возможность присоединять (attach) как себя, так и другую программу к **ham**, должна сначала подключиться (connect) к нему.

Я умышленно не стал приводить здесь никаких примеров программ, т. к. их вполне достаточно в электронной документации, поставляемой с QNX Momentics PE.

Восстановление клиента

Итак, за доступность сервера отвечает HAM. Но ведь мы можем потерять связь с сервисной программой не только из-за ее сбоя, но, например, из-за проблем с линиями связи. Для того чтобы повысить надежность соединения, предназначена библиотека Client recovery library (`libha.a`). Эта библиотека содержит функции-"обертки" системных вызовов установления соединений и ввода/вывода. Проиллюстрируем этот механизм:

```
// открываем устройство и регистрируем наш обработчик -
recover_read_fd()
fd = ha_open( "/net/node1/dev/device", O_RDONLY,
recover_read_fd, "/dev/device", 0 );
...
// Обработчик
int recover_read_fd( int old_fd, void *hdl )
{
  fname = (char *) hdl;
  delay(100); // дадим возможность восстановить линию
  new_fd = ha_reopen( old_fd, fname, O_RDONLY );
  return new_fd;
}
```