ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ

Брянский государственный технический университет

О.Л. Никольский

ПРОГРАММИРОВАНИЕ ПРИЛОЖЕНИЙ РЕАЛЬНОГО ВРЕМЕНИ ДЛЯ ИСПОЛНЕНИЯ В СРЕДЕ ОПЕРАЦИОННОЙ СИСТЕМЫ РЕАЛЬНОГО ВРЕМЕНИ QNX/NEUTRINO 2

Часть II

ОБРАБОТКА ПРЕРЫВАНИЙ В ОПЕРАЦИОННОЙ СИСТЕМЕ РЕАЛЬНОГО ВРЕМЕНИ QNX / NEUTRINO

(Версия 9-02-2007)

БРЯНСК 2007

Научный редактор Дроздов А.В.
Технический редактор Королёва Т.И.
Компьютерный набор Никольский О.Л.

Рецензенты: кафедра “Вычислительная техника и прикладная математика” Магнитогорского государственного технического университета им. Г.И. Носова”.

Цилюрик О.И., научный редактор переводной компьютерной литературы из-ва «Символ-Плюс» г.С-Петербург.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ 4
1. КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ 6
1.1. АППАРАТНЫЕ УСТРОЙСТВА СИСТЕМЫ ПРЕРЫВАНИЙ 6
1.2. ВЗАИМОДЕЙСТВИЕ С АППАРАТУРОЙ ОС QNX/NEUTRINO 9
1.3. РАЗДЕЛЕНИЕ ПРЕРЫВАНИЙ 12
1.3.1. Обработка прерываний от нескольких устройств с одинаковым запросом прерываний в QNX/Neutrino 14
1.3.2. Распределение устройств по доступным линиям прерываний 15
1.3.3. Определение запроса на прерывание IRQ конкретного устройства 17
1.4. ПРОГРАММНЫЕ СРЕДСТВА ОБРАБОТКИ ПРЕРЫВАНИЙ 19
1.4.1. Виды уведомлений о прерываниях 25
1.4.2. Использование InterruptAttach() 26
1.4.3. Использование InterruptAttachEvent() 31
1.4.4. Прерывания и временно́е поведение системы 33
1.4.5. Прерывания и драйверы устройств 40
РЕЗЮМЕ 47
2. ТЕКСТЫ ПРОГРАММ 49
2.1. ПРОГРАММА ПЕРЕДАЧИ ДАННЫХ МЕЖДУ ДВУМЯ КОМПЬЮТЕРАМИ ПО ПОСЛЕДОВАТЕЛЬНОМУ КАНАЛУ С ИСПОЛЬЗОВАНИЕМ ФУНКЦИИ INTERRUPTATTACH() 49
2.2. ПРОГРАММА ПЕРЕДАЧИ ДАННЫХ МЕЖДУ ДВУМЯ КОМПЬЮТЕРАМИ ПО ПАРАЛЛЕЛЬНОМУ КАНАЛУ С ИСПОЛЬЗОВАНИЕМ ФУНКЦИИ INTERRUPTATTACHEVENT() 56
2.3. ПРОСТЕЙШИЕ ПРОГРАММЫ ДЛЯ ПРОВЕРКИ КОММУТАЦИИ ПАРАЛЛЕЛЬНЫХ ПОРТОВ 61
2.4. ПРОГРАММА ПОЛУЧЕНИЯ ИНФОРМАЦИИ ОБ ОБРАБОТЧИКАХ АППАРАТНЫХ ПРЕРЫВАНИЙ ПРОЦЕССОВ 62
3. ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНЫХ РАБОТ И ПОРЯДОК ИХ ВЫПОЛНЕНИЯ 75

ПРИЛОЖЕНИЕ 1. НАСТРОЙКА И УПРАВЛЕНИЕ РАБОТОЙ ПОСЛЕДОВАТЕЛЬНОГО ПОРТА ПРИ ПЕРЕДАЧЕ ДАННЫХ МЕЖДУ КОМПЬЮТЕРАМИ ПО НУЛЬ-МОДЕМНОМУ КАБЕЛЮ 78

ПРИЛОЖЕНИЕ 2. НАСТРОЙКА И УПРАВЛЕНИЕ РАБОТОЙ СТАНДАРТНОГО ПАРАЛЛЕЛЬНОГО ПОРТА ПРИ ПЕРЕДАЧЕ ДАННЫХ МЕЖДУ КОМПЬЮТЕРАМИ ПО НУЛЬ-ПРИНТЕРНОМУ КАБЕЛЮ 84

ЗАКЛЮЧЕНИЕ 87

СПИСОК ИСПОЛЬЗОВАННОЙ И РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ 88

ВВЕДЕНИЕ

Аппаратные прерывания являются эффективным механизмом взаимодействия с устройствами. В отличие от режима циклического опроса состояния устройства (*polling*), занимающего вычислительные ресурсы системы, прерывания позволяют программе асинхронно обращаться к устройству только в тех случаях, когда состояние устройства изменилось.

Механизм прерываний широко используется в программном обеспечении задач реального времени. Операционные системы (ОС) реального времени (РВ) предоставляют программисту удобные высокоуровневые средства для обработки и управления прерываниями. При этом в отличие от ОС общего назначения ОС РВ характеризуются более эффективными возможностями обработки прерываний. В частности, для ОСРВ QNX/Neutrino можно отметить следующее:

- Очень малые промежутки времени от возникновения прерыва-ния до начала его обработки обработчиком прерываний (inter-rupt latency) – и от завершения работы обработчика прерыва-ний до запуска высокоприоритетной нити, выполняющей свя-занные с прерываниями действия (scheduling latency);
- Минимальный интервал времени, в течение которого ядро ОС маскирует прерывания;
- Практическая независимость временных характеристик обра-ботки прерываний от загрузки системы;
- На поддерживающих эти возможности аппаратуре реализацию ме-ханизма вложенных прерываний (nested interrupts), обеспечи-вающего вытесняемый порядок обработки одновременно воз-никших прерываний в соответствии с их приоритетами (preemptive interrupt scheduling).
- Возможность связывания нескольких обработчиков прерываний с одним и тем же запросом на прерывания (IRQ), что позволяет использовать устройства, разделяющие общий IRQ.

Как и для других разделов пособия по программированию приложений реального времени в ОС РВ QNX/Neutrino, обязательно изучение соответствующей главы книги [1], которая является необходи-

мым “букварём” любого QNX-программиста, а также знакомство с соответствующими разделами справочной системы QNX. Весьма полезным является ознакомление с учебным курсом Р.Кёртена, выложенным на его сайте по адресу http://parse.com/products/training/coursware/index.html.

1. КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Система прерываний компьютера представляет совокупность аппаратных и программных средств, реализующих механизм прерываний [2].

1.1. АППАРАТНЫЕ УСТРОЙСТВА СИСТЕМЫ ПРЕРЫВАНИЙ

Программирование обработки аппаратных прерываний, как никакая другая сфера программирования систем реального времени, требует учёта особенностей конкретной аппаратуры со всеми её плюсами и минусами и ближайшего к аппаратуре уровня программного обеспечения (firmware).

Изначально на платформах с процессорами Intel аппаратные прерывания воспринимал программируемый контроллер прерываний (Programmable Interrupt Controller - PIC) 8259A [3], имеющий 8 входных линий. Обычно использовался каскад из двух таких контроллеров, причём выход ведомого связан со входом 2 ведущего, что расширяло количество обслуживаемых источников внешних прерываний до 15 [2]. На современных материнских платах персональных компьютеров и серверов с процессорами Intel используются усовершенствованные контроллеры (Advanced PIC-APIC), которые поддерживают мультипроцессорные системы, увеличивают количество доступных линий прерываний, а также обладают рядом дополнительных возможностей по обработке разделяемых прерываний [4,5,6,7]. Сам термин APIC обозначает совокупность аппаратных компонентов, включающую:

- локальные контроллеры APIC (local APIC), встроенные в процессоры. В числе их функций – взаимодействие процессоров друг с другом через механизм прерываний;
- контроллер APIC ввода/вывода (IOAPIC), который обрабатывает прерывания, поступающие от устройств ввода/вывода. Среди прочего выполняет функции каскада контроллеров 8259A.

APIC имеет режимы работы, совместимые со стандартной связкой двух PIC (рис.1.1). Из недостатков APIC отмечается больший джиттер (флуктуации) времени задержки прерываний (interrupt latency) [8] по сравнению с каскадом традиционных PIC.

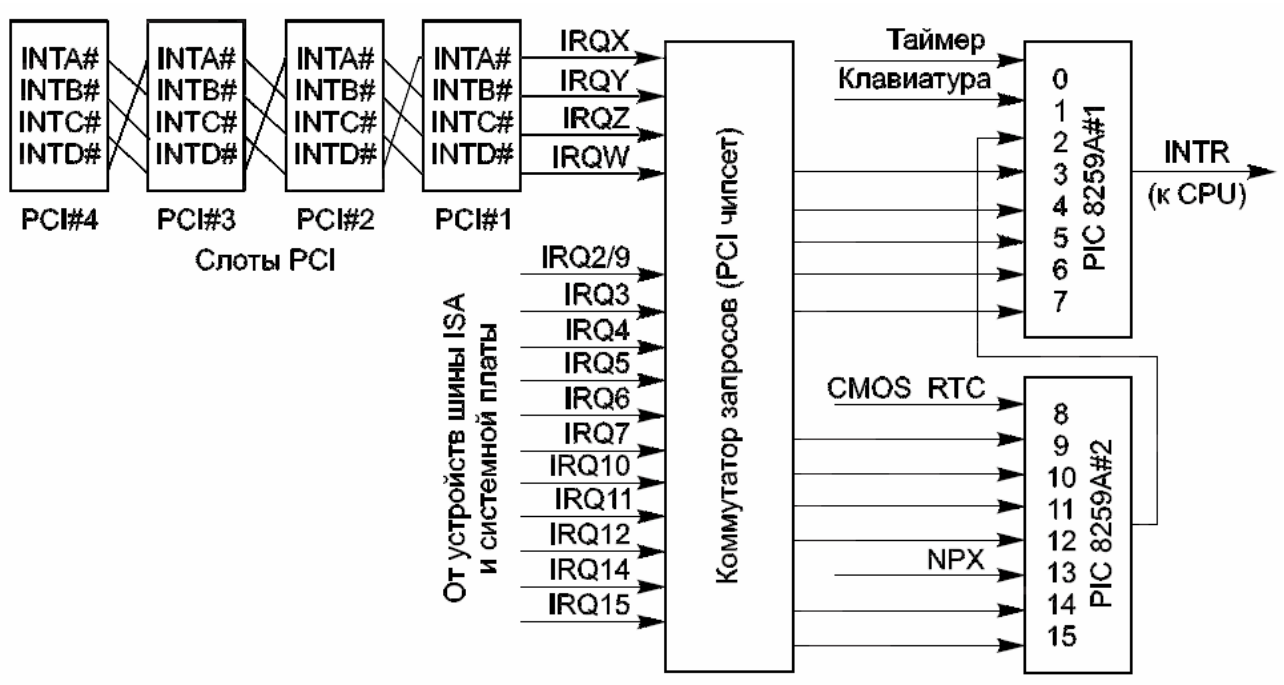

Рис.1.1. Формирование запроса прерываний
с использованием каскада PIC [5]

На материнской плате обычно располагается 4 PCI-слота, но может быть и больше. “На входы контроллеров прерываний поступают запросы от системных устройств (клавиатура, системный таймер, CMOS-таймер, сопроцессор), периферийных контроллеров системной платы и карт расширения. Запросы прерываний 0, 1, 8 и 13 на шины расширения не выводятся. Традиционно все линии запросов, не занятые перечисленными устройствами, присутствуют на всех слотах шины ISA/EISA. Эти линии обозначаются как IRQx и имеют общепринятое назначение. Часть этих линий отдается в распоряжение шины PCI. Для устройств PCI выделяется четыре проводных линии запросов (IRQX, IRQY, IRQZ, IRQW), соединяемых с контактами INTA#, INTB#, INTC# и INTD# всех слотов PCI с циклическим смещением цепей” [5]. Циклическое смещение определяет, что со входом IRQX связаны контакт INTA# слота 1, INTD# слота 2, INTC# слота 3 и INTB# слота 4, и т.п. (см. рис.1). Это позволяет установить в 4 соседних слота 4 простых (т.е. требующих только одну линию запроса прерываний) устройства, и каждое из них будет занимать отдельную линию запроса прерывания. Важно иметь в виду, что некоторые, а возможно, и все PCI-слоты используют линии прерываний параллельно с некоторыми устройствами материнской платы (USB контроллеры, встроенные сетевые и/или аудио адаптеры, слот AGP и т.д.). “Спаренность” слота с одним из “on-board” устройств жёстко

заложена в конструкции материнской платы, зафиксирована в карте прерываний материнской платы [9] и никоим образом не может быть изменена.

Рассмотренная модель аппаратной части системы прерываний является стандартной для платформы x86 под управлением ОС РВ QNX/Neutrino.

Помимо APIC, следуя рекомендациям спецификаций PC99 и PC2001, современные компьютеры с Intel-архитектурой, включая персональные, аппаратно поддерживают ACPI (Advanced Configuration and Power Interface) - расширенный интерфейс конфигурирования и управления питанием, ответственный за автоматическое распределение системных ресурсов, включая прерывания (interrupt sharing), внутри компьютера [9,10]. Спецификация ACPI устанавливает интерфейс для использования APIC-контроллеров на машинах с ACPI. Основное достоинство ACPI с точки зрения системы аппаратных прерываний – способность автоматического - без участия и с запретом вмешательства пользователя - распределения устройств по доступным линиям прерываний (interrupt steering, interrupt routing). Общие прерывания могут разделять PCI- и PnP ISA- устройства, традиционные ISA-устройства к использованию общей линии прерываний не способны [5]. Главную работу по распределению прерываний выполняет процедура POST BIOS, операционная система может либо оставить распределение устройств по линиям прерываний без изменения (QNX/Neutrino), либо провести дальнейшую оптимизацию распределения, если её возможности позволяют это (последние версии ОС Windows, см., например [11]). С точки зрения систем реального времени отдавать настройку системы прерываний на откуп ACPI не всегда приемлемо. “Если устройство соглашается работать в режиме кооперации с другим устройством, то есть все шансы, что ACPI посадит их на одну физическую линию. Если не контролировать данную ситуацию, то на одном физическом прерывании могут оказаться практически все устройства, установленные в компьютере, даже если есть свободные прерывания. Это приведет к сильнейшему торможению всей системы и серьезным сбоям в работе” [9].

Материнские платы персональных компьютеров могут иметь различные конфигурации [12]:

- Многопроцессорный компьютер с ACPI: Используется на многопроцессорных компьютерах с ACPI.
- Однопроцессорный компьютер с ACPI: Используется на многопроцессорных компьютерах с ACPI, но только с одним установленным процессором.
- Компьютер с ACPI: Используется для системных плат, позволяющих установить только один процессор.
- Однопроцессорный компьютер с MPS (соответствующий Intel MultiProcessor Specification - мультипроцессорной спецификации Intel [4]): Используется на двухпроцессорных системных платах без поддержки ACPI и только с одним установленным процессором.
- Многопроцессорный компьютер с MPS: Используется на двухпроцессорных системных платах без поддержки ACPI с двумя установленными процессорами.
- Стандартный компьютер: Любой компьютер без поддержки ACPI или без MPS. Это может быть система, основанная на процессоре i386, i486, Pentium, Pentium II или Pentium III
- Стандартный компьютер i486 степпинг-C.
- Некоторые другие варианты, не вошедшие в предыдущие группы.

Контроллер с расширенными возможностями IOAPIC и и традиционный PIC могут сосуществовать в чипсете одной материнской платы [13]. Хотя APIC разрабатывался для многопроцессорных плат, его использование даёт определённые преимущества с точки зрения обработки прерываний и на однопроцессорных [14].

Специализированные платы контроллеров и промышленных компьютеров, являющиеся аппаратной платформой для встроенных систем реального времени на основе QNX/Neutrino, могут не иметь некоторых из вышеперечисленных компонентов. В любом случае конфигурация ответственных за службу прерываний устройств конкретной платы известна её BIOS.

1.2. ВЗАИМОДЕЙСТВИЕ С АППАРАТУРОЙ ОС QNX/NEUTRINO

Независимо от типа контроллеров прерываний на материнской плате контроллеры инициализируются стартовым кодом ОС QNX/Neutrino при запуске операционной системы. Нет никаких ограничений на типы установленных на материнской плате контроллеров. Для каждого из них в стартовый код должны быть помещены со-

ответствующие процедуры, задающие порядок обслуживания аппаратных прерываний и их приоритеты (см. раздел справочной системы “Building Embedded Systems/Appendix: System Design Considerations”). Стартовый код заполняет область `intrinfo` системной страницы, содержащую информацию о системе прерываний, а также функции (`callouts`) для управления аппаратной частью контроллеров. При построении встроенных систем для нестандартного “железа” имеется возможность непосредственно запрограммировать в стартовом коде конкретные особенности и способы инициализации контроллеров (см. раздел справочной системы QNX/Neutrino “Customizing Image Startup Programs”). Примеры build-файлов для построения загружаемых образов с указанием подразумеваемого операционной системой распределения прерываний приведены в каталоге `/x86/boot/build.`

Для установленной на платформе x86 ОС QNX/Neutrino стартовый код - это программа `startup-bios`, которая поддерживает стандартную конфигурацию двух master-slave 8259-совместимых контроллеров (см. рис.1). Для платформы x86 по умолчанию устанавливаются приоритеты прерываний, приведенные в табл. 1.1.

Таблица 1.1

Типовые уровни прерываний IRQ и приоритеты прерываний устройств для платформы x86 (см. справку по `InterruptAttach()` и файлы `/x86/boot/build/*.build`)

Запрос прерывания IRQ	**Приоритет прерывания**	**Устройство, вызывающее прерывание**
0*	Высший 2	Канал 0 аппаратного таймера, отсчитывающий системные такты с периодом ClockPeriod()
1*	3	Клавиатура
2*		Каскадирование прерываний (перенаправление на второй контроллер 8259)
3*	12	Асинхронный порт COM2 (или COM4)
4*	13	Асинхронный порт COM1 (или COM3)
5	14	Резерв: Параллельный порт LPT2 / сетевая карта / звуковая карта / другие устройства

Окончание табл. 1.1

6	15	Контроллер флоппи-диска
7	Низший 16	Параллельный порт LPT1 / звуковая карта / другие уст-ва
8	4	Резерв (Могут использовать часы реального времени (CMOS RTC)
9	5	Перенаправленное IRQ 2
10	6	Резерв
11	7	Резерв
12	8	Резерв (может использовать мышь PS/2)
13	9	Математический сопроцессор
14	10	Первый контроллер жесткого диска
15	11	Второй контроллер жесткого диска

Примечание. Помеченные * логические векторы прерываний зафиксированы в build-файлах и соответствуют стандартной конфигурации материнских плат с процессорами x86. Все они соответствуют чувствительности контроллера PIC по возрастающему фронту (rising edge).

Присущий аппаратуре Intel наивысший приоритет прерываний от аппаратного таймера имеет как положительные, так и отрицательные стороны. В отчёте [15] отмечается, что обслуживание этих прерываний может замедлить критичную по времени реакцию системы на важные аппаратные прерывания в случае вытеснения обработчиком прерываний таймера обработчика прерывания другого устройства, хотя замедление это мало (4 мкс на Pentium 200 МГц). Вместе с тем это способствует более точному отсчёту системного времени и, соответственно, более точной работе абсолютных и относительных таймеров.

Разработчики QNX/Neutrino формулируют несколько практических рекомендаций, повышающих надёжность и скорость обработки прерываний:

– Не обращаться к контроллеру прерываний напрямую через его регистры. Пользоваться только функциями библиотеки;
– не использовать запросы прерываний IRQ7 и IRQ15 на чипе 8259, которые названы разработчиками QNX как "glitch interrupts" (“глюкавые прерывания”) и ведут себя ненадёжно. Проблема с этими прерываниями под названием “spurious IRQ7 interrupt” многократно обсуждалась на форумах Linux, FreeBSD и т.д., например [16,17]. Суть проблемы состоит в том [5], что независимо от чув-

ствительности по уровню или фронту “сигнал запроса аппаратного прерывания IRQ должен удерживаться генерирующей его схемой, по крайней мере, до цикла подтверждения прерывания процессором - именно в этот момент PIC определяет самый приоритетный незамаскированный запрос и по нему формирует вектор. Если к этому моменту запрос будет снят, источник прерывания корректно идентифицирован не будет и контроллер сообщит *ложный вектор прерывания* (spurious interrupt), соответствующий его входу с максимальным номером (IRQ7 для первого контроллера и IRQ15 для второго). Обычно периферийные устройства строят так, что сигнал запроса сбрасывается при обращении программы обслуживания прерывания к соответствующим регистрам адаптера, так что ложных прерываний возникать не должно”. Кратковременный сигнал IRQ может возникать также по причине наведённого “шума” на линиях прерываний, что никак не связано с нормальным функционированием устройств, возбуждающих прерывания .

– избегать использования немаскируемых прерывания (Non-Maskable Interrupt) на платформах x86. Платформы PPC, MIPS, ARM, и SH-4 их даже не поддерживают;
– не использовать слишком глубокую иерархию аппаратных контроллеров прерываний – один большой контроллер лучше цепочки нескольких вложенных.

1.3. РАЗДЕЛЕНИЕ ПРЕРЫВАНИЙ

При недостатке свободных линий прерываний или особенностях архитектуры вычислительной системы приходится использовать *разделяемые прерывания* между несколькими периферийными устройствами (*shared interrupts*). При этом возникает ряд проблем, в частности, распределение устройств по доступным линиям прерываний и обработка прерываний от нескольких устройств, разделяющих общий IRQ.

Для правильной организации совместного использования одной линии прерывания важно представлять способ восприятия контроллером изменение сигнала на линии прерываний шины. Контроллер прерываний может быть чувствителен (и запрограммирован соответственно) либо к уровню (level) сигнала прерывания, либо к его фронту, или перепаду (edge) [5]. В любом случае, по завершении обработ-

ки прерывания, контроллеру прерываний должен быть послан сигнал EOI (**E**nd**O**f**I**nterrupt), получив который, PIC готов обрабатывать очередное прерывание.

В чувствительной к перепаду архитектуре аппаратуры прерываний изменение состояния линии прерывания регистрируется, только когда нулевое состояние меняется на положительное. “Для шины ISA с её запросами прерываний по положительному перепаду разделяемость прерываний невозможна. Исключения составляют системные платы и устройства с поддержкой ISA PnP, которые можно заставить работать и по низкому уровню” [5]. Для устройства на шине ISA никакое следующее прерывание не может быть обнаружено, пока функция-обработчик прерывания ISR (Interrupt Service Routine) не завершит обработку текущего прерывания и очистит его источник, вернув линию в исходное (нулевое) состояние. Если до очистки источника прерывания другое устройство выдаст на шину сигнал прерывания (наложенное прерывание, прерывание с перекрытием), никакого очередного перепада относительно нулевого уровня не возникнет – линия прерывания уже находится в активном состоянии, так как источник прерывания не был очищен. После очистки источника прерывания первого устройства линия вернётся в нулевое состояние и наложенное прерывание от второго устройства будет потеряно.

“Для шины PCI с аппаратной точки зрения проблема разделения прерываний решена - здесь активным уровнем запроса является низкий, и контроллер прерываний чувствителен к уровню, а не перепаду ... Прерывание считается активным, пока линия прерывания находится в активном состоянии. Когда закончена обработка прерывания от одного устройства, ISR перед возвращением должен очистить источник прерывания, или замаскировать его. По завершении ISR ядро ОС посылает контроллеру прерываний сигнал EOI – уведомление о том, что прерывание обработано. Если после этого линия прерывания находится в активном состоянии, вступает в работу следующий драйвер...”[5]. Отметим, что некоторые аппаратные компоненты позволяют задавать конфигурацию их чувствительности к прерыванию либо по уровню, либо по фронту, например параллельный порт с расширенными возможностями (ECP - Extended Capabilities Port) [18].

Все драйверы (целиком или частью в форме ISR) устройств, разделяющих общее прерывание, выстроены в цепочку и запускаются последовательно. Необходимость формирования цепочки драйверов

связана с невозможностью идентификации устройства, вызвавшего разделяемое прерывание. “В первых версиях PCI (до PCI 2.2 включительно) не было общепринятого способа программной индикации и запрета прерываний. К сожалению, в конфигурационных регистрах {контроллера} не нашлось стандартного места для бита, индицирующего введение запроса прерывания данным устройством, — тогда бы в прерываниях для PCI не было бы проблем с унификацией поддержки разделяемых прерываний. В каждом устройстве для работы с прерываниями используются свои специфические биты операционных регистров, относящихся к пространству памяти или ввода/вывода (иногда и к конфигурационному). При этом определить, является ли данное устройство в текущий момент источником прерывания, может только его обработчик прерывания, входящий в драйвер данного устройства. Таким образом, у ОС нет иной возможности диспетчеризации разделяемых прерываний, кроме как выстроить их ISR-ы в цепочку” [5].

1.3.1. Обработка прерываний от нескольких устройств с одинаковым запросом прерываний в QNX/Neutrino

ОС QNX даёт возможность поместить функции-обработчики прерываний (Interrupt Service Routine - ISR), связанные с одним IRQ, в цепочку (QNX allows ISR handlers to be stacked), каждый член которой вызывается последовательно, определяя свою ответственность за обработку конкретного прерывания и обрабатывая его. Ядро само посылает контроллеру прерываний сигнал EOI только после того, как отработали все ISR в цепочке. Код пользовательской программы не должен генерировать команду EOI (См. раздел справочной системы QNX/Neutrino “Writing an Interrupt Handler”).

Необходимо учитывать, что функция-обработчик прерывания ISR имеет приоритет выше приоритета любого потока и запускается немедленно после возникновения прерывания, вытесняя любой поток. Поэтому для соблюдения требований реального времени рекомендуется код ISR делать максимально коротким, включающим либо короткую обработку прерывания и очистку источника прерываний, либо для устройств, требующих длительных действий по обработке прерывания, маскировку запроса прерывания и запуск специально предназначенной для этого нити (interrupt service thread, interrupt handling thread) (более подробно см. пп. 1.4.2). К таким устройствам от-

носятся, например, контроллеры жёстких и гибких дисков, требующие несколько миллисекунд для того, чтобы, опросив регистры, очистить источник и после этого размаскировать соответствующий IRQ. Требованиям минимальных затрат времени на выполнение ISR служит и ограниченное количество функций библиотеки и вызовов ядра (см. пп.1.4.2), которые могут в нём использоваться. В справке по каждой из функций указана допустимость использования её в ISR.

Важный вопрос при организации разделяемых прерываний - порядок вызовов микроядром драйверов устройств для обработки общего для них запроса прерываний. Относительно этого порядка программист не должен делать никаких предположений [19]. Вновь присоединяемый с помощью функций InterruptAttach() или InterruptAttachEvent() ISR по умолчанию ставится системой в начало списка драйверов для этого IRQ. С помощью флага _NTO_INTR_FLAGS_END (подробно см. пп. 1.4.2) можно расположить ISR в конце очереди. Нет возможности задать какой-либо иной порядок расположения, например, посередине, 5-м по порядку, 2-м, и т.п. (См. раздел справочной системы QNX/Neutrino “Writing an Interrupt Handler”).

1.3.2. Распределение устройств по доступным линиям прерываний

Распределять доступные линии прерываний между различными устройствами следует по возможности так, чтобы работающие по прерываниям устройства как можно меньше мешали друг другу и не тормозили работу системы в целом. Хотя единого универсального средства нет, существует несколько способов нормального разведения доступных линий прерываний. В зависимости от конкретного железа и BIOS эти способы могут объединять некоторые из нижеперечисленных действий (“Зачастую приходится действовать научным «методом тыка» :-)” [20]):

– В настройках BIOS следует отключить все ненужные устройства (например, один из последовательных или параллельных портов) и завершить все процессы-менеджеры этих устройств, если они загружаются операционной системой по умолчанию, освободив занятые ими линии прерываний.
– В настройках BIOS отключить опцию поддержки операционной системой стандарта Plug&Play.

– На однопроцессорных машинах отключить использование APIC.
– Линиям прерываний, требуемым стандартными ISA-устройствами, например IRQ7 для параллельного порта, в настройках BIOS следует установить режим “Legacy ISA”.
– Если такую возможность поддерживает BIOS, задать вручную распределение уровней прерываний по слотам шины PCI [9,20]. QNX/Neutrino позволяет автоматизировать этот процесс при загрузке, включив в стартовый код вызов утилиты `pci-bios` с соответствующими опциями (см. справку по утилите в справочной системе Neutrino).
– Подобрать установку PCI-устройств так, чтобы они по возможности располагались в слотах, не связанных жёстко друг с другом и со встроенными устройствами (по образному выражению Lestat “Таких друзей надо рассаживать по разным камерам”). В этом может помочь карта прерываний конкретной материнской платы. [9,20]. Помещать на одну линию прерываний только те устройства, которые терпимо относятся к соседям. По этому признаку все устройства условно (очень многое зависит от конкретной реализации) делят на 4 группы [9]:

  а) видеокарта и контроллеры жестких дисков. Обязательно должны монопольно занимать линию прерывания, иначе гарантированы большие задержки (“тормоза”) в работе системы;

  б) сетевая карта, ТВ-тюнер менее ресурсоёмки, могут терпеть соседей из следующих двух групп в) и г), но не друг друга. Для стабильной работы желательно тоже располагать каждое из этих устройств на отдельной линии прерываний;

  в) все порты ввода вывода, USB, COM-порты, возможно LPT, IEEE1394 (FireWire), аппаратные внутренние модемы. Могут висеть на одном прерывании друг с другом и при крайней необходимости с устройствами группы б);

  г) редкие устройства, вообще не требующие прерываний (например, ускорители трёхмерной графики Voodoo и Voodoo 2). Могут помещаться в любые свободные слоты независимо от того, как линии прерываний этих слотов накладываются на используемые линии других слотов.

  В любом случае не следует размещать медленные устройства на одной линии прерывания с быстрыми.

– “Если какой-то карте требуется 2 линии, то для монопольного использования прерываний нужно оставить соседний слот свободным. Однако не следует забывать, что PCI-устройства системной платы тоже задействуют прерывания с той же закономерностью (кроме контроллера IDE, который, к счастью, держится особняком). Порт AGP в плане прерываний следует рассматривать наравне со слотом PCI. Таким образом, может оказаться, что монопольные линии прерывания присутствуют далеко не на всех слотах” [5].

1.3.3. Определение запроса на прерывание IRQ конкретного устройства

Отдельный вопрос – как определить, какое устройство в конкретной системе использует какой IRQ и в случае ненормальной работы какой IRQ оно требует? Существуют разнообразные способы получить эту информацию:

– Распределение IRQ на уровне BIOS во многих случаях выводится на экран в момент начальной загрузки компьютера. Проще всего прочесть эту информацию после зависания компьютера при загрузке с несистемной дискеты [9] или приостановив процесс загрузки, нажав клавишу Pause.

– Если на компьютере установлены ещё другие, кроме QNX, операционные системы, правильно работающие с аппаратными устройствами, можно воспользоваться их информацией, например - посмотреть вывод диспетчера устройств ОС Windows или вывод специализированной программы SiSoftware Sandra. При этом нужно учесть, в отличие от QNX, некоторые ОС, в т.ч. Windows, могут сами распределить устройства по линиям прерываний (IRQ steering) независимо от того, что сделано BIOS.

– В установленной ОС QNX/Neutrino полезную информацию по IRQ можно извлечь, запустив утилиты-перечислители (энумераторы) и проанализировав их вывод. Для всех устройств информацию выдаёт enum-devices (командная строка `#enum-devices -vvvn 2`), отдельные классы устройств доступны с помощью утилит `enum-pnpbios`, `enum-pnpisa`, `enum-legacy`, `enum-par`.

– Для PCI-устройств подробная информация выводится утилитой `pci`. Запущенная без параметров утилита выводит сведения,

включая IRQ, для жёстких дисков, графического и сетевого адаптеров. С опцией `-vvv` утилита выводит информацию о всех PCI-устройствах. Пример вывода утилиты `pci` (строки, не относящиеся к IRQ, не показаны, конкретные значения соответствуют 82371AB/EB PIIX4 USB Interface):

- `PCI Int Pin = INT D` –линия запроса прерывания слота PCI,
- `Interrupt line = 10` - номер входа контроллера прерываний, доставшийся устройству.

Фильтрованный вывод о прерываниях с идентификаторами всех PCI-устройств выдаёт конвейер команд

```
pci -v | grep -e Device -e Interrupt.
```

Отметим, что PCI-сервер в QNX/Neutrino 6.2.1 имеет недоработки, поэтому некоторые данные, в частности адреса ввода / вывода устройств могут не отображаться [21].

– Специальная утилита участника форума Ed1k `picinfo` [22] позволяет определить настройки контроллера прерываний, ответив на вопрос, какая из линий чувствительна по уровню (к ней могут быть подключены несколько PCI-устройств, разделяющих общий IRQ), какая – по фронту (только одно ISA/EISA устройство). Имеется также разработанная тем же автором утилита `SETIRQ`, позволяющая устанавливать IRQ PCI-устройства, расположенного в конкретном слоте [23].

– Для определения конфигурации PCI-устройств из программы можно использовать функцию `pci_attach_device()`. Эта функция заполняет структуру `pci_dev_info`, один из членов которой содержит IRQ устройства, пример приведен в [24] (см. также справку QNX/Neutrino по `pci_attach_device()`).

– Другой вариант программного способа получения информации о прерываниях основан на извлечении её из файловой системы `/proc` [25]. Текст программы приведен в п. 2.4.

– Получить информацию от запущенного системой драйвера об уровне прерывания обслуживаемого им устройства можно, если этот уровень задаётся стартовым кодом как параметр драйвера. Например, команда `#pidin –F"%A"` для менеджера последовательных портов `devc-ser8250` выдаёт строку

```
devc-ser8250 -u1 3f8,4 -u2 2f8,3.
```

Согласно документации для этого драйвера в справочной системе видим адреса `3f8` и `2f8` и IRQ 4 и 3 соответственно для 1-го и 2-го портов. Разработчики QNX рекомендуют для разрабатываемых драйверов устройств предусматривать возможность задания номера прерывания в командной строке (см. справку по функции `InterruptAttach()`). К сожалению, сами создатели входящих в дистрибутив QNX драйверов редко следуют этой рекомендации, так что без специальных изысков бывает не ясно, использует ли драйвер устройства прерывания, и если использует, то какой IRQ им соответствует.

Дополнительные рекомендации по обработке прерываний от устройств, используемых в системах реального времени, можно найти на уже упоминавшемся русскоязычном форуме http://qnx.org.ru/forum/, а также на англоязычном форуме http://www.openqnx.com/ и в группах новостей inn.qnx.com (более подробно см. пп. 1.4.5).

На используемых в лабораторных работах по курсу “Системы реального времени” персональных компьютерах PCI-BIOS распределяет ресурсы PCI-устройств и настраивает соответствующие линии прерываний контроллера для работы по уровню. Прерывания последовательных и параллельных портов, используемых для коммуникации компьютеров, обрабатываются контроллером в режиме чувствительности по фронту.

1.4. ПРОГРАММНЫЕ СРЕДСТВА ОБРАБОТКИ ПРЕРЫВАНИЙ

> “…А вообще с прерываниями в QNX … никаких проблем никогда не возникало, все не просто, а очень просто.”
>
> М.Горчак (lestat), активный участник форума qnx.org.ru/forum, автор многочисленных программных разработок и публикаций по тематике QNX.

Несмотря на сложную структуру системы прерываний, в программах, предназначенных для исполнения в среде QNX/Neutrino явным образом обращаться к контроллеру прерываний на уровне портов вода/вывода не требуется. Микроядро ОС берёт все низкоуров-

невые действия по управлению контроллером на себя, предоставляя программисту удобный высокоуровневый API (табл. 1.2).

В отчёте [15] эксперты следующим образом описывают методику обработки прерываний в QNX/Neutrino: “Обработчик прерываний в микроядре осуществляет управление прерываниями на их начальной стадии. Нить, имеющая соответствующие привилегии, может динамически устанавливать и удалять обработчики прерываний путем передачи микроядру адреса функции-обработчика (Interrupt Service Routine - ISR)”. В ядре могут также регистрироваться события, которыми ядро уведомляет процесс или нить о возникшем прерывании. “При возникновении прерывания, обработка передается сначала в микроядро, которое содержит код для его переадресации. Перед вызовом ISR микроядро сохраняет контекст исполняемой нити и переустанавливает регистры процессора таким образом, что ISR имеет доступ к адресному пространству нити, в которой он содержится.

Это позволяет ISR выполнить обработку, пользуясь данными и буферами нити, в которой она содержится, или буферизовать принятые данные для последующей обработки этой нитью. Это также дает возможность ISR осуществлять доступ ко всем устройствам, находящимся в домене ответственности (области префиксов) данной нити, и непосредственно выполнять операции ввода-вывода. QNX/ NEUTRINO RTOS v6.2 поддерживает разделение прерываний. При возникновении прерывания, каждая программа обработки соответствующего прерывания, связанного с аппаратным прерыванием, обрабатывается по очереди. Пользователь не должен брать на себя ответственность за порядок вызова программ обработки прерываний. Прерывания не блокируются во время выполнения управляющей прерываниями программы, таким образом, прерывания могут быть вложенными. Немаскируемые прерывания могут быть обслужены во время выполнения уже запущенной программы обработки прерывания.”

Для того, чтобы программа реагировала на аппаратные прерывания, вначале необходимо привязать (attach) прерывание с заданным IRQ к обработчику прерываний и заполнить структуру-событие (`struct sigevent`), определяющую тип и параметры уведомления, с помощью которого ядро ОС или функция-обработчик (ISR) информирует пользовательскую программу о наступлении прерывания.

Все источники прерываний, не связанные с к.-л. ISR, маскируются ядром и автоматически размаскируются, когда хотя бы один ISR присоединяется к источнику (раздел справочной системы QNX/Neutrino “Writing an Interrupt Handler”). Это позволяет предотвратить непредсказуемое поведение системы в ответ на аппаратное прерывание, когда не задан адрес обработчика.

Программный код, обрабатывающий прерывания, может выполняться:

А. На уровне потока. В этом случае для присоединения прерывания используется библиотечная функция `InterruptAttachEvent()`. Эта функция принуждает микроядро генерировать событие-уведомление о прерывании, воспринимаемое потоком, обрабатывающим прерывание. Этот поток диспетчируется ядром на выполнение по тем же правилам, что и все остальные потоки системы, в частности, он может быть вытеснен более высокоприоритетным потоком, обработчиком сигнала или обработчиком прерывания ISR;

Б. На уровне сверхприоритетной функции-обработчика ISR, которая выполняет самую необходимую работу по идентификации источника прерываний и его очистке. Здесь используется библиотечная функция `InterruptAttach()`. Обработчиков ISR может быть несколько, если один и тот же IRQ используется более чем одним устройством. Последующие, связанные с прерываниями действия, если они необходимы, выполняет специально предназначенная для этого нить, запускаемая из ISR с помощью события-уведомления. Функция-обработчик (ISR) имеет приоритет выше, чем приоритет любой нити программы, поэтому диспетчеризуется ядром немедленно. Чтобы обеспечить наиболее быструю реакцию системы на возникающие одно за другим прерывания, действия, выполняемые ISR, должны быть минимальными. В частности, допускается использование в нём только строго ограниченного круга библиотечных функций и системных вызовов (подробно см. пп. 1.4.2).

Таблица 1.2

Сводка вызовов микроядра для работы с прерываниями

Имя вызова ядра	**Назначение**	**Указания по применению и примечания**
1	2	3
InterruptAttach()	Присоединяет функцию-обработчик прерывания к устройству-источнику прерываний	Как и для многих других функций, имеются варианты вызовов *_r, которые не возвращают глобальные коды ошибок errno. Использование описано ниже и проиллюстрировано текстами программ.
InterruptAttachEvent()	Связывает генерируемое ядром событие с устройством-источником прерывания	
InterruptDetach()	Разрывает связь обработчика прерываний с заданным IRQ.	
InterruptWait()	Блокирует вызывающий поток до наступления аппаратного прерывания	
InterruptDisable()	Запрещает все аппаратные прерывания до вызова InterruptEnable()	Аналог ассемблерной инструкции cli
InterruptEnable()	Разрешает все аппаратные прерывания после их запрета вызовом InterruptDisable()	Ф-ция должна вызываться как можно скорее после InterruptDisable() чтобы уменьшить возможную задержку обработки других прерываний и не ухудшить “реальновременное” функционирование системы. Запрет/разрешение прерываний выполняется в процессоре.

Продолжение табл. 1.2

Имя вызова ядра	Назначение	Указания по применению и примечания
InterruptLock()	Используется как средство взаимоисключающего доступа к разделяемым потоком и обработчиком прерываний данным. Захватывает переменную - примитив синхронизации `spinlock` когда прерывания запрещены. Если `spinlock` не доступен, загружает процессор циклом до тех пор, пока `spinlock` не станет доступен.	Оба вызова работают как на однопроцессорных, так и многопроцессорных машинах. Могут вызываться из потоков и обработчиков прерываний. Важно освобождать `spinlock` как можно скорее после его захвата, поместив в критической секции ограниченное количество строк кода без циклов. Рекомендуется использовать эти вызовы вместо InterruptDisable() и InterruptEnable() соответственно, что даёт возможность обработчику прерываний работать на любом процессоре SMP-архитектуры.
InterruptUnlock()	Освобождает критическую секцию, защищённую примитивом синхронизации `spinlock`, снова делая возможным обработку прерываний.	

Примечания.

- Работу с прерываниями имеет право выполнять только пользователь с правами root.
- Перед обращением к ядру для обработки прерываний поток должен получить права на операции с портами ввода/вывода и изменение флага прерываний процессора с помощью вызова ThreadCtl(_NTO_TCTL_IO, 0).
- Обратите внимание в справках по функциям библиотек QNX/Neutrino, какие из них не допустимо использовать в обработчиках прерываний.

Окончание табл. 1.2

Имя вызова ядра	Назначение	Указания по применению и примечания
InterruptMask()	Запрещает обработку прерываний с заданным IRQ для конкретного обработчика прерываний.	Используются в случаях, когда очистка источника прерываний занимает длительное время. В этом случае работа с устройством выполняется не обработчиком прерываний со сверхвысоким приоритетом, а выносится в отдельный поток. На время его работы прерывание маскируется (например, в драйвере флоппи-диска). Закончив работу и очистив источник прерываний, поток вновь разрешает их обработку. Могут вызываться как из потока, так и из функции – обработчика прерываний. Маскирование выполняется в контроллере прерываний.
InterruptUnmask()	Разрешает обработку прерываний с заданным IRQ конкретным обработчиком прерываний.	
_intr_v86()	Выполняет программное прерывание реального режима для потока, временно переведённого в виртуальный 8086 режим.	Даёт возможность доступа к функциям ROM BIOS, предназначенным для выполнения в 16-ти разрядном реальном режиме. Функции BIOS, требующие для своего выполнения аппаратных прерываний, не поддерживаются.

Обработчики ISR работают в своем множестве приоритетов (см. табл. 1.1). Множества приоритетов программных потоков и ISR не пересекаются, любой приоритет ISR выше приоритета любого программного потока. Аналогично потокам, ядро осуществляет вытесняющую диспетчеризацию ISR на основе приоритетов.

1.4.1. Виды уведомлений о прерываниях

В качестве уведомления может быть использован один из следующих типов:

I. Импульс (`SIGEV_PULSE`). Для приёма импульса применяются системные вызовы `MsgReceive*()`, `MsgReceivePulse*()`. Все уведомления о прерываниях ставятся в очередь и обрабатываются в соответствии с приоритетами импульсов. Несколько потоков могут ожидать импульса, что делает эту схему весьма гибкой.

II. Сигнал (`SIGEV_SIGNAL`, `SIGEV_SIGNAL_CODE`, …). Применяется в паре с одной из многочисленных библиотечных функций (`pause(), Sigwait(), SignalSuspend(), SignalWaitinfo()`,…), предназначенных для приёма и обработки сигналов. Имеется возможность выбрать, какие уведомления ставить в очередь на обработку. В сочетании с `sigwaitinfo()` работает несколько быстрее, чем импульс [26].

III. Событие-прерывание (`SIGEV_INTR`). Самый простой и быстрый способ. Применяется вместе с системным вызовом `InterruptWait()`. Только один поток может ожидать это событие - тот, который вызвал `InterruptAttach()` или `InterruptAttachEvent()`. Ядро не ставит в очередь более одного уведомления.

Различные сочетания функций-обработчиков прерываний по п. **А** и **Б**, схем уведомления по пп. **I**, **II**, **III**, системных вызовов для маскирования/ демаскирования - `InterruptMask()`, `InterruptUnmask()` и запрещения/ разрешения `InterruptLock()`, `InterruptUnlock()` прерываний позволяют гибко подстраиваться под особенности аппаратных средств и нужды конкретного приложения, обеспечивая быструю реакцию на

прерывания с возможностью обработки очередей уведомлений о прерываниях.

1.4.2. Использование InterruptAttach()

Синтаксис вызова ядра следующий:

```
int InterruptAttach (
  int intr,          /*логический номер вектора прерывания, т.е. IRQ*/
  const struct sigevent * (* handler)(void * area, int id),//адрес ISR
  const void * area,   /*указатель на  коммуникационную область
                         между ISR и процессом, в котором ISR за-
                         регистрирован */
  int size,            /*размер коммуникационной области*/
  unsigned flags ).    /*Флаги, задающие дополнительные
                         режимы/*.
```

Функция-обработчик прерывания *handler* возвращает событие, которое должно существовать и после завершения работы ISR. Для передачи события процессу или потоку служит коммуникационная область. В качестве альтернативного способа можно описать событие в пределах ISR как статическое или как глобальную переменную, в этих случаях в качестве параметра area передаётся `NULL`, size = 0. Преимущество использования “области устойчивых данных” *area* [1] состоит в возможности применить один ISR для различных прерываний, передавая контекст как параметр. Возможность использования глобальных переменных связана с тем, что “куча” ISR отображается в адресное пространство процесса. Второй параметр функции *handler* – идентификатор “привязки прерывания”, возвращаемый самим вызовом `InterruptAttach()`.

Флаг `_NTO_INTR_FLAGS_END` позволяет поместить ISR в конец цепочки обработчиков, разделяющих общее прерывание вместо начала цепочки по умолчанию. Использование флага даёт возможность в некоторой степени упорядочить вызовы ISR для общего разделяемого прерывания. Флаг `_NTO_INTR_FLAGS_PROCESS` связывает ISR с процессом вместо потока по умолчанию. С этим флагом обработчик удаляется системой при завершении процесса независимо от состояния (“жив/мёртв”) потока, вызвавшего `InterruptAttach()`.

Флаг _NTO_INTR_FLAGS_TRK_MSK обеспечивает правильную работу пары функций InterruptMask()/InterruptUnmask(). Количество вызовов обеих функций должно быть одинаковым. Флаг заставляет ядро считать, сколько раз было замаскировано прерывание с тем, чтобы после отсоединения ISR от источника прерываний ядро нужное количество раз выполнило размаскирование, если это не было сделано при работе пользовательской программы. Этот флаг рекомендуется устанавливать всегда.

В зависимости от конкретной ситуации и требованиям к системе ISR может:

– выполнить сам всю работу по обработке прерываний, включая очистку источника, если это занимает мало времени. В этом случае он возвращает NULL;

– или после выполнения необходимых быстрых действий, включая очистку источника, запустить дальнейшую более длительную обработку прерываний на уровне специально предназначенного для этого потока, вернув ему заданное событие. Таким событием может быть импульс, сигнал или специальное событие-прерывание SIGEV_INTR, разблокирующее ожидающий на InterruptWait() поток (см. пп. 1.2.1).

ISR выполняется в пространстве ядра на сверхвысоком приоритете асинхронно по отношению к потоку, вызвавшему InterruptAttach(). Поэтому любые изменяемые в ISR переменные должны быть объявлены как volatile, что указывает компилятору не помещать их в кэш, поскольку они могут быть изменены в любой точке выполнения программы. Так как. сам ISR может быть прерван другим ISR, обслуживающим более приоритетное прерывание, необходимо гарантировать непрерываемость любых операций с внутренними переменными ISR. Для этого следует либо временно запретить прерывания (InterruptLock(), InterruptUnlock()), либо использовать атомарные операции (функции atomic_*() библиотеки, описанные в <atomic.h>).

Если аппаратная часть компьютера настроена на разделение одного прерывания несколькими устройствами, микроядро само выдаёт контроллеру прерываний сигнал EOI после завершения последнего из драйверов устройств в цепочке (неважно, какую схему обработки использует драйвер – с ISR или без него). Код пользовательской про-

граммы не должен содержать EOI (См. раздел справочной системы QNX/Neutrino “Writing an Interrupt Handler”).

Жёсткая связь ISR с ядром накладывает ряд ограничений на использование этого способа обработки прерываний.

Во-первых, ISR увеличивает размер единой точки отказа (SPOF - **S**ingle **P**oint **O**f **F**ailure) QNX/Neutrino [26]. Микроядерная в истинном смысле этого слова ОС имеет единую точку отказа – микроядро. Ошибки в драйверах и других системных и пользовательских программах могут привести к краху соответствующих процессов, но не системы в целом – пока живо ядро с менеджером процессов, любая программа может быть перезапущена без перезагрузки системы в целом. Однако если ошибка содержится в ISR, который работает в пространстве микроядра, система целиком может повиснуть, что для ПО реального времени является недопустимым. Сложность выявления ошибок в коде ISR связана с тем, что обычный отладчик для этого не применим [26]. Именно поэтому программно обрабатывать аппаратные прерывания имеет право не любой пользователь, а только с правами root. Только программы, запущенные под учётной записью root, имеют право задавать привилегии (I/O Privity) программному потоку обращаться к портам ввода/вывода аппаратуры и вызывать функции обработки прерываний.

Во-вторых, ISR работает в очень ограниченном окружении [Кёртен]:

– Не может использовать вызовы ядра, за исключением очень ограниченного количества:
  * `InterruptEnable()`
  * `InterruptDisable()`
  * `InterruptLock()`
  * `InterruptUnlock()`
  * `InterruptMask()`
  * `InterruptUnmask()`.
– Не может использовать любые библиотечные функции, которые могут обращаться к ядру. Для каждой функции библиотеки справка содержит сведения о допустимости использования этой функции в ISR;
– Не может выполнять операции с плавающей точкой.

Отмеченные ограничения на ISR отнюдь не означают, что альтернативный вариант обработки прерываний в QNX/Neutrino с

InterruptAttachEvent() (см. пп. 1.2.3) более хорош, чем рассматриваемый с InterruptAttach(). На выбор влияет много факторов, анализ которых сделан в [1] и частично разобран ниже.

Разработчики ОСРВ QNX/Neutrino рекомендуют использовать комбинацию Б-III в качестве очень простого, элегантного и быстрого способа обработки прерываний (См. раздел справочной системы QNX/Neutrino “Writing an Interrupt Handler”).

Общая структура программного кода в этом случае выглядит следующим образом.

```
// Это обработчик прерываний ISR
const struct sigevent * isr_handler (void *arg,
int id)
{
     /* Проверить, вызвано ли прерывание интересующим нас
        устройством. Если нет – просто возвратить (NULL);
        Если прерывания активны по уровню – очистить источник
        прерываний, или, по меньшей мере, выполнить
        InterruptMask()чтобы лишить возможности контроллер
        прерываний заново прервать ядро*/
      ...
     /* Возвратить указатель на структуру события struct
        sigevent (инициализированную в main()), которая
        содержит SIGEV_INTR в качестве типа уведомления. Это
        вызовет разблокирование InterruptWait() в потоке
        “int_thread"

     return (&event);
}
// Этот поток предназначен для обработки и управления прерываними
void * int_thread (void *arg)
{
    // Предоставить привилегии ввода/вывода в регистры устройства
     ThreadCtl (_NTO_TCTL_IO, NULL);
     ...
     // инициализировать аппаратуру
     ...
     // Присоединить обработчик прерываний к IRQ устройства
```

```
        InterruptAttach (IRQ, isr_handler, NULL,0,0);
        ...
        // При необходимости задать здесь приоритет потока
        ...
        // Теперь обслуживать аппаратуру по указанию ISR
        while (1)
        {
             InterruptWait (NULL, NULL);
            /* В этом месте, когда InterruptWait()разблокируется,
            ISR возвратил SIGEV_INTR, указывающее, что должна
            быть выполнена некоторая работа */
             ...
            // выполнить эту работу
             ...
            /* Если isr_handler замаскировал прерывания, вызвав
               InterruptMask(), тогда поток должен выполнить
               InterruptUnmask() чтобы  разрешить прерывания от
               аппаратуры*/
        }
}
main ()
{
        // Выполнить  инициализацию и другие необходимые действия
        ...
        // Запустить поток, предназначенный для обработки прерываний
        pthread_create (NULL, NULL, int_thread, NULL);
        ...
        // Выполнять другие необходимые действия
        ...
}
```

К преимуществам этого способа относятся:

- В приложении не требуется создавать канал и вызывать `MsgReceive()` для приёма уведомления-импульса.
- Приложение не нуждается в функции обработки уведомления-сигнала.
- Если обслуживание прерывания критично по времени, может быть использована нить `int_thread()` с высоким приорите-

том. Это позволит ей запускаться немедленно после того, как событие `SIGEV_INTR` возвращено из функции `isr_handler()`. Здесь невозможна задержка по причине того, что после посылки импульса обработчиком ISR другая нить случайно выполнит `MsgReceive()` и примет импульс.

Недостатком использования `InterruptWait()` является то, что только один присоединённый к прерыванию поток может ожидать события `SIGEV_INTR`.

Изложенный способ обработки прерываний реализован в программе 3-ser_rd-int1-lab.c (см. п. 2.1). Пример обработки прерываний аппаратного таймера с использованием `InterruptAttach()` привёдён также в разделе “Inteppurt handling” справке по микроядру справочной системы QNX Neutrino.

1.4.3. Использование InterruptAttachEvent()

Функцию `InterruptAttach()` в связи с наличием сверхприоритетного ISR целесообразно использовать тогда, когда нужно обращаться к аппаратным средствам непосредственно после прерывания. В остальных случаях преимуществом обладает функция `InterruptAttachEvent()`. Её использование принуждает само ядро генерировать заданное событие (см. п. 1.2.1) при возникновении прерывания и затем маскировать соответствующее прерывание для предотвращения его рекурсивного вызова в аппаратуре, чувствительной по уровню. Принявший событие-уведомление поток после обработки прерывания и очистки его источника должен размаскировать прерывание с помощью функции `InterruptUnmask()`.

Преимущества использования `InterruptAttachEvent()` следующие:

– Минимальное время, проводимое процессором в пространстве ядра.
– Отсутствие ISR и, соответственно, необходимости его сложной отладки.
– Встроенная “фильтрация” высокочастотных прерываний за счёт маскирования прерывания перед запуском обрабатывающей его нити и размаскирование после завершения обработки.

Недостатки применения `InterruptAttachEvent()` связаны с возможным замедлением реакции системы из-за:

– Задержки выполнения других драйверов устройств, разделяющих замаскированное прерывание.
– Увеличения времени задержки планирования потоков (scheduling latency).

Целесообразность выбора одного из двух вариантов присоединения прерывания к обрабатывающему программному коду может зависеть и от наличия или отсутствия принимающего/передающего буфера в устройстве. Так, UART с отключённым буфером будет надёжнее работать с `InterruptAttach()`, иначе возможная задержка обработки прерывания по приходу символа может вызвать потерю символа – "забой" его следующим пришедшим. В то же время сетевая карта с большим буфером "на борту" может надёжно работать с `InterruptAttachEvent()`.

Структура программного кода обработки прерываний на основе `InterruptAttachEvent()`следующая:

```
void *interrupt_thread (void * data)
/* Нить для обработки прерываний. Она требуется только если основная программа должна выполнять вычисления, не связанные с прерываниями.  В противном случае достаточно только main(). */
{
    struct sigevent event;
    int  id;

  /* Заполнить структуру "события" */
    memset(&event, 0, sizeof(event));
    event.sigev_notify = SIGEV_INTR;

  /* intNum – требуемый уровень прерываний */
    id = InterruptAttachEvent (INTNUM, &event, 0);

  /*... Здесь размещается необходимый код ... */

    while (1) {
        InterruptWait (NULL, NULL);
     /*  Выполнить действия в ответ на прерывание. После этого
      * снять с прерывания маску */
        InterruptUnmask(INTNUM, id);
```

```
    }
}

int main(int argc, char **argv) {
      /* ….*/
  /* Запустить нить, предназначенную для обработки прерываний*/
     pthread_create (NULL, NULL, interrupt_thread,
                          NULL);
      /* ….*/ }
```

Пример использования функции `InterruptAttachEvent()` иллюстрирует программа 3-par_rd-int2-lab_4.c (см. п.2.2).

Предпочтительность применения каждой из двух рассмотренных функций – `InterruptAttach()` и `InterruptAttachEvent()` зависит от многих факторов, возможные сценарии и рекомендации разобраны в [1]. Существенным при этом является обеспечение минимального времени реакции на прерывания системы целиком и отдельных устройств, чувствительных к интервалу от момента выставления прерывания до отработки ISR.

Если обработка прерываний в приложении больше не требуется, нужно отсоединить источник прерываний от обработчика с помощью системного вызова `InterruptDetach()`. Для цепочки обработчиков разделяемого прерывания удаление одного из обработчиков не влияет на взаимное положение других обработчиков и событий.

1.4.4. Прерывания и временно́е поведение системы

Аппаратные прерывания вмешиваются в нормальный ход выполнения программ, забирая ресурсы системы на соответствующую обработку и замедляя работу программ. В свою очередь сама скорость обработки прерываний, характеризующая реакцию на внешние события, зависит от программного окружения, в том числе от свойств самой операционной системы. С точки зрения требований к системам реального времени это - критически важные вещи. Поэтому связи аппаратных прерываний и времени в СРВ уделяется особое внимание.

1.4.4.1. ВременнЫе характеристики обработки прерываний

Обработка прерываний требует определённого времени работы ядра и, дополнительно,:

– работы функции-обработчика ISR на сверхвысоком приоритете,

– либо обрабатывающего прерывание потока,
– или ISR и потока-обработчика совместно.

Затраты времени на обработку прерываний важны для реактивных, отвечающих на внешние события систем реального времени. Разработчики ОС РВ QNX рассматривают две характеристики, описывающие временно́е поведение системы обработки прерываний.

Первая из характеристик носит название scheduling latency - промежуток времени между последней инструкции пользовательского ISR и первой инструкцией потока-драйвера. Scheduling latency в случае вытесняемого прерывания иллюстрируется рис.1.2 [26]. В момент времени **2** поток **A** вытесняется ISR **B** 1-го по времени прерывания, который, в свою очередь, в момент **3** вытесняется обработчиком **C** 2-го более приоритетного прерывания. Закончив работу в точке **4**, ISR **C** запускает поток-обработчик D для завершения обработки второго по счёту прерывания. Однако в момент **5** поток **D** не начинает выполняться, так как процессор занимает прерванный ISR **B** сверхвысокого по отношению к **D** приоритета. D получит процессор только по завершении ISR **B.** Sheduling latency между моментами времени **4** и **7** включает загрузку процессора как выполнением кода ISR **B**, так и кода микроядра.

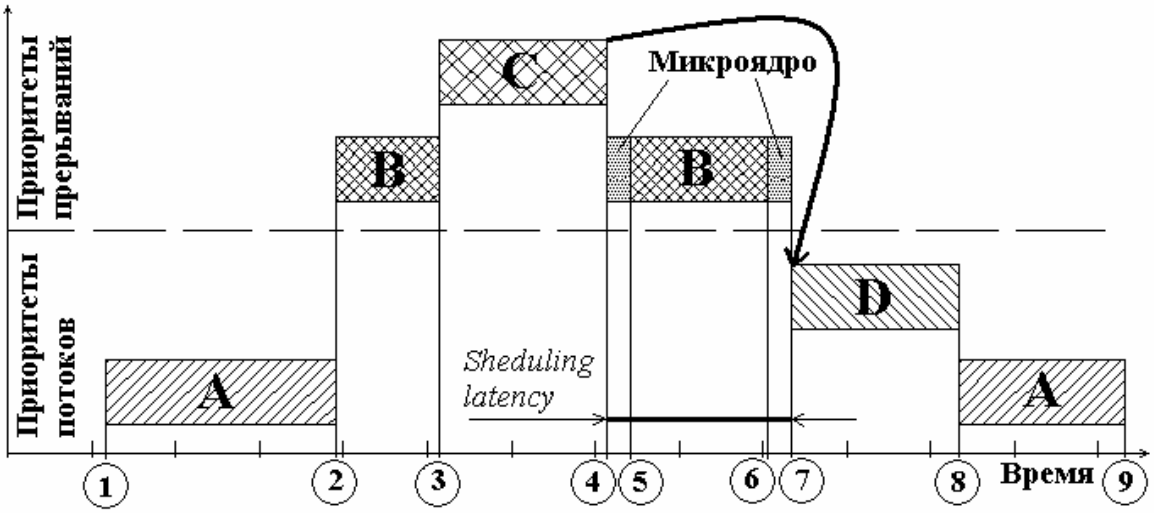

Рис.1.2. Задержка диспетчеризации (scheduling latency) вытесняемого прерывания

В общем случае на sheduling latency влияют:

– время, потраченное на выполнение других ISR;
– приоритет запускаемого потока-обработчика;

– время выполнения потоков с более высоким приоритетом, которые перед запуском потока-обработчика находятся в состоянии готовности (Ready) [26].

Вторая временна́я характеристика обработки прерываний - промежуток времени от возникновения аппаратного прерывания до начала выполнения первой инструкции ISR драйвера устройства - interrupt latency. Иллюстрация приведена на рис.1.3 для случая невытесняемого прерывания. В точке **2** поток вытесняется ISR **B**. Интервал времени между точками **2** и **3** тратится в простейшем случае на сохранение контекста потока **A** и подготовку к выполнению **B**. Поток **A** возобновит работу в точке **6** только после завершения цепочки ISR **B** и запущенного ISR **C** потока-драйвера **D**.

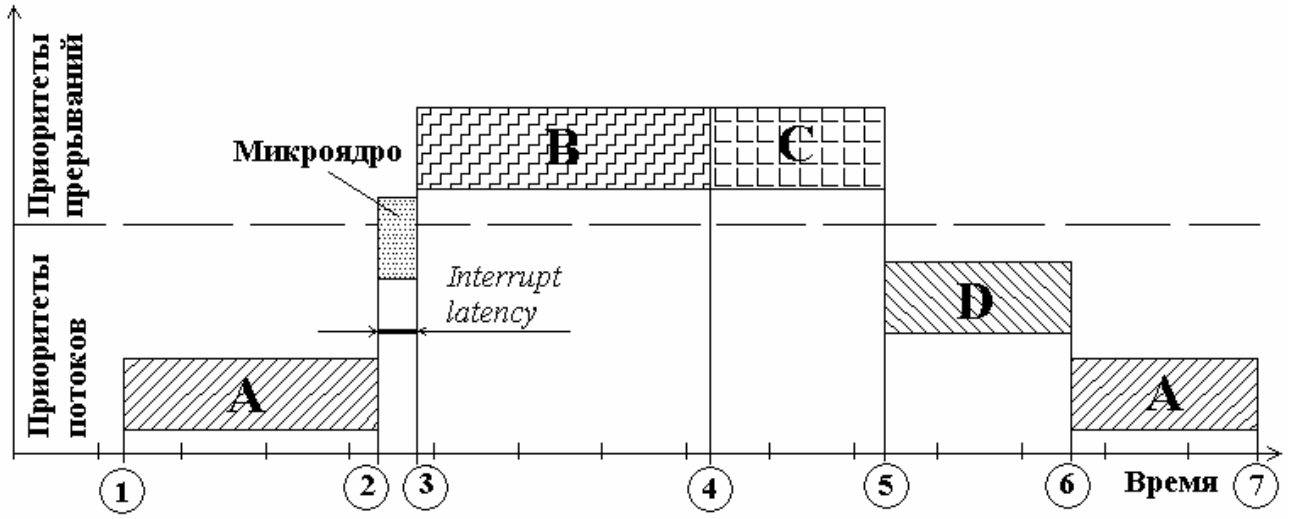

Рис.1.3. Interrupt latency для невытесняемого прерывания

В общем случае interrupt latency складывается из:

– времени, в течение которого прерывания запрещены;
– времени выполнения ISR большего приоритета;
– времени, потраченного обработчиками в цепочке для этого IRQ;
– времени, в течении которого этот уровень прерывания маскирован (типично для обработки разделяемого прерывания с использованием `InterruptAttachEvent()`) [26].

Учитывая важность быстрой и предсказуемой по времени обработки прерываний системами реального времени, эксперты Dedicated Systems Experts [27] включили соответствующее требование в набор пяти основных требований к операционным системам реального времени: “Our fourth requirement is that interrupts have a fixed upper bound on latency, and support for nested interrupts” (“наше четвёртое требова-

ние состоит в том, чтобы прерывания имели фиксированную верхнюю границу задержки, и поддержка вложенных прерываний“). По мнению экспертов, для того, чтобы операционная система могла рассматриваться как ОС РВ, производитель ОС обязан обеспечить предсказуемость и сообщить разработчикам ПО основные характеристики ОС, включающие [28]:

– interrupt latency;
– наибольшее время, в течение которого ОС и драйверы маскируют прерывания;
– уровни прерываний в системе (System Interrupt Levels);
– IRQ драйверов устройств и наибольшее время, которое затрачивается драйверами на обработку прерываний.

1.4.4.2. Результаты тестирования систем обработки прерываний различных ОС

Проводя тестирование ОС РВ, эксперты определяют наименьшие, наибольшие и средние значения следующих временных характеристик работы системы прерываний конкретных ОС [29]:

- Время ожидания обработки прерывания Interrupt Latency, сокращённо IL – интервал времени от возникновения прерывания до начала выполнение ISR или потока-обработчика прерывания, работающего с наивысшим приоритетом (“the time it takes to switch from a running thread to an interrupt service thread (running at the highest priority level)”);
- Время обработки прерывания Interrupt Dispatch Latency, сокращённо IDL – интервал времени от завершения обработчика прерывания до возобновления выполнения приостановленной прерыванием нити (“the time it takes to switch from the Interrupt Service Routine back to the interrupted thread. The measurement starts in the ISR itself and stops when the interrupted thread gets control again”);

– Время ожидания обработки высокоприоритетного прерывания в случае двух одновременно возникающих прерываний (Simultaneous interrupts. Interrupt latency of the high priority ISR);
– Время ожидания обработки низкоприоритетного прерывания в случае двух одновременно возникающих прерываний (Simultaneous interrupts. Interrupt latency of the low priority ISR).

Измерения проводятся в следующих сценариях работы тестируемой системы [15]:

– Одиночные прерывания, не приводящие к перепланировке заданий;
– Одиночные прерывания с перепланировкой заданий ;
– Одновременное возникновение двух прерываний;
– Вложенные прерывания;
– Максимальная поддерживаемая частота прерываний, не приводящая к их потерям.

Некоторые результаты тестирования приведены в табл. 1.3, 1.4, 1.5.

Таблица 1.3

Время ожидания обработки прерывания (IL) и время обработки прерывания (IDL) на уровне ISR, мкс.[30,31]

Характеристика \ ОС	QNX 6.1		QNX 6.2		VxWorks AE 1.1		Windows CE.NET		ELDS* v.1.1	
	Сред.	Макс.	Сред.	Макс.	Сред.	Макс.	Сред.	Макс.	Сред.	Макс.
IL	1.7	4.1	1.7	4.3	1.7	6.8	2.4	5.6	3.2	4.0
IDL	1.8	2.1	1.9	2.7	1.9	8.4	–	–	1.2	2.0

Примечание. *Red Hat Embedded Linux Developer Suite (ELDS) version 1.1. Разработан фирмой Red Hat на базе ядра Red Hat 7.2 (вариант ядра 2.4.5 Linux).

Тест табл.3 измеряет время ожидания обработки прерывания при сильно загруженной системе. Для этого теста была создана высокоприоритетная нить, устанавливающая обработчик прерываний. Прерывания с помощью специальной PCI-платы расширения с собственным процессором генерировались приблизительно каждые 50 мкс и прерывали работу нити. Каждый раз при возникновении прерывания запускался их обработчик ISR, писал отсчёт на PCI шину и завершался. После завершения обработчика прерываний, прерванная нить возобновляла работу. Частота прерываний такова, что их обработка занимает всю доступную мощность ЦП. Система не перепланировалась. Шина лабораторного компьютера была соединена с аппаратным анализатором шины PCI Отсчёты в аппаратном анализаторе шины PCI снабжались точными временными метками и записыва-

лись в память анализатора. После выполнения заданного числа циклов временные метки пересылались в компьютер и статистически обрабатывались.

Таблица 1.4

Наименьший интервал времени Δ между периодически повторяемыми прерываниями, при котором ОС успевает обрабатывать все прерывания без их потери, мкс [30,31]

Класс ОС	Название ОС	Δ
ОС РВ	QNX/Neutrino 6.1	10
	QNX/Neutrino 6.2	9
	Microsoft Windows CE .NET	11
	Microsoft Windows AE 11	25
ОС общего назначения	ELDS* v.1.1	60

Примечание. *Red Hat Embedded Linux Developer Suite (ELDS) version 1.1. Разработан фирмой Red Hat на базе ядра Red Hat 7.2 (вариант ядра 2.4.5 Linux).

В тесте табл.4 генерируется миллиард (10^9) прерываний (на одном и том же уровне IRQ) на программно изменяемой частоте и затем подсчитывается, сколько прерываний было обслужено и сколько потеряно. Прерывания обслуживаются на уровне ISR. Тест считается успешным, если ни одно прерывание не теряется. Прерывание “потеряно”, когда ко времени генерации следующего прерывания предыдущее прерывание все еще обслуживается или замаскировано прерыванием с более высоким приоритетом (например, прерыванием от часов). Таким образом, этот тест дает довольно хорошее представление о времени реакции на прерывание в ОСРВ в худшем случае. Но необходимо помнить, что данный тест можно считать стресс–тестом не только самой RTOS, но и аппаратного оборудования PC. Таким образом, никаких определенных суждений о том, какая именно часть системы виновна в потере прерываний, вынести нельзя. Но так как в тестах сравниваемых операционных систем используется одно и то же оборудование, результаты тестирования различных ОС могут быть

сопоставлены между собой. Тест более короткой продолжительности – 1000 прерываний, в течение которых вероятность потерь прерываний, вызванных аппаратурой, пренебрежимо мала, позволяет выявить ответственность самой операционной системы за пропуск прерываний. Для QNX/Neutrino 6.2 Δ составляет 5.5 мкс.

Таблица 1.5

Промежуток времени (interrupt latency) от возникновения прерывания до начала его обработки обработчиком прерываний, мкс операционной системы QNX/Neutrino 6.2, мкс. [15]

Приоритеты двух одновременно возникающих прерываний	interrupt latency		
	Минимальное значение	Среднее значение	Максимальное значение
Высокий (IRQ9)	1.6	1.6	2.5
Низкий (IRQ10)	4.0	4.1	4.9

В тесте табл.1.5 генерировались два одновременных (почти) прерывания. Это было реализовано путем добавления в систему второй программы генерирования прерываний PCI (PDrive). Разница во времени возникновения прерываний не превышала 100 нс, что по определению классифицируется как одновременное возникновение прерываний:

– PDrive_HI генерирует высокоприоритетные PCI-прерывания по линии C (INTC#) с уровнем IRQ 9.
– PDrive_LO генерирует низкоприоритетные PCI-прерывания по линии B (INTB#) с уровнем IRQ 10.

Две обрабатывающие прерывания процедуры (ISR_HI and ISR_LO) сопоставлены уровням IRQ 9 и 10 соответственно. Время ожидания обработки прерываний оценивалось и для ISR_HI, и для ISR_LO. Время ожидания обработки прерывания измерялось как время между выполнением последней команды прерываемой нити и первой команды ISR.

Результаты сравнительных тестов показывают, что реализация службы прерываний ОС QNX/Neutrino весьма эффективна. Есть небольшое ухудшение производительности при обработке прерываний

версии 6.2 по сравнению с 6.1, но синхронные и вложенные прерывания обрабатываются немного быстрее [15].

При рассмотрении результатов тестов для целей оценки временно́го поведения конкретной проектируемой системы реального времени важно отдавать себе отчёт, что экспертные тесты проводились в специально созданном программном окружении, с помощью специальной аппаратуры и методик [29]. Специальные метки, ограничивающие участок кода, время выполнения которого замеряется, выставляются на шину PCI и записываются в локальную память анализатора шины PCI (PCI bus analyzer). По завершении теста временные метки загружаются в компьютер. Полученные путем вычитания времён записи двух соседних меток интервалы затем анализируются с помощью статистических инструментов. Использование внешнего устройства - анализатора шины PCI – позволяет устранить непредсказуемое влияние операционной системы при замерах интервалов с помощью программных таймеров самой ОС. Повторить эти замеры, не имея соответствующих аппаратных средств, невозможно. Вот мнение участника форума http://qnx.org.ru/forum Evgeniy по поводу попыток замера interrupt latency на конкретной системе: “Я долгое время очень тесно общался с разработчиками ОС РВ - ДОС АСПО. Так вот они утверждали, что разработка методов измерения характеристик системы была вполне сопоставима по сложности с разработкой компонентов средней сложности самой системы. Мой личный опыт в этой области только подтверждает мысль о том, что измерение временных характеристик системы, если их проводить корректно, является очень серьезным самостоятельным исследованием. И оценивать результаты отдельных измерений практически не имеет смысла”. Как правило, для реальной компьютерной, не лабораторной, системы, результаты замеров временнЫх характеристик обработки прерываний дают значения, значительно превышающие тестовые, и не только из-за погрешностей способов и средств измерения, но и по причине существенного влияния прерываний всего комплекса аппаратуры. В частности, на форумах отмечается значительный вклад в общее время обработки прерываний контроллеров жёстких дисков и сетевых карт.

1.4.5. Прерывания и драйверы устройств

Многие классы устройств используют прерывания как средство уведомления системы об изменении своего состояния. В ответ на прерывания запускаются драйверы, выполняющие необходимую ра-

боту по обслуживанию устройств. В отличие от ОС с монолитным ядром, где драйверы являются частью самого ядра, в микроядерной ОС QNX/Neutrino драйверы представляют собой отдельные процессы, каждый из которых может быть запущен и перезапущен по необходимости без перекомпиляции/перезагрузки всей ОС.

Обработка аппаратных прерываний в драйверах, как поставляемых в составе ОС, так и разрабатываемых пользователями, осуществляется с использованием функций библиотеки QNX/Neutrino (см. табл. 2). С точки зрения реактивных систем реального времени драйвер должен обеспечить обслуживание вызвавшего прерывание устройства таким образом, чтобы в минимальной степени задержать обслуживание прерываний других устройств, в особенности тех, которые завязаны на выполняемые системой задачи с жёсткими временными ограничениями. Приоритеты запросов аппаратных прерываний (см. табл.1) лишь частично помогают решить эти вопросы, так как:

– не любой IRQ может быть предоставлен устройству по причине конкуренции за линии прерываний со стороны других устройств;
– не каждое устройство способно нормально работать с любым свободным IRQ;
– самые приоритетные запросы прерываний на платформе x86 жёстко закреплены за аппаратным таймером (IRQO) и клавиатурой (IRQ1) и не могут использоваться для других целей.

Дополнительную проблему вносит разделение прерываний, когда недостаточно свободных линий прерываний для всех периферийных устройств (см. пп. 1.3.1, 1.3.2). Поэтому разработчики драйверов должны уделять особое внимание архитектуре своих программ, выбрав наилучшую для конкретного устройства схему обработки прерываний – на уровне ISR (`InterruptAttach()`), на уровне потока-обработчика (`InterruptAttachEvent()`) или их комбинации.

Неправильное программирование драйверов устройств встречается нередко и для систем реального времени является критичным вопросом, способным сделать неработоспособной систему, хорошую во всех других отношениях. “Обработчики прерываний устройств должны вести себя корректно, учитывая возможность попадания в цепочку обработчиков разделяемого прерывания. В процессе обработки прерывания очередной обработчик в цепочке чтением известного ему регистра своего устройства должен определить, не это ли устройство вызвало прерывание. Если это, то обработчик должен вы-

полнить необходимые действия и сбросить сигнал запроса прерывания от своего устройства, после чего передать управление следующему обработчику в цепочке; в противном случае он просто передает управление следующему обработчику. Встречается типичная ошибка обработчика прерываний: прочитав регистр состояния устройства и не обнаружив признака запроса, драйвер «на всякий случай» выполняет сброс всех источников запроса (а то и сброс всего устройства). Эту ошибку порождает незадачливый разработчик драйвера, не учитывающий возможности разделяемости прерываний и не доверяющий разработчикам аппаратных средств. Увидев в процессе отладки эту неожиданную ситуацию (прерывание вызвано, а источник не виден), он ее «учитывает» введением вредного фрагмента программного кода. Вредность заключается в том, что с момента чтения регистра устройства (не давшего признака запроса) и до выполнения этого ненужного сброса в устройстве может возникнуть запрос прерывания, который будет «вслепую» сброшен и, следовательно, потерян.

Однако и при корректности обработчиков, выстроенных в цепочку, разделяемые прерывания для разнотипных устройств в общем случае работоспособными считать нельзя: возможны потери прерываний от устройств, требующих быстрой реакции. Это может происходить, если обработчик такого устройства окажется в конце цепочки, а предшествующие ему обработчики окажутся «нерасторопными» (не самым быстрым способом обнаружат, что прерывание чужое). Поведение системы в такой ситуации может меняться в зависимости от порядка загрузки драйверов. Для нескольких однотипных устройств (например, сетевых адаптеров на однотипных микросхемах контроллеров), пользующихся одним драйвером, разделяемые прерывания работают вполне успешно” [5].

Проблема с драйверами не обошла стороной и ОС QNX. Участник форума http://qnx.org.ru/forum Lestat делится опытом: “…Бывают кривые драйвера железа, которые, например, получая прерывание не для себя, не проверяют (бывает не сделано аппаратно), а их ли это прерывание и начинают думать, как будто им оно пришло лично. В результате - задержка доставки прерывания для остальных устройств и, бывает, срывает крышу первому устройству”. Большую проблему представляет отсутствие информации о том, как запрограммирован драйвер. Это приводит к увеличению неопределённости в поведении системы, когда разработанное пользовательское ПО с известными ха-

рактеристиками кооперируется с непредсказуемым для программиста из-за отсутствия сведений о нём ПО драйверов. Уже на этапе проектирования нарушается основное требование к системам реального времени – предсказуемость.

Поскольку аппаратные прерывания генерируются “железом”, для правильной работы очень важна продуманная и безошибочная реализация устройств. К сожалению, как показывает практика, конструктивные недоработки встречаются нередко. Lestat высказывает две практически важных рекомендации для случая, когда есть подозрение, что “кривые” железо и/или драйвер тормозят работу системы:

– “если есть подозрения на железку, замени ее другой совершенно на другом чипе, другого производителя, обычно все начинает себя вести немного по-другому, если все по-старому, то дело не в этой железке. Кроме проб и ошибок - другого способа не придумали.”
– “… к RTOS еще нужен и RTHW. Тут всякой китайской дешевой фигней не обойдешься ...” { RTHW - Real Time HardWare}.

Среди практических программистов сформировалось мнение, что QNX/Neutrino не вполне правильно реализует спецификацию шины PCI из-за ошибок в PCI-сервере [21], а также разделение прерываний. Lestat отмечает в теме форума http://qnx.org.ru/forum про обработку прерывания нестандартного устройства: “Как показала практика, IRQ routing - т.е. последовательность вызова прерываний для каждого из устройств на общем прерывании - ядро QNX'a не делает как надо... BIOS делает IRQ роутинг для устройств и прерываний, только которые он знает, например USB любит на 12 садиться. А до нашего устройства прерывание, например, после PS/2 (12 IRQ) мышки просто не доходит. Дело в QNX'е - там пахабно сделана поддержка прерываний в этом плане”.

Много нареканий со стороны пользователей вызывают драйверы сетевых карт, в основном из-за больших задержек обработки прерываний других устройств, разделяющих с сетевой картой общую линию прерываний. Содержательные дискуссии о драйверах состоялись в 2004 г. и июне 2005 г. в группах новостей inn.qnx.com [32,33]. Некоторые тезисы обсуждения, полезные с точки зрения понимания проблем обработки прерываний и возможных способов их решения, приведены ниже.

Радикальное решение – выделить для сетевой карты отдельную линию прерывания, не используемую более никаким устройством.

Практически это сделать невозможно, так как все PCI-слоты спарены по линиям прерываний с к.-л. встроенными чипами материнской платы. Производителей BIOS материнских плат, используемых в системах реального времени, следует попросить реализовать максимально возможный ручной контроль за распределением линий прерываний по устройствам.

Подробная документация по драйверам, входящим в поставку QNX/Neutrino, отсутствует. В частности, не документированы приоритеты. Возможность управления поведением драйверов в сочетании с другими программами со стороны программиста, создающего конечный программный продукт, мала или отсутствует вовсе. Отсюда неприятные “сюрпризы” с большими значениями interrupt latency и т.п.

По-видимому, большинство драйверов сетевых карт используют функцию `InterruptAttachEvent()`. Схема обработки прерываний на основе `InterruptAttach()` применяется редко, т.к обработка прерывания в сетевой карте требует довольно большого объёма операций, а на время их выполнения в сверхприоритетном ISR вся остальная деятельность системы на уровне потоков будет приостановлена. Кроме того, для некоторых, преимущественно старых сетевых адаптеров, схема на основе ISR не применима в принципе.

Функция `InterruptAttachEvent()` маскирует в ядре прерывания с соответствующим IRQ и посылает блокированному потоку-обработчику (Interrupt Handling Thread – IHT) событие. Получив его, IHT обрабатывает прерывание, снимает маску и завершается. Аппаратная реализация многих сетевых карт такова, что пока не обработаны все данные, прерывание не может быть размаскировано, что значительно увеличивает interrupt latency разделяющего ту же линию прерываний устройства. Если устройство имеет внутренний аппаратно реализованный буфер достаточного размера, время маскирования прерываний можно сделать малым. Символьные устройства с внутренними буферами управляются драйверами на основе `InterruptAttach()`, причём ISR выполняет минимальную работу по подсчёту количества символов в буфере, и лишь по заполнении буфера ISR генерирует событие и возбуждает считывание данных из буфера.

`InterruptAttachEvent()` заставляет пересекаться в системе две различные области приоритетов – программных потоков и

прерываний. Прерывание с низким приоритетом может запустить поток-обработчик с высоким приритетом, послав, например, ему высокоприоритетный импульс. Возможна обратная ситуация. Приоритет IHT очень важен. Если он низок, IHT в любой момент может быть прерван более высокоприоритетным потоком. Возникает ситуация, схожая с проявлением инверсии приоритетов. Текущее прерывание перестаёт обрабатываться, все прерывания текущего уровня остаются замаскированными на заранее неопределённое время, что не только ухудшает реакцию системы, но и может привести к потере прерываний.

Правильное назначение и при необходимости динамическое управление приоритетами потоков-обработчиков, входящих в состав драйверов, с учётом приоритетов прерываний устройств, занятых задачами жёсткого реального времени, является сложной проблемой. В таких системах, в отличие от настольных персональных компьютеров, полагаться целиком на заложенные по умолчанию в операционной системе приоритеты нельзя. В сложных системах от программиста требуется работа на уровне искусства, причём наличие информации о приоритетах драйверов обязательна.

Если система не занимается ничем, кроме обработки прерываний, лучшие результаты по быстродействию и детерминированности даст программирование железа напрямую, не используя операционную систему. Вот мнение участника форума qnx.org.ru/forum Ed1k: “Вообще-то опыт подсказывает, что если требования столь жестки, что приоритетность прерываний начинает волновать, то надо отказываться от PC/AT, PCI и QNX4. А если нету таких жестких требований, то и не морочить себе голову такими пустяками. В шестерке (QNX6) исходники начальной загрузки продаются и правкой кода можно добиться любого чуда от PIC”.

В случае `InterruptAttachEvent()` работа IHT может состоять из двух фаз:

- быстро выполнив самую необходимую работу для того, чтобы можно было снять маску (1ая стадия),
- IHT затем может продолжить обработку (2ая стадия) без торможения новых прерываний.

Аналогично ведёт себя смешанная схема – ISR после минимальной обработки запускает с помощью события IHT, который выполняет остальную работу.

Маскирование прерываний может осуществляться как в ISR, так и в IHT, размаскирование – перед возвратом IHT. Маскирование прерываний плохо совместимо с разделением прерываний. В случае разделяемых прерываний поведение системы не до конца предсказуемо. Недостатка, связанного с возможностью вытеснения IHT, лишена схема, использующая ISR (`InterruptAttach()`), однако, как указано выше, в драйверах сетевых карт она применяется редко.

Чтобы уменьшить задержку разделяемого прерывания драйвер устройства, нужного критичной по времени задаче, следует располагать первым в цепочке. Хорошее решение, если оно может быть реализовано, состоит в маскировке драйвером прерываний на уровне того устройства, которое в данный момент обслуживается, вместо маскирования на уровне контроллера прерываний.

Другой подход связан с управлением приоритетами IHT и остальных потоков в системе. Естественно выбрать приоритет IHT выше приоритета любого другого потока, когда в системе отсутствуют задачи жёсткого реального времени, не связанные с прерываниями. Возможная реализация – на уровне ядра, чтобы оно само запускало IHT с наивысшим приоритетом, однако это требует внесения изменений в микроядро, что для разработчиков прикладных программ недоступно.

В ОС Windows NT реализована приблизительно такая схема: если ISR, выполняемый на уровне приоритета аппаратных прерываний (DIRQLs), решает, что требуется дополнительная обработка, которая может быть сделана на более низком приоритете, он ставит в очередь вызов отложенный процедуры (DPC). DPC будет выполнен позже в порядке очереди с приоритетом DISPATH_LEVEL, который ниже DIRQLs, но выше приоритета любого пользовательского потока в системе. Близкая по смыслу схема реализована и в ОС Linux. Возложение управления приоритетами обработчиков прерываний на ОС хорошо подходит лишь для настольных систем, но не для систем реального времени.

В QNX/Neutrino, если не модифицировать микроядро, можно предусмотреть возможность для пользователя задавать вручную приоритет драйвера при его запуске. Такая возможность уже реализована в некоторых драйверах сетевых карт, входящих в поставку QNX 6.3. В дальнейшее развитие этого подхода участник обсуждения, сотрудник фирмы QSSL Robert Craig, сформулировал запрос на модифи-

кацию драйверов сетевых карт с возможностью задания вручную приоритетов как 1-й, так и 2-й стадий обработки прерываний в IHT. Если устройство монопольно занимает линию прерываний, оба приоритета могут быть низкими. При разделении прерываний выполнение 1-й стадии на наивысшем (255) приоритете займёт несколько сотен наносекунд, что соответствует времени выполнения ISR. Такое малое время маскирования прерываний снимет задержки обработки цепочки разделяющих общее прерывание драйверов. Аналогичное предложение по заданию приоритетов было высказано и по поводу доработки драйверов символьных устройств. По-видимому, единственный недостаток предоставления программисту возможности управления приоритетами драйверов состоит в том, что для сложных систем, обрабатывающих несколько задач жёсткого реального времени, среди которых есть и непериодические, трудно заранее подобрать соотношение приоритетов, оптимальное для всех возможных комбинаций наступления критичных по времени внешних событий, требующих своевременной реакции.

Дополнительным итогом обсуждения темы сетевых драйверов QNX явилось обнаружение неточностей в справке по функции `InterruptEnable()`. Правильно так: “Before calling this function, the thread must request I/O privity by calling:

```
ThreadCtl( _NTO_TCTL_IO, 0 );
```

If the thread doesn't do this, it might SIGSEGV when `InterruptDisable()`(вместо ошибочного `InterruptUnlock()`) is called”.

РЕЗЮМЕ

Программирование реакции на изменение состояния устройств систем реального времени требует от программиста чёткого представления о поведении аппаратной части этих устройств, а также о функционировании аппаратной части системы прерываний вычислительного компонента. Число вариантов реализации программных модулей, “отлавливающих” прерывания и реагирующих на них, велико, особенно с учётом того, что различные типы микропроцессоров имеют значительно отличающиеся системы прерываний, включая особенности присваивания приоритетов, а также того, что архитектура подсистемы прерываний может относиться либо к классу “чувстви-

тельной по уровню”, либо “чувствительной по фронту”. Дополнительные сложности вносит наличие нескольких устройств на одной линии прерываний.

Универсальных, пригодных на все случаи жизни решений по программированию управляемых аппаратными прерываниями систем реального времени не существует. Кажущаяся простота использования в программах предоставляемого QNX Neutrino API обработки прерываний требует большой подготовительной работы по анализу взаимодействия генерирующих прерывания устройств между собой и системой в целом с учётом временных ограничений. При этом надо иметь в виду, что некоторые средства API по обработке прерываний могут существенным образом влиять на функциональность операционной системы. Так, обработчик прерываний, “привязываемый” с помощью системного вызова `InterruptAttach()`, работает в контексте ядра, тем самым потенциально расширяя размер единой точки отказа (Single Point of Failure-SPoF) операционной системы. Ошибки в коде функции-обработчика прерываний могут привести к зависанию микроядра и системы в целом, что для систем реального времени равносильно отказу. Другие средства могут “затормозить” системное время, тем самым потенциально нарушить работу таймеров и временно́е поведение системы, что также является критичным для систем реального времени. Нерациональное программирование драйверов аппаратуры, работающей по прерываниям, может привести к значительной задержке реакции на прерывания от других устройств. Поэтому при разработке программ реального времени следует на обработку аппаратных прерываний обращать особое внимание.

2. ТЕКСТЫ ПРОГРАММ

2.1. ПРОГРАММА ПЕРЕДАЧИ ДАННЫХ МЕЖДУ ДВУМЯ КОМПЬЮТЕРАМИ ПО ПОСЛЕДОВАТЕЛЬНОМУ КАНАЛУ С ИСПОЛЬЗОВАНИЕМ ФУНКЦИИ InterruptAttach().

/* Программа **3-ser_rd-int1-lab.c**. Принимает байт по последовательному каналу /dev/ser2 при возникновении прерывания, вызванного заполнением буфера приёма. Использует коммутацию нуль-модемным кабелем. Работает в паре с передающей программой 3-ser_lab.c. Иллюстрирует применение ф-ции InterruptAttach() для подключения обработчика прерываний. Использует отдельную высокоприоритетную нить для выполнения действий, связанных с прерыванием. */

```
#include <stdio.h>
#include <sys/neutrino.h>
#include <stdlib.h>
#include <hw/inout.h>
#include <pthread.h>

#define REG_RX   0      /* Адрес регистра-буфера приёма/передачи
                           относительно базового адреса AA  /dev/ser2 */
#define IER    1        //  Относительный адрес регистра IER
#define IIR    2        //  Относительный адрес регистра IIR
#define LCR    3        //  Относительный адрес регистра LCR
#define IIR_MASK 0x07  // Маска для выделения битов 0,1,2 регистра IIR
#define HW_SERIAL_IRQ 3     // IRQ  асинхронного адаптера /dev/ser2
```

```
#define N       26              // Получаем 25 символов + “\n”

volatile int serial_rx;     // Сохранённое значение буфера приёма

static int base_reg = 0x2f8;        // Базовый адрес  /dev/ser2

int interruptID;            // Идентификатор прерывания
struct sigevent event;      /* Структура, описывающая генерируемое  при возникновении
                               прерывания событие */
struct sched_param param ;/* Структура, характеризующая параметры  потока, запускаемого
                               после возникновения прерывания */
int S[N];                   // Массив для хранения принимаемых символов
//-------------------------------------------------------------------------------------------------------------
// Обработчик прерываний
const struct sigevent *handler( void *arg, int id ) {
    int  iir;           // Значение регистра идентификации прерываний
    struct sigevent *event = (struct sigevent *)arg;
    iir = in8 (base_reg + IIR) & IIR_MASK;    /* Считать биты, характеризующие
                                                 источник  прерываний и очистить регистр IIR */
    if (iir & 1) {
            return (NULL);/* Не было прерывания от устройства /dev/ser2 -  не возвращать
                             событие.  Прерывание в общем случае могло быть вызвано
                             другим устройством,  разделяющим тот же IRQ */
    }
```

```
    switch (iir) {                      // Определить причину прерывания
    case 0x04:                          // 100 –  прерывание вызвано получением байта
        serial_rx = in8 (base_reg + REG_RX);   // Прочесть  полученный байт
        return (event);                 // Разбудить поток-обработчик, возвратив событие
        break;
    case 0x02: // 010 – прерывание вызвано отправкой байта, в  рамках  лаб. работы обрабатывать
               // его не нужно
        break;
    default:
        break;
    }
}
//-------------------------------------------------------------------------------------------
```

// Функция потока, реагирующего на прерывания

```
void *int_thread (void *arg){
    int i;
    ThreadCtl( _NTO_TCTL_IO, 0 ); /* Запрос привилегии потоку на чтение/запись из/в
                                     регистры устройств и на присоединение обработчика прерываний */
    mmap_device_io( 8, base_reg );      /* Получение доступа к восьми регистрам
                                           последовательного порта  для чтения и записи */
```

/* ЗАДАНИЕ СКОРОСТИ И ПАРАМЕТРОВ КОММУНИКАЦИИ (альтернативный вариант с использованием драйвера ПАА devc-ser8250 и утилиты stty или структуры termios и функций группы управления терминалом смотри в справочной системе QNX/Neutrino по входам devc-ser8250, stty, termios, cfsetospeed(), sfsetispeed()): */

```
    out8(base_reg + LCR, 0x80);   // Установить в “1” бит DLAB
```

```
    out8(base_reg+REG_RX, 0x02);  /* Установить младший байт  делителя частоты для
                                     скорости передачи  57600 бит/с */
    out8(base_reg+IER, 0x00);     /* Установить старший байт делителя частоты для
                                     скорости передачи  57600 бит/с */
    out8 (base_reg + LCR, 0x03);  /* Обнулить DLAB и установить параметры
                коммуникации: нет проверки  на чётность, 1 стоповый бит, длина слова 8 бит */
    out8(base_reg+IER, 0x01);                  // Разрешить прерывания по получению байта
    event.sigev_notify = SIGEV_INTR;           // Инициализировать структуру события
    interruptID = InterruptAttach(HW_SERIAL_IRQ,&handler, &event,
                                  sizeof(event), 0);
    // Подключиться к источнику прерываний
    if (interruptID == -1)  {
       fprintf (stderr, "Attach error to IRQ %d\n", HW_SERIAL_IRQ);
       perror (NULL);
       exit (EXIT_FAILURE);
    }
    while (1) {
        InterruptWait( 0, NULL );/* Ждать события, которое уведомит нить о наступлении
                                    прерывания
        S[i++] = serial_rx; }      // Сохранить полученный байт в массиве
}
int main() {
   pthread_t  threadID;
   pthread_attr_t attrib;
   int *p;
```

```
    long long int i, Big = -10000000000;
```
/* Задать параметры – политику диспетчеризации и приоритет нити, запускающейся после возникновения прерывания : */
```
    pthread_attr_init (&attrib);
    pthread_attr_setinheritsched (&attrib, PTHREAD_EXPLICIT_SCHED);
    pthread_attr_setschedpolicy (&attrib, SCHED_RR);
    param.sched_priority = 11;// Проверьте, как влияет приоритет на получаемые значения
    pthread_attr_setschedparam (&attrib, &param);
    pthread_create (&threadID, &attrib, int_thread,NULL);
```
/* С этой точки программы потоки main() и дочерний выполняются параллельно*/

```
   for (i=0; i<100000000; i++) Big++;  /* Операции, выполняемые в main() параллельно с
                                          приёмом байтов по последовательному каналу */
   printf("Received symbols are : ");
   for(i=0; i < N; i++) printf("%c ",S[i]);  // Вывод полученных символов на экран

   InterruptDetach(interruptID);              // Отсоединение обработчик прерываний
   printf("Big = %lld\n", Big);        /* Что параллельно с приёмом
                                          символов насчитала основная программа*/
}
```
/* Программа **3-ser-lab.c**. Посылает 25 случайно выбранных символов по последовательному каналу с интервалом времени ≈ 20 мкс при на скорости передачи ПАА 57600 бит/с. Использует нуль-модемный кабель. Работает в паре с принимающей программой 3-ser_rd-int1-lab.c. */
```
#include <stdlib.h>
#include <hw/inout.h>
#include <sys/neutrino.h>
```

```
#define N  25          //  Посылаем  N=25 символов

int main() {
  int i, code, *p;
  int S[N+1];                          // В этом массиве храним отправленные символы
  int NL = 10;                         // “\n”
  static int base_reg = 0x2f8;         // Базовый адрес  /dev/ser2
  #define REG_RX       0   /* Адрес регистра-буфера приёма/передачи относительно базового
                              адреса AA  /dev/ser2  */
  #define IER      1            //  Относительный адрес регистра IER
  #define LCR      3            //  Относительный адрес регистра LCR

  ThreadCtl( _NTO_TCTL_IO, 0 );    /* Запрос привилегии потоку на чтение/запись из/в
                                      регистры устройств
  mmap_device_io(8, base_reg);     /* Получение доступа к восьми регистрам
                                      последовательного порта для чтения и записи */
    // ЗАДАНИЕ СКОРОСТИ И ПАРАМЕТРОВ КОММУНИКАЦИИ :
  out8 (base_reg + LCR, 0x80);     // Установить в “1” бит DLAB
  out8(base_reg+REG_RX, 0x02);     /* Установить младший байт  делителя частоты для
                                      скорости передачи  57600 бит/с */
  out8(base_reg+IER, 0x00);        /* Установить старший байт делителя частоты для
                                      скорости передачи   57600 бит/с */
  out8 (base_reg + LCR, 0x03);     /* Обнулить DLAB и установить параметры
                коммуникации: нет проверки  на чётность, 1 стоповый бит, длина слова 8 бит */
```

```
  for(i=0; i < N; i++) {
    code = (int) rand()/256;
    while ((code > 255) || (code < 32)) code = (int) rand()/256;
    S[i] = code;
    out8 (base_reg + REG_RX, code);   // Сгенерировать и послать случайный символ
    usleep(20);                       // Сделать паузу ≈ 20 мкс
    }
    out8(base_reg+REG_RX, NL);        // Послать символ перехода на новую строку

    p=S;
    printf ("Was sent symbols: ");
    for(i=0; i<N; i++) printf("%c ", *p++);/* Вывести на экран N посланных символов
                                              из массива S */
    printf("\n");
    exit( EXIT_SUCCESS );
}
```

2.2. ПРОГРАММА ПЕРЕДАЧИ ДАННЫХ МЕЖДУ ДВУМЯ КОМПЬЮТЕРАМИ ПО ПАРАЛЛЕЛЬНОМУ КАНАЛУ С ИСПОЛЬЗОВАНИЕМ ФУНКЦИИ InterruptAttachEvent().

/* Программа **3-par-lab_2.c**. Посылает младший полубайт случайно выбранного символа вместе с сигналом прерывания по параллельному каналу в полубайтном режиме, после чего делает паузу на случайное число секунд. Использует нуль-принтерный кабель. Работает в паре с принимающей программой 3-par_rd-int2-lab_4.c. Иллюстрирует доступ к портам ввода/вывода устройств ПК. */

```
#include <sys/neutrino.h>
#include <hw/inout.h>

static int data_port = 0x378; // Адрес регистра данных 1-го параллельного порта /dev/par1

int par_send(unsigned char x);
int main() {
    unsigned char symbol, paus;
    static unsigned int sd = 1329;  // Исходное значение для генератора случайных чисел
    int  randnumb;
    ThreadCtl( _NTO_TCTL_IO, 0 ); /* Запрос привилегии потоку на чтение/запись из/в
                                регистры   устройств и на присоединение обработчика прерываний */
    mmap_device_io(3, data_port); /* Получение доступа к трём регистрам параллельного
                                          порта для чтения и записи */
     while (1)                                  // Бесконечно отправлять символы
     {
        randnumb = rand_r(&sd);
```

```
        symbol = randnumb/147+31;       // Сгенерировать случайный символ
        par_send(symbol);               // Послать символа по параллельному каналу
        paus = randnumb/8000+1;         // Случайная пауза
        printf("suspended for %u sec.  ", paus);
        sleep(paus);                    // Заблокироваться на время не менее paus секунд
      }
}
//-------------------------------------------------------------------------------------------
int par_send(unsigned char symbol)
```
/* Функция формирует полубайт и отправляет его на принимающий компьютер вместе с сигналом прерывания */
```
{
    const unsigned char mask = 0x17;        // bin 00010111
    const unsigned char mask02 = 0x07;          /* 00000111 маска для выделения битов
                                                   0..2 полубайта */
    const unsigned char mask3 = 0x08;           // 00001000 маска для выделения битов 3 полубайта
    const unsigned char mask_invert = 0x10;   /* 00010000: маска для инвертирования
                                                    бита 7 регистра   состояния */
    unsigned char aux, aux2, nibble ;
    out8(data_port, 0x00);      /* Сбросить  биты 3..7 регистра состояния на принимающей
                                   машине, подготавливая приём следующего символа */
        // Сформировать отправляемый полубайт:
    nibble = symbol & 0x0F;         // Для символа ']'  nibble = 'D'=1101
     aux = nibble & mask02;         // 00000101
```

```
    aux2 = nibble & mask3;          // 00001000
    aux2 = aux2 << 1;               // 00010000
    aux = aux | aux2;               // 00010101
    aux2 = aux & mask;              // 00010101
    aux = aux2 ^ mask_invert;       // 00000101
    out8(data_port, aux | 0x08);  /* 00001101 Отправить полубайт вместе с сигналом
                                    прерывания, записав в регистр данных
    printf("low nibble %X of symbol %c was sent\n", nibble, symbol);
}
```

/* Программа **3-par_rd-int2-lab_4.c**. Принимает полубайт по параллельному каналу при возникновения
 * прерывания. Использует нуль-принтерный кабель. Работает в паре с передающей программой
 * 3-par_lab_2.c. Иллюстрирует применение ф-ции InterruptAttachEvent() для подключения обработчика
 * прерываний */


```
#include <stdio.h>
#include <sys/neutrino.h>
#include <stdlib.h>
#include <hw/inout.h>
#define HW_PARALLEL_IRQ 7                   // IRQ  для параллельного порта   /dev/par1
    const int data_port = 0x378;  // Адрес регистра данных параллельного порта /dev/par1
    const int status_port = 0x379;     //  Адрес регистра состояния 1-го параллельного
                                          порта
    const int control_port = 0x37A;//  Адрес регистра управления 1-го паралл.  порта
//-------------------------------------------------------------------------------------------------
int main()
```

```
{
    unsigned char mask = 0xB8;          /* 10111000 Маска для выделения битов 3..5,7
                                           регистра состояния */
    unsigned char mask345 = 0x38;       /*  00111000  Маска для выделения битов 0..2
                                           полученного полубайта */
    unsigned char mask7 = 0x80;         /*  10000000      Маска для выделения  бита  3
                                           полученного полубайта */

    unsigned char  aux, aux2;
    volatile unsigned char Bits;
    int interruptID;
     struct sigevent event;     /* event - указатель на структуру   struct sigevent, описываю-
                                   щую событие-прерывание */
     system("slay devc-par");  /* Завершить  два системных процесса - возможных
                                   конкурентов по доступу  к  параллельному  порту */
    system("slay spooler");
    ThreadCtl( _NTO_TCTL_IO, 0 );       /* Запрос привилегии потоку на чтение/запись из/в
                          регистры  устройств и на присоединение обработчика прерываний */
    mmap_device_io(3, data_port);       /* Получить доступ к трём регистрам параллельного
                          порта для  чтения и записи  */

     out8(control_port, in8(control_port) | 0x10); /*  Разрешить контроллеру
                          параллельного порта  генерировать прерывания */
     event.sigev_notify = SIGEV_INTR;/* Инициализировать структуру event для
                     уведомления  о наступлении генерируемого события - аппаратного прерывания*/
    interruptID = InterruptAttachEvent(HW_PARALLEL_IRQ, &event, 0);
```

```
                                                    /* Подключиться к   источнику прерываний */
   if (interruptID == -1)  {
      fprintf (stderr, "Attach error to IRQ %d\n",
               HW_PARALLEL_IRQ);
      perror (NULL);
      exit (EXIT_FAILURE);
   }

    while (1) // Бесконечно читать приходящие символы
   {
        InterruptWait( 0, NULL); // Ждать наступления прерывания
       Bits = in8(status_port); /* Считать 8 бит регистра состояния. Для символа
']' = 00001101  младший полубайт = 'D'=1101, в  статусном регистре должны быть  биты 1x101xxx */
       // Выделить полубайт :
       Bits = Bits & mask;          // 1x101000
       aux = Bits & mask345;        // 00101000
       aux = aux >> 3;              // 00000101
       aux2 = Bits & mask7;         // 10000000
       aux2 = aux2 >> 4;            //  00001000
       Bits = aux | aux2;           //  00001101 = 'D' для символа ']'

       printf("low = %X\n  ", Bits);   // receive low nibble
       InterruptUnmask(HW_PARALLEL_IRQ, interruptID); /* Демаскировать
            замаскированное ядром прерывание, чтобы оно могло сработать вновь  */
   }
```

```
}
```

2.3. ПРОСТЕЙШИЕ ПРОГРАММЫ ДЛЯ ПРОВЕРКИ КОММУТАЦИИ ПАРАЛЛЕЛЬНЫХ ПОРТОВ

/* Программа **par-write-test-2**.c. Посылает 5 бит по нуль-принтерному кабелю на параллельный порт компьютера-получателя. Работает в паре с программой par-read-test.c. */

```
#include <sys/neutrino.h>
#include <hw/inout.h>

main() {
    unsigned char symbol; /* Для символа 'w' с кодом 01110111 по кабелю передаются биты
```


0..4, т.е. 10111. На принимающей машине они попадают в разряды 3..7 статусного регистра. Бит 7 инвертируется контроллером параллельного порта. Считывая принятые биты из статусного регистра имеем 00111000 = 0x38 */

```
    int data_port = 0x378; //Адрес регистра данных dev/par1
    ThreadCtl( _NTO_TCTL_IO, 0 ); /* Получить
                возможность доступа к портам ввода-вывода */
    mmap_device_io(3, data_port); /* Получить доступ к
                регистрам параллельного порта для чтения и записи */
    symbol = getchar(); /*Ввести посылаемый символ с
                                клавиатуры */
    out8(data_port, symbol); // Послать биты
    printf("Symbol %c with code %X was sent\n",
           symbol, symbol);
```

```
}
```

/* Программа **par-read-test.c**. Читает и выводит на экран содержимое статусного регистра параллельного порта. Работает в паре с программой par-write-test-2.c. */

```
#include <sys/neutrino.h>
#include <hw/inout.h>

main() {
    const int data_port = 0x378;
    const int status_port = 0x379;
    unsigned char Bits;

    ThreadCtl( _NTO_TCTL_IO, 0 );
    mmap_device_io(3, data_port);
    Bits = in8(status_port);
    printf("Received = %X\n  ", Bits); /* Для
        отправленных пяти битов символа ‘w’ получаем 0x38 */
}
```


2.4. ПРОГРАММА ПОЛУЧЕНИЯ ИНФОРМАЦИИ ОБ ОБРАБОТЧИКАХ АППАРАТНЫХ ПРЕРЫВАНИЙ ПРОЦЕССОВ

/* На основе [25]. Программа работает в версиях QNX/Neutrino 6.2.1 и более новых.

В старых версиях отсутствует команда DCMD_PROC_IRQS (см.<sys/procfs.h>)*/
// # qcc -o procfs_my4.out procfs_my4.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
#include <ctype.h>
#include <dirent.h>
#include <fcntl.h>
#include <sys/procfs.h>

static  void    dump_procfs_irq (int fd, int pid, char *name);
static  void    dump_sigevent (struct sigevent *s);
static  void    iterate_process (int pid);
static  void    iterate_processes (void);
static  void    do_process (int pid, int fd, char *name);
//-------------------------------------------------------------------------------------------------
int main (int argc, char **argv)
{
    iterate_processes ();
    return (EXIT_SUCCESS);
}
//-------------------------------------------------------------------------------------------------
```

```
static void iterate_processes (void)
{// Находит все процессы с цифровыми идентификаторами как файлы в файловой системе /proc
 // Кроме файлов с цифровыми именами, /proc содержит также объекты-каталоги boot/, dumper#, self/
    struct dirent  *dirent;
    DIR            *dir;
    int            pid;

    dir = opendir ("/proc");                    // Открыть каталог /proc
    while (dirent = readdir (dir)) {
        if (isdigit (*dirent -> d_name)) {      // Если имя файла – цифровое, извлечь
            pid = atoi (dirent -> d_name);      // pid соответствующего процесса
            iterate_process (pid);
        }
    }
    closedir (dir);
}
//-----------------------------------------------------------------------------------------------------
static void iterate_process (int pid)
{// Находит информацию о процессе с заданным pid
    char   paths [PATH_MAX];
    int    fd;

    static struct {
        procfs_debuginfo  info;
        char              buff [PATH_MAX];
    } name;
```

```
    sprintf (paths, "/proc/%d/as", pid);// Путь к адресному пространству as процесса с
                                                    // заданным pid

    fd = open (paths, O_RDONLY);
    devctl (fd, DCMD_PROC_MAPDEBUG_BASE, &name, sizeof (name), 0);
                                                    // Получает имя процесса
    do_process (pid, fd, name.info.path);
    close (fd);
}
//-------------------------------------------------------------------------------------------------
static void do_process (int pid, int fd, char *name)
{/* "Обрабатывает" процесс - получает нужную информацию о нём. Здесь интересуют только
    прерывания, связанные с заданным процессом */
    //procfs_status   status;
    dump_procfs_irq (fd, pid, name);
}
//-------------------------------------------------------------------------------------------------
#define MAX_IRQS    512  // Максимальное IRQ с запасом
static void dump_procfs_irq (int fd, int pid, char *name)
{// Получает информацию об обработчиках прерываний заданного процесса и выводит её
    procfs_irq     irqs [MAX_IRQS];
    int            nirqs;
    int            i;

    devctl (fd, DCMD_PROC_IRQS, irqs, sizeof (irqs), &nirqs);
```

/* Команда DCMD_PROC_IRQS принуждает функцию devctl() заполнить массив структур irqs данными об обработчиках прерываний для заданного процесса*/

```
    if (nirqs > MAX_IRQS) {
        fprintf (stderr, "  process %d has more than %d IRQs (%d in fact)
!!! ***\n",  pid, MAX_IRQS, nirqs);
        exit (EXIT_FAILURE);
    }

    for (i = 0; i < nirqs; i++) {//вывод членов стуктуры irqs
        printf ("\nPROCESS ID %d  NAME %s\n", pid, name);
        printf ("Info from DCMD_PROC_IRQS\n");
        printf ("\tBuffer %3d\n", i);
        printf ("\t\tpid        %d\n", irqs [i].pid);
        printf ("\t\ttid        %d\n", irqs [i].tid);
        printf ("\t\thandler    0x%08X\n",
                                    (unsigned int) irqs [i].handler);
        printf ("\t\tarea       0x%08X\n", (unsigned int) irqs [i].area);
        printf ("\t\tflags      0x%08X\n", irqs [i].flags);
        printf ("\t\tlevel      %d\n", irqs [i].level);
        printf ("\t\tmask_count %d\n", irqs [i].mask_count);
        printf ("\t\tid         %d\n", irqs [i].id);
        printf ("\t\tvector     %d\n", irqs [i].vector);
        dump_sigevent (&irqs [i].event);
    }
}
```

```
//-------------------------------------------------------------------------------------------------
static void dump_sigevent (struct sigevent *s)
{// Анализирует и выводит вид события, уведомляющего о прерывании
    printf ("\t\tevent (sigev_notify type %d)\n", s -> sigev_notify);
    switch (s -> sigev_notify) {
    case   SIGEV_NONE:
        printf ("\t\t\tSIGEV_NONE\n");
        break;
    case   SIGEV_SIGNAL:
        printf ("\t\t\tSIGEV_SIGNAL (sigev_signo %d,
                sigev_value.sival_int %d)\n",
                s -> sigev_signo, s -> sigev_value.sival_int);
        break;
    case   SIGEV_SIGNAL_CODE:
        printf ("\t\t\tSIGEV_SIGNAL_CODE (sigev_signo %d,
                sigev_value.sival_int %d, sigev_code %d)\n",
                s -> sigev_signo, s -> sigev_value.sival_int,
                s -> sigev_code);
        break;
    case   SIGEV_SIGNAL_THREAD:
        printf ("\t\t\tSIGEV_SIGNAL_THREAD (sigev_signo %d,
                sigev_value.sival_int %d, sigev_code %d)\n",
                s -> sigev_signo, s -> sigev_value.sival_int,
                s -> sigev_code);
        break;
    case   SIGEV_PULSE:
```

```
        printf ("\t\t\tSIGEV_PULSE (sigev_coid 0x%08X,
            sigev_value.sival_int %d, sigev_priority %d, sigev_code %d)\n",
                 s -> sigev_coid, s -> sigev_value.sival_int,
                 s -> sigev_priority, s -> sigev_code);
        break;
    case   SIGEV_INTR:
        printf ("\t\t\tSIGEV_INTR\n");
        break;
    case   SIGEV_UNBLOCK:
        printf ("\t\t\tSIGEV_UNBLOCK\n");
        break;
    default:
        printf ("\t\t\t*** unknown sigev_notify type\n");
        break;
    }
}
```


Вывод программы (кроме IRQ важной информацией является вид события и его приоритет (в случае импульса)):

```
PROCESS ID 1 NAME /home/builder/daily/x86/boot/sys/procnto (менеджер процесов)
```

```
Info from DCMD_PROC_IRQS
    Buffer   0
        pid         1
        tid         1
        handler     0xF0027BA1
        area        0x00000000
        flags       0x00000002
        level       16
        mask_count  0
        id          0
        vector      -2147483648
        event (sigev_notify type 0)
            SIGEV_NONE
```



```
PROCESS ID 1  NAME /home/builder/daily/x86/boot/sys/procnto (менеджер процессов)
Info from DCMD_PROC_IRQS
    Buffer   1
        pid         1
        tid         1
        handler     0xF002764A
        area        0x00000000
        flags       0x00000002
        level       0
        mask_count  0
        id          1
        vector      0                       (IRQ 0)
```

```
        event (sigev_notify type 0)
            SIGEV_NONE

PROCESS ID 6  NAME proc/boot/devb-eide   (драйвер жёстких дисков)
Info from DCMD_PROC_IRQS
    Buffer   0
        pid         6
        tid         2
        handler     0x00000000
        area        0xEFFE6160
        flags       0x0000000E
        level       14
        mask_count  0
        id          2
        vector      14                      (IRQ 14)
        event (sigev_notify type 4)
            SIGEV_PULSE (sigev_coid 0x40000002, sigev_value.sival_int 0,
                         sigev_priority 21, sigev_code 2)

PROCESS ID 6  NAME proc/boot/devb-eide   (драйвер жёстких дисков)
Info from DCMD_PROC_IRQS
    Buffer   1
        pid         6
        tid         3
        handler     0x00000000
        area        0xEFFE61D8
```

```
        flags      0x0000000E
        level      15
        mask_count 0
        id         3
        vector     15                        (IRQ 15)
        event (sigev_notify type 4)
            SIGEV_PULSE (sigev_coid 0x40000005, sigev_value.sival_int 0,
                         sigev_priority 21, sigev_code 2)

PROCESS ID 7  NAME pkgs/repository/qnx/os/drivers-2.1.4/x86/sbin/devc-con
                                                  (драйвер клавиатуры)
Info from DCMD_PROC_IRQS
    Buffer   0
        pid        7
        tid        1
        handler    0x00000000
        area       0xEFFE6218
        flags      0x00000006
        level      1
        mask_count 6457
        id         4
        vector     1                         (IRQ 1)
        event (sigev_notify type 4)
            SIGEV_PULSE (sigev_coid 0x40000002, sigev_value.sival_int 0,
                         sigev_priority 15, sigev_code 2)
```

```
PROCESS ID 77839  NAME sbin/devc-ser8250 (драйвер последовательного порта)
Info from DCMD_PROC_IRQS
    Buffer   0
        pid         77839
        tid         1
        handler     0x080499C4
        area        0x080562E8
        flags       0x00000000
        level       4
        mask_count  0
        id          5
        vector      4                          (IRQ 4)
        event (sigev_notify type 4)
            SIGEV_PULSE (sigev_coid 0x40000002, sigev_value.sival_int 0,
                         sigev_priority 15, sigev_code 2)

PROCESS ID 77839  NAME sbin/devc-ser8250 (драйвер последовательного порта)
Info from DCMD_PROC_IRQS
    Buffer   1
        pid         77839
        tid         1
        handler     0x080499C4
        area        0x080575A0
        flags       0x00000000
        level       3                          (IRQ 3)
        mask_count  0
```

```
        id          6
        vector      3
        event (sigev_notify type 4)
            SIGEV_PULSE (sigev_coid 0x40000002, sigev_value.sival_int 0,
                         sigev_priority 15, sigev_code 2)

PROCESS ID 106513  NAME sbin/devb-fdc     (Драйвер контроллера флоппи-диска)
Info from DCMD_PROC_IRQS
    Buffer   0
        pid         106513
        tid         2
        handler     0x00000000
        area        0xEF156138
        flags       0x0000000A
        level       6
        mask_count 0
        id          8
        vector      6                     (IRQ 6)
        event (sigev_notify type 6)
            SIGEV_INTR

PROCESS ID 77842  NAME sbin/io-net        (Драйвер сетевой карты)
Info from DCMD_PROC_IRQS
    Buffer   0
        pid         77842
        tid         3
```

```
handler    0x00000000
area       0xEF2705C8
flags      0x0000000E
level      11
mask_count 0
id         7
vector     11                  (IRQ 11)
event (sigev_notify type 4)
    SIGEV_PULSE (sigev_coid 0x40000014, sigev_value.sival_int 0,
                 sigev_priority 21, sigev_code 0)
```

3. ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНЫХ РАБОТ И ПОРЯДОК ИХ ВЫПОЛНЕНИЯ

Подготовьте лабораторные компьютеры:

1. Соедините пару лабораторных компьютеров нуль - модемным и нуль - принтерным кабелями (делать только при отключённом питании обоих компьютеров!).
2. Проверьте работоспособность коммутации. Для последовательного канала используйте утилиту **cat**. На машине – отправителе набрать команду

   #cat - > /dev/ser*

   и вводить с клавиатуры отправляемые строки. Читать их на машине - получателе, используя команду

   #cat /dev/ser*. (* означает 1 или 2)

   Для параллельного канала запустите простейшие программы отправки полубайта путём записи его в регистр данных и чтение полубайта из регистра без использования прерываний (см. приложение 5).
3. Запустите, убедитесь в работоспособности и проанализируйте работу программ-примеров из раздела 2.

Выполните индивидуальные задания:

1. Циклическая передача 8-битных случайно выбранных символов по параллельному каналу. Использовать функцию InterruptAttach().
2. Циклическая передача 8-битных случайно выбранных символов по параллельному каналу. Использовать функцию InterruptAttachEvent().
3. Циклическая передача случайно выбранных символов по последовательному каналу. Использовать функцию InterruptAttachEvent().
4. Циклическая посылка 8-битных случайно выбранных символов по X каналу и немедленный возврат по Y каналу. Канал X организовать с использованием функции F1, канал Y – с использованием функции F2. Варианты:

№ варианта	Канал X	Канал Y	Функция F1	Функция F2
4.1	Пос.	Пос.	InterruptAttach	InterruptAttach
4.2	Пос.	Пос.	InterruptAttach	InterruptAttachEvent
4.3	Пос.	Пос.	InterruptAttachEvent	InterruptAttach
4.4	Пос.	Пос.	InterruptAttachEvent	InterruptAttachEvent
4.5	Пос.	Пар.	InterruptAttachEvent	InterruptAttach
4.6	Пар.	Пос.	InterruptAttachEvent	InterruptAttachEvent
4.7	Пар.	Пос.	InterruptAttach	InterruptAttach
4.8	Пос.	Пар.	InterruptAttach	InterruptAttachEvent

5. Задания 1-4 выполнить, используя в качестве уведомления импульс.
6. Задания 1-4 выполнить, используя в качестве уведомления сигнал реального времени.
7. Циклическая одновременная посылка 8-битного символа по последовательному и параллельному каналам с одного компьютера на другой. Замерить интервалы между отсылкой соседних символа на 1ой машине и интервалы между доставкой соседних символов отдельно по обоим каналам на 2ой машине. Накопить статистику временнЫх интервалов и сравнить, используя статистические критерии (см. часть 1 настоящего пособия), эмпирические функции распределения интервалов отправки и приёма, а также интервалы приёма по обоим каналам. Какой из каналов передачи меньше искажает статистические характеристики временных интервалов?
8. При обсуждении проблем драйверов сетевых карт среди прочих высказывалось мнение, что маскирование прерывания с приоритетом *P* с помощью `InterruptMask()` вызывает также маскирование всех прерываний более низкого приоритета. Проверьте, так ли это, замаскировав прерывание последовательного порта и определив влияние этой маски на прерывания параллельного порта.

Для устойчивой работы программ передачи данных может понадобиться временный запрет или маскирование прерываний при обработке очередной порции данных (`InterruptLock()`–`InterruptUnlock()`,`InterruptMask()`-`InterruptUnmask()`).

ПРИЛОЖЕНИЯ

ПРИЛОЖЕНИЕ 1. Настройка и управление работой последовательного порта при передаче данных между компьютерами по нуль-модемному кабелю

Порт последовательной передачи данных, или последовательный асинхронный адаптер (ПАА), в персональных компьютерах реализован на чипе универсального асинхронного приёмопередатчика (UART) семейства 8250. ПАА имеет ряд регистров данных, управления и состояния, имеет возможность генерировать прерывания по отправке и/или получению порции данных, а также по изменению линии состояния приемника и состояния входных линий от модема. ПАА позволяет организовать обмен данными с внешними устройствами на основе интерфейса RS-232C, RS-485 и др. с использованием различных видов кабелей в зависимости от назначения устройства, в частности даёт возможность коммутации двух компьютеров между собой [34,35].

Менеджер `devc-ser8250` последовательного порта ОСРВ QNX, утилиты семейства стандартов POSIX (например, `stty`), а также POSIX 1003.1 структура `termios` и многочисленные библиотечные функции языка С – собственные QNX/Neutrino, POSIX - и UNIX-совместимые (`tcsetattr()`, `cfsetispeed()`, `devctl()`, `open()`, `write()`, `tcdrain()`, `read()`, `readcond()`, `select()`,…) позволяют эффективно сконфигурировать и использовать ПАА для передачи данных в различных режимах.

В соответствии с целями лабораторных работ по курсу “Системы реального времени” используется лишь малая часть возможностей ПАА – организуется передача байтов между двумя компьютерами по нуль-модемному кабелю с обработкой полученных данных по прерыванию. Настройка ПАА осуществляется на уровне регистров (все регистры - восьмиразрядные) без использования высокоуровневых утилит и функций, перечисленных выше. Необходимая для лабораторной работы информация об ПАА и их коммутации приведены ниже [36]. Для удобства чтения документации и программ на английском языке русскоязычные термины дополнены общепринятыми англоязычными сокращениями.

Таблица П.1.1

Общие сведения о портах последовательной передачи данных *

№ АА	Обозначение DOS/Windows	Обозначение QNX	Базовый адрес (base)	IRQ
1	COM1	/dev/ser1	0x3F8	4
2	COM2	/dev/ser2	0x2F8	3

*Примечание. Если на лабораторных компьютерах установлена мышь USB или PS/2, для целей коммуникации можно использовать оба порта.

Регистр приёма-передачи.

Адрес регистра Base + 0.

Функции регистра зависят от старшего бита (DLAB) регистра управления линией (LCR):

DLAB = 0. При посылке регистр содержит отправляемый байт, при приёме – получаемый байт данных (буферы передачи и приёма).

DLAB = 1. Регистр содержит младший байт делителя частоты (1.8432 МГц) тактового генератора микросхемы UART.

Регистр управления прерываниями (IER)

Адрес регистра Base+1.

Функции регистра зависят от старшего бита (DLAB) регистра управления линией (LCR):

DLAB = 0.

- Установка бита 0 в уровень “1” разрешает прерывания при получении байта данных.
- Установка бита 1 в уровень “1” разрешает прерывания после отправки байта данных.

DLAB = 1. Регистр содержит старший байт делителя частоты (1.8432 МГц) тактового генератора микросхемы UART.

Регистр идентификации прерываний (IIR)

Адрес регистра Base+2. Доступен только для чтения.

Таблица П.1.2

Назначение разрядов регистра идентификации прерываний

Биты			Назначение регистра
6 и 7	Бит 6	Бит 7	**Наличие буфера FIFO**
	0	0	UART не имеет буфера FIFO (16450 и более старые)
	0	1	Буфер имеется, но не используется из-за ошибки реализации (16550)
	1	1	UART имеет работоспособный буфер FIFO (16550a и новее)
1 и 2***	Бит 2	Бит 1	**Условие возникновения прерывания**
	0	1	Байт отправлен, буфер передатчика (регистр Base+0) пуст *
	1	0	Байт принят и доступен для чтения из буфера приёма (регистр Base+0)**
0	0		Есть прерывания, ожидающие обслуживания (pending interrupt)
	1		Нет прерываний, ожидающих обслуживания

Примечания. * - Бит сбрасывается после чтения байта из регистра приёма (Base+0).

** - Бит сбрасывается после записи нового байта в регистр передачи (Base+0)

*** - Причины прерываний расположены в порядке увеличения приоритета

Регистр управления FIFO - буферами ввода/вывода (FCR)

Адрес регистра Base+2. Доступен для записи.

Таблица П.1.3

Назначение разрядов регистра управления буферами FIFO

Биты			Назначение регистра
6 и 7	Бит 6	Бит 7	**Уровень генерации прерывания ***
	0	0	1 байт
	0	1	4 байта
	1	0	8 байтов

Окончание табл. П.1.3

	1	1	14 байтов
2	Очистить FIFO буфер передачи		
1	Очистить FIFO буфер приёма		
0	Сделать возможным использование буферов FIFO		

Примечание. * Контроллер ПАА генерирует прерывание также в случае приостановки потока данных независимо от выставленного значения уровня генерации прерывания (Interrupt Trigger Level).

Регистры **IIR** и **FCR** позволяют легко определить тип ПАА. Для этого бит 0 регистра **FCR** устанавливается в “1” и читаются биты 6 и 7 регистра **IIR**. Вывод делается на основании табл.П.1.2.

Регистр управления линией (LCR)

Адрес регистра Base+3. Регистр позволяет задать базовые параметры коммуникации. Для правильной передачи данных параметры коммуникации обоих связанных ПАА должны быть одинаковы.

Таблица П.1.4

Назначение битов регистра LCR

Бит	Назначение бита			
Бит 7	1	(DLAB—Divisor Latch Access Bit—Бит-защёлка делителя частоты тактового генератора). Регистры Base и Base+1 используются для загрузки делителя частоты (см. таблицу П.1.5)		
	0	Регистры используются по прямому назначению		
Биты 3..5	Бит 5	Бит 4	Бит 3	Выбор режима проверки принимаемых данных по чётности
	X	X	0	контроль на чётность не используется
	0	0	1	контроль на нечётность
	0	1	1	контроль на чётность
Бит 2	Количество стоповых бит			
	0	1 бит		
	1	2 бита		
Биты 0 и 1	Бит 1	Бит 0	Длина передаваемого слова	
	1	1	8 бит	

Регистр состояния линии (LSR)

Адрес регистра Base+5. Значение “1” бита 0 означает, что получен байт и буфер приёма готов для чтения. Это используется при организации обмена данными через ПАА в режиме опроса (polling). Режим

опроса во многих случаях менее эффективен, чем режим, основанный на прерываниях, поскольку загружает процессор непрерывным циклом проверки состояния бита 0.

Таблица П.1.5

Значение скорости передачи данных в зависимости от делителя частоты

Скорость Бит/с	Делитель	Старший байт делителя	Младший байт делителя
50	2304	0x09	0x00
300	384	0x01	0x80
600	192	0x00	0xC0
2400	48	0x00	0x30
4800	24	0x00	0x18
9600	12	0x00	0x0C
19200	6	0x00	0x06
38400	3	0x00	0x03
57600	2	0x00	0x02
115200	1	0x00	0x01

Таблица П.1.6

Схема разводки нуль-модемного кабеля

Номера контактов		Назначение контакта	Направление передачи	Назначение контакта	Номера контактов	
Разъём DB9	Разъём DB25				Разъём DB25	Разъём DB9
3	2	Передаваемые данные (TD)	→	Принимаемые данные (RD)	3	2
2	3	Принимаемые данные (RD)	←	Передаваемые данные (TD)	2	3
5	7	Сигнальное заземление (SG)	↔	Сигнальное заземление (SG)	7	5

Схема распайки заглушки для тестирования ПАА и использования его в качестве источника периодических прерываний приведена на рис. 4.

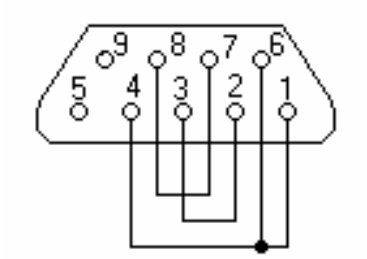

Рис. П.1.1. Схема распайки заглушки ПАА

Алгоритм передачи байта по последовательному каналу с использованием прерываний от заполнения буфера приёма ПАА иллюстрируют программы 3-ser-lab.c и 3-ser_rd-int1-lab.c (см. п. 2.1).

ПРИЛОЖЕНИЕ 2. Настройка и управление работой стандартного параллельного порта при передаче данных между компьютерами по нуль-принтерному кабелю

Стандартный параллельный порт (ПП) функционирует при установке в BIOS настроек SPP (Standard Parallel Port) или Normal. С соответствующими кабелями порт может обеспечивать однонаправленную передачу восьмибитных данных, двунаправленную - четырёхбитных и способен вырабатывать запрос на прерывание IRQ7 [34,35]. Менеджер параллельного порта devc-par ОСРВ QNX обеспечивает минимальные возможности управления работой ПП, недостаточные для использования его в целях коммуникации двух компьютеров (например, чтение из ПП эквивалентно чтению из /dev/null). Поэтому в рамках данной лабораторной работы devc-par не используется, а программируется непосредственно работа ПП на уровне регистров с обработкой прерываний.

Мы используем первый из нескольких (до 4х) поддерживаемых BIOS ПП с базовым адресом Base = 378h - принтерный порт LPT1, или устройство /dev/par1 в архитектуре ОС QNX. Передача данных осуществляется в полубайтном режиме с использованием нуль-принтерного кабеля “LapLink” (или “InterLink”). Порт имеет 3 восьмиразрядных регистра – регистр данных, регистр состояния и регистр управления. Необходимая для выполнения лабораторной работы информация о регистрах приведена в таблицах [37]:

Таблица П.2.1

Регистр данных

Адрес в адресном пространстве ввода/вывода	**Возможность чтения/записи из программы**	**Бит №**	**Назначение (№ контакта разъёма DB-25)**
Base + 0	Только запись	4	Бита данных 3 (Pin 6)
		3	Сигнал Ack для вызова прерывания на стороне приёма (Pin 5)
		2	Бит данных 2 (Pin 4)
		1	Бит данных 1 (Pin 3)
		0	Бит данных 0 (Pin 2)

Таблица П.2.2

Регистр состояния

Адрес в адресном пространстве ввода/вывода	**Возможность чтения/записи из программы**	**Бит №**	**Назначение (№ контакта разъёма DB-25)**
Base + 1	Только чтение	7	Бит данных 7 (Pin 11) *
		6	Сигнал Ack, вызывающий прерывание (Pin 10)
		5	Бит данных 5 (Pin 12)
		4	Бит данных 4 (Pin 13)
		3	Бит данных 3 (Pin 15)

Примечание. * Бит 7 аппаратно инвертируется контроллером – записанная по кабелю “1” при чтении в программе даёт “0”, и наоборот.

Таблица П.2.3

Регистр управления

Адрес в адресном пространстве ввода/вывода	**Возможность чтения/записи**	**Бит №**	**Назначение**
Base + 2	Чтение/Запись	4	Разрешить прерывания по сигналу линии Ack

Схема распайки нуль-принтерного кабеля и соответствующие биты регистров контроллера ПП приведены на рис.П.2.1.

Как видно из рис.П.2.1, кабель LapLink даёт возможность передавать одновременно пять бит, четыре из которых являются частью пересылаемой информации, а 5-ый используется для выработки прерывания, связанного с получением очередного полубайта.

Алгоритм передачи полубайта по параллельному каналу с использованием прерываний следующий:

1. Разрешаем прерывания на принимающем контроллере ПП, установив в “1” бит 4 регистра управления.
2. Подготавливаем принимающий контроллер ПП к восприятию сигнала прерывания, переслав по кабелю в бит “6” регистра состояния уровень “0”.

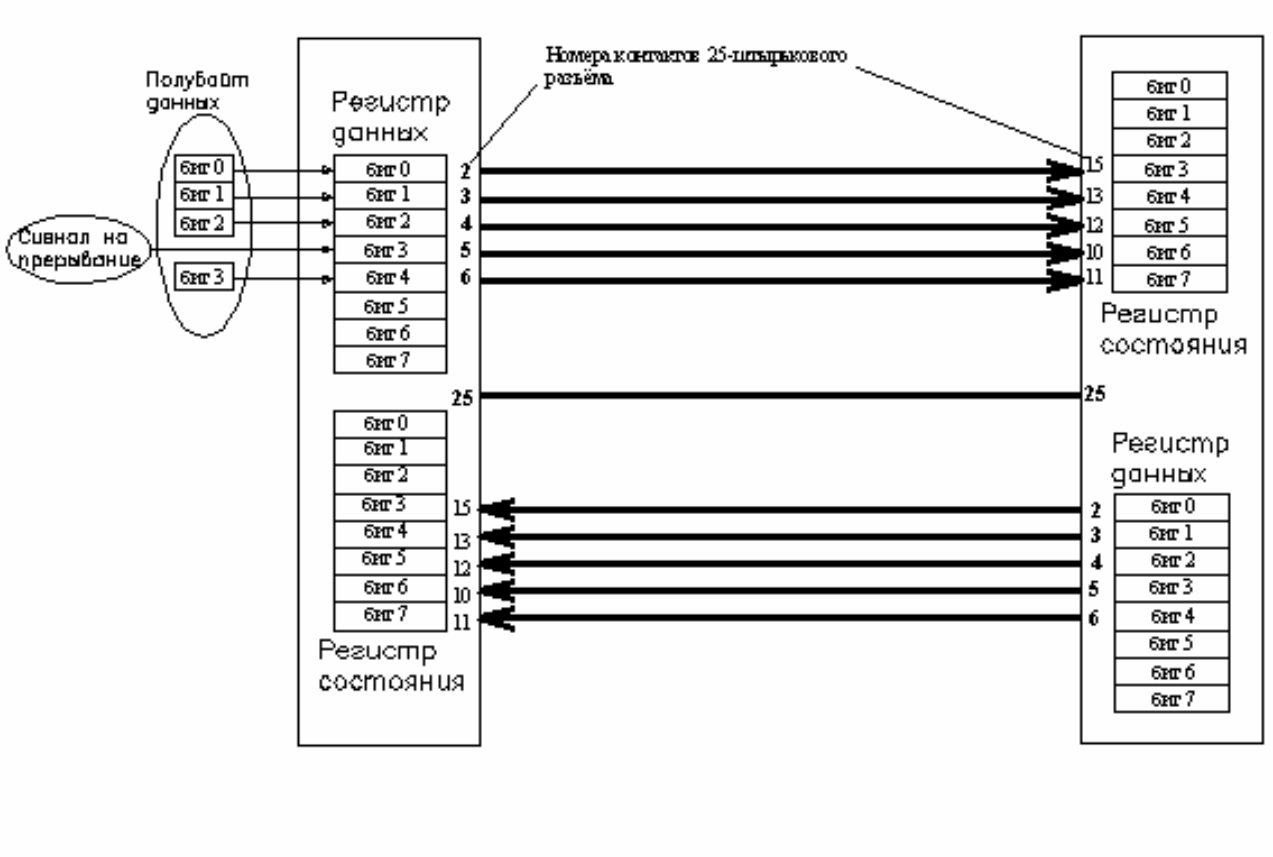

Рис.П.2.1. Схема соединения двух параллельных портов для полубайтной передачи данных

3. На машине-отправителе формируем 5 бит = 4 бита отправляемого полубайта + уровень “1” сигнала прерывания Ack (см. рис), записываем сформированные 5 бит в биты 0..4 регистра данных контроллера ПП.
4. Считываем биты 3..5, 7 регистра состояния ПП на принимающей машине, формируем полубайт данных и используем его по назначению.
5. Передача следующих полубайтов осуществляется повторением п.п. 2,3,4.

Указанный алгоритм реализован в программах: 3-par-lab_2.c для отправляющей машины и 3-par_rd-int2-lab_4.c для принимающей машины (см. пп. 2.2).

ЗАКЛЮЧЕНИЕ

Изложенные в настоящей части пособия материалы следует рассматривать как введение в достаточно специфичную область программного управления и взаимодействия с аппаратурой, входящую в программирование систем реального времени. Эта область требует внимательного изучения и проработки двух направлений – архитектуры и способов взаимодействия конкретных аппаратных устройств на уровне физических интерфейсов и способов программного управления этим взаимодействием.

Программное направление чётко реализуется стандартными средствами API, предоставляемыми ОС QNX, и при внимательном изучении документации может быть достаточно быстро и без ошибок реализовано программистом. Перед тем, как приступить к программированию, необходимо иметь чёткое представление об архитектуре и физической реализации системы.

Что касается особенностей аппаратного устройства конкретных компонентов систем реального времени, объединенных в систему с возможностью передавать/получать данные и/или управлять отдельными устройствами по прерываниям, то здесь существует большое количество различных решений и требуется специальный анализ для каждой конкретной системы. Способы программирования конкретной системы существенным образом зависят от аппаратной реализации, в частности, настройки чувствительности контроллера прерываний по уровню или фронту, типов шин, к которым подсоединены устройства, и наличия одного или более устройств на одной линии прерывания.

Другой группой факторов, определяющих архитектуру программ для управляемых прерываниями систем является совокупность временнЫх ограничений на функционирование. Это связано с тем, что функция-обработчик прерываний выполняется на сверхвысоком приоритете, вытесняя обычные потоки и обработчики прерываний более низкого приоритета. Как и во многих других случаях, может существовать несколько вариантов программы, которые могут отличаться по временнЫм характеристикам обработки прерываний и выполнения задач жёсткого реального времени. В конечном счёте должен быть найден такой вариант, который удовлетворяет требованиям реального времени в наихудших условиях работы системы.

СПИСОК ИСПОЛЬЗОВАННОЙ И РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

Рекомендуемые источники выделены полужирным шрифтом

1. **Кёртен, Р. Введение в QNX Neutrino 2: руководство по программированию приложений реального времени в QNX Realtime Platform/ Р. Кёртен – СПб.: Петрополис, 2001 - 480 с. - СПб.:БХВ-Петербург, 2005 доп. тираж -400 с.**
2. Юров В., Хорошенко С. Ассемблер: учебный курс – СПб: Издательство “Питер”, 2000.-672 с.
3. Intel 8259A Programmable Interrupt Controller Datasheet http://www.ee.bilkent.edu.tr/~ee212/8259a.pdf, http://www.ledman.ch/eti_hardware/18lstgdwnld.html
4. Intel MultiProcessor Specification Version 1.4. http://developer.intel.com/design/pentium/datashts/24201606.pdf
5. **Гук М.Ю.Шины PCI, USB и FireWire. Энциклопедия. / М.Ю.Гук - 2005.544 с.** www.wasm.ru/docs/10/Guk_PCI.pdf
6. Mike Rieker. Advanced Programmable Interrupt Controller http://osdev.berlios.de/pic.html
7. Key Benefits of the I/O APIC http://www.microsoft.com/whdc/system/sysperf/IO-APIC.mspx
8. APIC http://en.wikipedia.org/wiki/Intel_APIC_Architecture
9. Статья Д. Зиновьев. Страдания по IRQ. http://www.ferra.ru/online/system/25464/
10. Advanced Configuration and Power Interface Specification (Hewlett-Packard Corporation/Intel Corporation/Microsoft Corporation/ Phoenix Technologies Ltd./Toshiba Corporation) Revision 3.0a December 30, 2005 http://cdgenp01.csd.toshiba.com/content/pr/download/AdvancedConfiguration_PowerSpecificationv3.pdf
11. IRQ STEERING AND DEVICE ENUMERATION. http://members.bellatlantic.net/~mrscary/devenum.htm
12. Как указать вручную драйвер уровня аппаратных абстракций во время установки или обновления Microsoft Windows XP http://support.microsoft.com/?kbid=299340
13. Intel 82093AA I/O ADVANCED PROGRAMMABLE INTERRUPT CONTROLLER (IOAPIC) http://www.intel.com/design/chipsets/datashts/290566.htm
14. The Importance of Implementing APIC-Based Interrupt Subsystems

on Uniprocessor PCs
http://www.microsoft.com/whdc/system/sysperf/apic.mspx
15. Отчёт группы экспертов Dedicated Systems Experts “QNX NEUTRINO RTOS v6.2.0”. Русский перевод http://www.cniil.org/QNX_NEUTRINO_RTOS_V6_2_0.html
16. Spurious IRQ interrupt 8259… http://lists.debian.org/debian-user-french/2001/11/msg01042.html
17. Режимы реагирования на уровень и на фронт http://www.xserver.ru/computer/computer/control/1/7.shtml
18. Interfacing the Extended Capabilities Port Version 1.0. http://www.senet.com.au/~cpeacock
19. Отчёт группы экспертов Dedicated Systems Experts QNX® NEUTRINO® RTOS V6.3 Evaluation Report EVA-2.9-OS-QNX-01 January 19, 2005 http://www.dedicated-systems.com
20. Статья Виталий ЯКУСЕВИЧ. BIOS и его настройки. http://mycomp.kiev.ua/text/9654;jsessionid=667F9A14D9F40836F8551E17087A97B9
21. Igor Kovalenko "pci -v" used on QNX 6.2.1/QNX4 http://sysadminforum.com/showthread.php?s=19d2d73d61fdadc7b64d99bc4aff0931&p=259875#post259875
22. Страничка Ed1k на форуме qnx.org.ru. Утилита picinfo. http://ed1k.qnx.org.ru/picinfo.html
23. Страничка Ed1k на форуме qnx.org.ru. Утилита setirq. http://ed1k.qnx.org.ru/utils/setirq/qnx6/SETIRQ.TGZ
24. Статья Dave Donohoe. Talking to hardware under QNX Neutrino. http://www.qnx.com/developers/articles/article_304_2.html
25. Krten, R.. The QNX CookBook: Reciepes for Programmers. PARSE Software Devices, 2003, ISBN 0-9682501-2-2.
26. Учебный курс фирмы QNX Software Systems Ltd.“Realtime Programming for the QNX® Neutrino RTOS” 2004/04/28 R22
27. Сместный веб-семинар QNX Software Systems и Dedicated Systems Experts “Beyond Moore’s Law: Breaking the Performance Barrier Requires a Real-time Solution” 5 декабря 2002г. http://www.techonline.com/community/home/21507
28. Real-Time systems FAQ http://www.dedicated-systems.com/encyc/publications/faq/rtfaq.htm
29. Отчёт Dedicated Systems Experts DSE-RTOS-EVA-002 EVALUATION REPORT DEFINITION http://dedicated-systems.com/encyc/BuyersGuide/RTOS/Evaluations/

30. Отчёт Dedicated Systems Experts COMPARISON BETWEEN QNX RTOS V6.1, VXWORKS AE 1.1 AND WINDOWS CE .NET 31.05.2002 http://www.dedicated-systems.com
31. Отчёт Dedicated Systems Experts о тестировании QNX Neutrino 6.2, VxWorks AE 1.1, Windows CE .NET и ELDS 1.1 7.11.2002 http://www.dedicated-systems.com
32. General Programming-problem about share interrupt http://www.openqnx.com/PNphpBB2+viewtopic-t-1798.html
33. qnx.ddk - hardware interrupt IRQ conflict http://www.openqnx.com/index.php?name=PNphpBB2&file=printview&t=5978&start=0-qnx.ddk
34. А.В.Фролов. Аппаратное обеспечение IBM PC./ Фролов А.В., Фролов Г.В.- М.: "Диалог-МИФИ", 1997, (Библиотека системного программиста, Т.33).
35. **М.Гук. Аппаратные средства PC. Энциклопедия./ Гук М.- СПб.: Питер Ком, 2001.**
36. Craig Peacock. Interfacing the Serial / RS232 Port. http://www.beyondlogic.org/serial/serial.htm.
37. Craig Peacock. Interfacing the Standard Parallel Port. http://www.beyondlogic.org/spp/parallel.htm.