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 I

СЛУЖБА ВРЕМЕНИ ОПЕРАЦИОННОЙ СИСТЕМЫ РЕАЛЬНОГО ВРЕМЕНИ QNX / Neutrino. СИСТЕМНОЕ ВРЕМЯ, ЧАСЫ, ТАЙМЕРЫ, ТАЙМАУТЫ

(Версия 9-02-2007)

Брянск 2007

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

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

ПРЕДИСЛОВИЕ 6

ВВЕДЕНИЕ 8

1. СРЕДСТВА СЛУЖБЫ ВРЕМЕНИ QNX/Neutrino И ИХ ИСПОЛЬЗОВАНИЕ 10
1.1. НЕКОТОРЫЕ ПОНЯТИЯ И ТЕРМИНЫ 10
1.2. ИСТОЧНИКИ ПЕРИОДИЧЕСКИХ АППАРАТНЫХ ПРЕРЫВАНИЙ 13
1.3. УСТАНОВКА И КОРРЕКТИРОВКА СИСТЕМНОГО ВРЕМЕНИ 18
1.3.1. Средства графической среды Photon 19
1.3.2. Утилиты командной строки 21
1.3.3. Установка и корректировка даты/времени из программы 23
1.4. МОДЕЛЬ СИСТЕМНОГО ВРЕМЕНИ ОС РВ QNX/NEUTRINO 23
1.5. КЛАССИФИКАЦИЯ ТАЙМЕРОВ 28
1.6. КАК РАБОТАЮТ ТАЙМЕРЫ 30
1.6.1. Интервальные таймеры уведомления 35
1.6.2. Таймеры задержки 39
1.7. ИСПОЛЬЗОВАНИЕ ТАЙМЕРОВ В ПРОГРАММАХ 44
1.7.1. Таймеры задержки 44
1.7.2. Короткие неблокирующие задержки 46
1.7.3. Таймеры уведомления 47
1.7.4. Таймауты 51
1.8. ИЗМЕРЕНИЕ ВРЕМЕННЫХ ИНТЕРВАЛОВ 55
1.9. ИСПОЛЬЗОВАНИЕ ДОПОЛНИТЕЛЬНЫХ ИСТОЧНИКОВ ПЕРИОДИЧЕСКИХ АППАРАТНЫХ ПРЕРЫВАНИЙ В КАЧЕСТВЕ ТАЙМЕРОВ 58
РЕЗЮМЕ 59

2. ПОГРЕШНОСТИ ЧАСОВ, ТАЙМЕРОВ И СТАТИСТИЧЕСКАЯ ОБРАБОТКА РЕЗУЛЬТАТОВ ИЗМЕРЕНИЙ ВРЕМЕННЫХ ИНТЕРВАЛОВ 61
2.1. ДИСКРЕТНОСТЬ КОМПЬЮТЕРНОГО ВРЕМЕНИ 65
2.2. АППАРАТНЫЕ ПРЕРЫВАНИЯ 66

2.3. ПОГРЕШНОСТИ ПЕРИОДИЧЕСКИХ ИМПУЛЬСОВ АППАРАТУРЫ 66
2.4. АРХИТЕКТУРНЫЕ ОСОБЕННОСТИ МИКРОПРОЦЕССОРОВ 68
2.4.1. Макрос ClockCycles() 68
2.5. ДОСТИЖИМАЯ НА КОМПЬЮТЕРАХ ТОЧНОСТЬ ХОДА ЧАСОВ И ИЗМЕРЕНИЯ ВРЕМЕНИ 72
2.5.1. Синхронизация системных часов 73
2.5.2. Точная привязка внешних событий к реальному времени 76
2.6. СТАТИСТИЧЕСКАЯ ОБРАБОТКА РЕЗУЛЬТАТОВ ИЗМЕРЕНИЙ ПЕРИОДИЧЕСКИХ ВРЕМЕННЫХ ИНТЕРВАЛОВ 77
2.6.1. Краткое описание используемых статистических методов и критериев 80
2.6.2. Периодические таймеры реального времени 86
2.6.3. Измерение интервалов срабатывания однократных таймеров 98
РЕЗЮМЕ 104

3. ТЕКСТЫ ПРОГРАММ 106
3.1. ПРОГРАММА, ИЛЛЮСТРИРУЮЩАЯ ПРИМЕНЕНИЕ СРЕДСТВ ЗАДАНИЯ, ИЗМЕНЕНИЯ И ПОДГОНКИ СИСТЕМНОГО ВРЕМЕНИ 106
3.2. ПРОГРАММА, РЕАЛИЗУЮЩАЯ АБСОЛЮТНЫЙ ТАЙМЕР ЗАДЕРЖКИ НА ОСНОВЕ ФУНКЦИИ clock_nanosleep( ) 116
3.3. ПРИМЕРЫ ПЕРИОДИЧЕСКИХ ТАЙМЕРОВ, РЕАЛИЗОВАННЫХ СИСТЕМНЫМИ ВЫЗОВАМИ TimerAlarm() И TimerSettime() С УВЕДОМЛЕНИЕМ СИГНАЛАМИ 118
3.4. ПРОГРАММА, РЕАЛИЗУЮЩАЯ АБСОЛЮТНЫЙ ПЕРИОДИЧЕСКИЙ ТАЙМЕР 128
3.5. ПРОГРАММА ПОЛУЧЕНИЯ ИНФОРМАЦИИ О ТАЙМЕРАХ ПРОЦЕССОВ 130
3.6. . ПРОГРАММА, ИЛЛЮСТРИРУЮЩАЯ ИСПОЛЬЗОВАНИЕ ТАЙМАУТОВ ЯДРА 142
3.7. ПРОГРАММА TIME_INTERVAL_MEASURE_3.C, ИЛЛЮСТРИРУЮЩАЯ ПРИМЕНЕНИЕ СРЕДСТВ ИЗМЕРЕНИЯ ВРЕМЕННЫХ ИНТЕРВАЛОВ 148

3.8. ИСПОЛЬЗОВАНИЕ ПРЕРЫВАНИЙ ОТ ПОСЛЕДОВАТЕЛЬНОГО ПОРТА /dev/ser1 КАК ИСТОЧНИКА ПЕРИОДИЧЕСКИХ ИМПУЛЬСОВ..... 152
3.9. ПРОГРАММА “КАЛИБРОВКИ” ДЛИТЕЛЬНОСТИ МАКРОСА ClockCycles() И ОПРЕДЕЛЕНИЯ ВРЕМЕНИ, ПОТРАЧЕННОГО ПРОЦЕССОРОМ НА ВЫПОЛНЕНИЕ ПРОЦЕССА И ПОТОКА..... 158
3.10. ПРОГРАММА ДЛЯ ЗАМЕРА И ЗАПИСИ В ФАЙЛ ДЛИТЕЛЬНОСТИ ИНТЕРВАЛОВ ПЕРИОДИЧЕСКОГО ТАЙМЕРА 162
3.11. ПРОГРАММА ЗАМЕРА ДВУМЯ СПОСОБАМИ СЛУЧАЙНОГО ИНТЕРВАЛА СРАБАТЫВАНИЯ ОДНОКРАТНОГО ТАЙМЕРАС ЗАПИСЬЮ РЕЗУЛЬТАТОВ В ФАЙЛ..... 164
3.12. ЗАГОЛОВОЧНЫЙ ФАЙЛ ДЛЯ ИСПОЛЬЗОВАНИЯ БИБЛИОТЕКИ paired_stat.lib..... 165
3.13. ПРОГРАММА ПОСТРОЕНИЯ ГИСТОГРАММ ДВУХ СЛУЧАЙНЫХ ВЕЛИЧИН С ИСПОЛЬЗОВАНИЕМ ncurses..... 168

4. ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОЙ ПОДГОТОВКИ 175

ЗАКЛЮЧЕНИЕ 184

СПИСОК ИСПОЛЬЗОВАННОЙ И РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ 187

ПРЕДИСЛОВИЕ

Настоящее учебное пособие предназначено для ознакомления студентов с основами функционирования системного программного обеспечения реального времени и обучения навыкам практической работы по программированию приложений реального времени. В качестве платформы разработки (host system) и платформы исполнения (target system) выбрана одна из лучших современных операционных систем реального времени QNX Neutrino (разработчик-фирма QSSL (QNX Software Systems ltd.), Канада [1].

Пособие включает необходимые теоретические сведения, рекомендации по использованию конкретных средств API, а также готовые к компилированию и исполнению исходные тексты программ, иллюстрирующие применение тех или иных приёмов программирования. Особое внимание уделяется использованию функций группы стандартов POSIX, позволяющих писать переносимый код, а также специфическим системными вызовами (по терминологии QNX - вызовами ядра - kernel calls) QNX Neutrino, обеспечивающим дополнительные по сравнению со стандартом POSIX возможности. Затрагиваются также смежные вопросы для формирования у студентов общего представления о структуре и характеристиках систем реального времени и операционных систем реального времени.

Пособие разбито на несколько частей, каждая из которых раскрывает одну из логически законченных тем. В первую очередь издаются следующие части:

Часть 1. Служба времени. Время, таймеры, таймауты

Часть 2. Обработка прерываний.

В дальнейшем предполагается издание в электронном виде дополнительных частей:

- Средства организации многозадачности и управления параллельным выполнением задач.
- Межзадачная коммуникация, средства обмена сообщениями.
- Средства синхронизации.
- Средства организации взаимодействия с устройствами. Менеджеры ресурсов.
- Средства построения человеко-машинного интерфейса.
- Средства создания загрузочных образов и размещение их на целевых машинах.

Пособие предназначено для обучения по курсу “Системы реального времени” студентов всех форм обучения специальностей 230105 “Программное обеспечение вычислительной техники и автоматизированных систем” и 010503 “Математическое обеспечение и администрирование информационных систем”. Может также использоваться при освоении материалов других курсов, в которых программирование строится на базе API стандартов POSIX. Предполагается знание студентами основ языка программирования C.

Освоение всего материала пособия позволяет студентам в рамках аудиторных и самостоятельных занятий создавать работающие модели систем реального времени, в том числе создание загрузочных образов операционной системы для встроенных приложений.

Учитывая современное состояние и особенности учебного процесса, автор везде, где это возможно, предполагает использование студентами некоммерческой версии QNX Neutrino. Задачи, для решения которых возможностей некоммерческой версии недостаточно, оговариваются особо. Учитывая малое количество доступных студентам и рассчитанных на обучение книг по программированию систем реального времени и вместе с тем достаточно большой объём наработок специалистов по этому вопросу, изложенных в специальной печатной литературе и в Интернете, главным образом в форме статей, обсуждений на форумах [2,3] и текстов программ, автор счёл одной из своих задач собрать необходимые материалы вместе в одной книге, так, чтобы их было достаточно для выполнения учебных проектов, и они давали направления для дальнейшего более подробного изучения. Этой же цели служат помещённые в пособии тексты программ с подробными комментаниями и многочисленные ссылки на публикации.

Работа над пособием велась параллельно с участием в образовательной программе для высших учебных заведений фирмы QSSL [1], включающей непосредственное взаимодействие с региональным дистрибьютором операционной системы QNX Neutrino – фирмой SWD Software, г.С.-Петербург [4], организовавшей обучение и предоставившей ценные учебные материалы.

- Служба времени - это одна из самых сложных и загадочных и путаных служб ОС, и не QNX, а любой ОС, исходя из формулировок POSIX.
- Когда возишься в программе с понятием "время" вообще (при измерениях временных интервалов, синхронизации времён и т.п., что здесь на форуме обсуждалось), то готовься ко множеству сюрпризов…

Мнение О.И. Цилюрика – программиста, использующего QNX в реальных проектах, автора многочисленных публикаций по теме QNX, с которым автор настоящего пособия во многом согласен.

ВВЕДЕНИЕ

Служба времени – важная составная часть любой операционной системы (ОС). Для операционных систем реального времени (ОС РВ) отсчёт текущего времени, измерение и задание временнЫх интервалов, быстрое выполнение системных действий с гарантированной верхней границей продолжительности – наиболее востребованные функции. Поэтому для ОС РВ служба времени является одной из важнейших. Предъявляемые к службе времени таких систем требования значительно шире, чем для операционных систем общего назначения. Это прежде всего относится:

1) к предоставлению пользовательским приложениям широкого набора программных интервальных таймеров с возможностью задавать промежутки времени от десятков наносекунд до нескольких лет;

2) к возможности в широких пределах регулировать разрешающую способность системных часов, ответственных за шаг по времени всех периодических действий ОС, включая диспетчеризацию потоков и обновление графического интерфейса.

ОС РВ на базе микроядра QNX/Neutrino в полной мере обладает указанными возможностями, причём интерфейс прикладного программирования (API) содержит как стандартизованные функции библиотеки языка С – главным образом семейства стандартов POSIX, ANSI и UNIX, так и свои собственные системные вызовы и основанные на них функции и макросы, существенно расширяющие возможности программиста по работе с подсистемой времени.

Рассмотренные в первой части пособия вопросы в основном ограничены тематикой курса “Системы реального времени”. Не затрагиваются некоторые практически важные аспекты службы времени, например, установка аппаратных часов реального времени по UTC с возможностью использовать её для правильного отсчёта местного времени любой из установленных на компьютере операционных систем, включая ОС Windows, а также синхронизация системного времени по протоколу NTP. Интересующихся этими вопросами автор отсылает на форумы [2,3,5]. Вместе с тем некоторые темы, касающиеся механизмов работы таймеров, рассмотрены подробно с иллюстрациями (в ущерб лаконичности) для того, чтобы сформировать у читателей правильное представление и лишить распространённых до сих пор иллюзий относительно “точности” времени в операционных системах реального времени. В первой части пособия кроме основной темы - таймеров, измерения временнЫх промежутков и т.п. в прилагаемых программах приведены примеры использования сигналов и средств пакета `ncurses`. Всё это понадобится при разработке итоговых проектов – компьютерных моделей систем управления реального времени. Дополнительно приводится краткая сводка сведений из математической статистики, без которых невозможно правильно понять методику и результаты статистической обработки случайных временнЫх интервалов.

Все приводимые в тексте программы, если не оговорено другое, компилировались и выполнялись в среде QNX Neutrino 6.2.0 PE Patch A и 6.2.1 NC.

Предполагается элементарное знание читателями основ работы с ОС QNX/Neutrino, включая пользование утилитами для работы с файловой системой, компилированием и запуском на выполнение программ из командной строки терминала. Для начинающих хорошим введением являются книги [6,7].

1. СРЕДСТВА СЛУЖБЫ ВРЕМЕНИ QNX/Neutrino И ИХ ИСПОЛЬЗОВАНИЕ

Служба времени обеспечивает взаимодействие ОС с аппаратными таймерами, на основе этого проводит отсчёт внутреннего системного времени и реализует генерирование различного вида периодических и однократных событий с заданными временнЫми промежутками. Предоставляя программисту соответствующий программный интерфейс – системные вызовы, функции, макросы и типы данных (далее обобщённо обозначенные термином “средства”), эта служба является важнейшим компонентом любой операционной системы, в особенности ОС РВ. Знание средств службы времени является необходимым условием программирования приложений реального времени.

1.1. НЕКОТОРЫЕ ПОНЯТИЯ И ТЕРМИНЫ

Относящаяся к службе времени терминология определена частично в справочном руководстве по ОС QNX, частично – в стандартах POSIX. Для правильного понимания терминов ниже приведена их трактовка.

Часы (*clock*) – согласно стандарту [8] - это механизм для измерения течения времени (is a mechanism for measuring passage of time). Часы с наивысшей разрешающей способностью, идущие при включённом компьютере, использующие в качестве источника периодических импульсов аппаратный таймер и временн**у**ю базу CLOCK_REALTIME (см. ниже) отсчитывают системное время и называются **системными** (*system clock*). Каждый отсчёт системных часов называется системным тактом, или тиком (system tick). Термином tick также обозначается интервал дискретности отсчёта (период хода, *clock_period, tick size, size of timer tick*) системных часов. Системные часы распространяются на всю систему и доступны для всех процессов. Дополнительно к системным, согласно [8] могут существовать виртуальные часы, не обязательно связанные с каким-либо аппаратным таймером и не обязательно доступные для всех процессов. Виртуальные часы работают в других временных базах и отсчитывают не реальное, а виртуальное время (*virtual time*).

Системное время (*system time*) – время, отсчитываемое системными часами. Начало отсчёта системного времени - момент начала эпохи UNIX (*UNIX Epoch*) – 00 часов 00 минут 00 секунд UTC (*Uni-*

(*Universal Time Coordinated* – Универсального Координированного Времени) 1 января 1970 г. В ОС QNX внутреннее представление времени в наносекундах имеет тип `uint64_t` и рассчитано на даты до января 2554 г. (2554-1970=584 г. ≈ 2^{64} нс) в отличие от предусмотренного стандартом POSIX 32-х битного представления, обеспечивающего диапазон, ограниченный 2038 годом. Разнообразные функции работы со временем используют свои типы данных, например `time()` возвращает системное время в секундах, используя тип `_Uint32t, clock_gettime()` возвращает системное время в секундах и наносекундах последней неполной секунды, и т.д.

В основу времени UTC (прежде - до 1972 г.-называвшееся GMT – *Greenwich Mean Time* – время по Гринвичу) положена секунда, задаваемая цезиевыми атомными часами. Одна атомная секунда равна 9192631770 периодам колебаний, соответствующих переходу между двумя сверхтонкими уровнями атома цезия 133. Атомное время TAI (*International Atomic Time*) отсчитывается 230 часами в 65 лабораториях времени, расположенных в 30 разных странах [9]. Различие значений TAI между всеми атомными часами в лабораториях времени не превышает 100 наносекунд, погрешность хода не более 1 мкс в год. Атомное время течёт равномерно. В отличие от атомного, астрономическое время, замеряемое как продолжительность между двумя наивысшими положениями Солнца над горизонтом в течение суток на гринвичском (нулевом) меридиане, непостоянно. Различают UTO – усреднённое астрономическое время с коррекцией эффекта некруговой орбиты Земли, UT1 - скорректированное UTO с учётом качания полюсов Земли (достигает величины 15 м), за счёт чего положение нулевого меридиана “плавает” (UT1 имеет неопределённость ± 3 мс в день), и UT2 - скорректированное UT1 с учётом сезонных вариаций скорости вращения Земли (помимо колебательных флуктуаций среднее значение UT2 имеет тенденцию к росту за счёт ежегодного уменьшения скорости вращения Земли на 2×10^{-10} своей величины). Значение UTC получается как показания атомных часов, скачкообразно скорректированные в соответствие с астрономическим временем UT2 [10], а также [11,12]. Разница между астрономическим UT2 и универсальным UTC временами поддерживается не выше 0.5 секунды. Начиная с 1972 г. UTC рассчитывается путём вычитания 1 секунды (*leap second*) из показаний атомных часов 1 января или 1 июля каждого года. Никакие часы – бытовые или компьютерные – не знают

значений UTC до тех пор, пока их показания не будут синхронизированы с UTC. Синхронизация может осуществляться вручную по сигналам точного времени, передаваемым по радио и телевидению. Для ЭВМ предусмотрена автоматическая синхронизация путём получением значений UTC со специальных серверов точного времени, общающихся с компьютерами-клиентами по сети на основе протокола NTP-Network Time Protocol или его упрощённой (Simple) версии SNTP.

ВременнАя база (*timing base*) задаёт особенности поведения системных часов и таймеров в некоторых специфичных условиях, используется как аргумент под именем *clock_id* (идентификатор часов) и входит в вызовы многих связанных со временем функций, включённых в API ОС РВ QNX/Neutrino. Предусмотрены следующие временные базы (см. справку по системному вызову `TimerCreate()`):

CLOCK_REALTIME – стандартные часы, определённые в POSIX. Непрерывно отсчитывают системное время. Могут быть подкорректированы с помощью системных вызовов `ClockAdjust()` и `ClockTime()`. Нередко сравниваются с настенными часами (*wall clock*), стрелки которых можно “подводить”;

CLOCK_SOFTTIME – часы, аналогичные CLOCK_REALTIME, но останавливающие отсчёт времени, когда процессор находится в спящем режиме. Основанный на этой временной базе таймер не “разбудит” спящий процессор в тот момент времени, когда использующее таймер приложение должно разблокироваться. Если процессор не находится в спящем режиме, база CLOCK_SOFTTIME идентична базе CLOCK_REALTIME;

CLOCK_MONOTONIC – определяет часы, отсчитывающие постоянно возрастающее время с заданной частотой. Показания этих часов не могут быть скорректированы.

На момент написания данного пособия (январь 2006г.) временные базы CLOCK_SOFTTIME и CLOCK_MONOTONIC ещё не реализованы, предполагается их внедрение в новых выпусках ОС РВ QNX/Neutrino.

Абсолютное время (*absolute time*) – время срабатывания различного вида таймеров, отсчитываемое от начала UNIX Epoch. С помощью специальных функций обычные календарные дата и время преобразуются в абсолютное время.

Относительное время (*relative time*) – интервал времени срабатывания различного вида таймеров, отсчитываемый от момента запуска таймера.

Календарное время (*calendar time*) – в узком смысле (библиотеки функций QNX/Neutrino) то же, что и системное время, но отсчитываемое в целых секундах. Используется в сочетании с различными временнЫми функциями для определения даты/времени по астрономическому календарю и задания времён срабатывания таймеров;

Местное время (*local time*) **–** то же, что и календарное время, но с учётом часового пояса, заданного на данном компьютере.

Разделённое время (*broken-down time*) – форма представления времени по астрономическому календарю в виде, привычном для человека – год, месяц, день и т.д. Основой является календарное время.

Таймер (программный)– в широком смысле это любая функция API или системный вызов, позволяющие задать интервал времени, по истечении которого будет выполнено определённое действие. Справочная система QNX (см. …/Library Reference/Glossary) трактует понятие “таймер” в узком смысле, привязывая его к вполне определённым функциям библиотеки:

timer: A kernel object used in conjunction with time-based functions. A timer is created via timer_create() and armed via timer_settime(). A timer can then deliver an event, either periodically or on a one-shot basis (таймер: объект ядра, используемый вместе с функциями времени. Таймер создаётся с помощью функции `timer_create()` и запускается с помощью `timer_settime()`. После запуска таймер может доставлять событие либо периодически, либо одноразово).

1.2. ИСТОЧНИКИ ПЕРИОДИЧЕСКИХ АППАРАТНЫХ ПРЕРЫВАНИЙ

В ОС РВ QNX за отсчёт временнЫх интервалов, используемых для собственных системных нужд и программных таймеров в пользовательских приложениях, ответственно микроядро Neutrino. В качестве источника временных импульсов микроядро использует прерывания аппаратного таймера. На IBM PC совместимых компьютерах – это микросхема Intel 8254 и её аналоги. Частота генератора внутренних импульсов чипа i8254 составляет 1.1931816 МГц [13], что соответствует периоду $T_{ат}$ = 838095345 фемтосекунд = 838.095345 нс. Эта

частота уменьшается до заданных допустимых значений путём загрузки ядром соответствующего целочисленного значения в 16-битный регистр-счётчик аппаратного таймера (см. п. 1.4). Наибольшее значение счётчика определяет наименьшую частоту генерируемых аппаратных прерываний. Каждое прерывание соответствует тику системных часов. На компьютерах с архитектурой Intel прерывание от аппаратного таймера имеет наивысший приоритет, и обработка этих прерываний может внести задержку реакции на другие прерывания с более низким приоритетом. Обработка прерываний от таймера в простейшем случае включает обновление счётчика системного времени и занимает на тестовой машине 4 мкс [14]. Если после обновления счётчика истекает время какого-либо из таймеров, обработка прерывания может занять больше времени. Значение 4 мкс хотя и мало, но не является предельным возможным. За счёт архитектурного решения системных часов в ОС Windows CE 5.0, которая также относится к классу ОС РВ, обеспечивается время обработки тика системных часов 2.9 мкс [15].

Для формирования системного времени на этапе загрузки операционной системы QNX/Neutrino использует аппаратные часы реального времени. На платформе x86 обычно они совмещены в одном кристалле с питаемой от батарейки “энергонезависимой” CMOS-памятью (см. справку по утилите rtc).

Для высокоточного измерения коротких временных интервалов используется 64-разрядный регистр-счётчик TSC (time-stamp counter) современных микропроцессоров, инкременируемый с тактовой частотой микропроцессора. На процессорах, начиная с i586, доступ к этому счётчику осуществляется с помощью ассемблерной инструкции RDTSC, библиотека Neutrino имеет соответствующий макрос `ClockCycles()`, определённый в /usr/include/x86/neutrino.h.

Представляет интерес рассмотреть, какие аппаратные источники периодических импульсов для служб времени используют другие операционные системы. Некоторые из них, например FreeBSD, используют TSC в качестве альтернативного аппаратного таймера для отсчёта системного времени. Побочным эффектом этого является сильное отставание системных часов на компьютерах, поддерживающих технологию SpeedStep (преимущественно переносных) при переходе процессора в режим пониженного энергопотребления с уменьшенной тактовой частотой [16,17].

Windows NT сервер 3.51 и 4.0 синхронизирует отсчитываемое аппаратным таймером i8254 системное время с аппаратными часами реального времени CMOS каждый раз, когда разница времени обеих часов превышает 1 минуту, если не установлен режим синхронизации с внешним источником точного времени [18]. ОС Windows в качестве одного из источников временных меток умеет использовать также тактовый генератор контроллера шины PCI [19].

Интересную схему совместного использования таймера i8254 и TSC реализовали разработчики UTIME [20] – расширения службы времени ОС Linux в версии KU Real-Time [21] (KU – University of Kansas). Назначение KURT Linux – поддержка выполнения приложений, относимых к классу "*firm real-time*". Эти приложения требуют соблюдения строгих временных ограничений, аналогичных приложениям жёсткого реального времени (*hard real-time*), и вместе с тем использования сервисов операционных систем, присущих ОС общего назначения и обычно отсутствующих в специализированных ОС РВ. На момент создания KURT в 1997 г. требуемые сервисы содержали ОС РВ LynxOS и QNX, однако из-за дороговизны они не могли быть использованы разработчиками, поскольку их исследовательская организация имела ограниченный бюджет. Основных целей UTIME две: а) увеличить разрешающую способность системных часов с 10 мс в стандартной Linux до 10 мкс и б) модифицировать планировщик (scheduler) таким образом, чтобы задачи класса *firm real-time* имели преимущественное право выполнения для завершения в заданные сроки. Планировщик вступает в работу, проверяет список задач и запускает одну из них при каждом прерывании от аппаратного таймера. Простое уменьшение периода хода (в Linux используется термин *jiffy*) системных часов неприемлемо из-за увеличения накладных расходов по времени на обработку прерываний и перепланирование. Авторы [20] предпочли другой подход, основываясь на существенном различии в частоте хода системных часов и частоте возникновения внешних событий, которые должны успевать обработать задачи класса *firm real-time* – хотя относительные крайние сроки (*deadlines*) от запуска до завершения этих задач могут составлять микросекунды, сами эти задачи воникают реже, чем с интервалом в 1 мкс. Авторы отказались от фиксированной частоты прерываний от аппаратного таймера и применили схему программирования времени следующего прерывания в зависимости от ближайшего времени запуска одной из

firm real-time задач. При этом микросекундная разрешающая способность аппаратного таймера сохраняется, но исключаются частые (с периодом 1 мкс) периодические прерывания, при которых не требуется перепланирования. Отрицательной стороной такой схемы является лишение отдельных служб ядра необходимого для их правильной работы периодического счётчика интервалов времени (периодических "сердцебиений"- *heartbeat*) так, как это происходит в обычной Linux, где каждый *jiffy* = 10 мс. В UTIME этот недостаток компенсируется регулярным отсчётом интервалов *jiffy* с помощью TSC процессора. При каждом прерывании от аппаратного таймера планировщик UTIME сравнивает между собой время до запуска ближайщей *firm real-time* задачи и до достижением счётчиком TSC очередного *jiffy*, и задаёт время следующего аппаратного прерывания в момент наступления ближайшего из этих двух событий. Таким образом, все периодические прерывания аппаратного таймера отрабатываются, как в стандартной Linux с периодом 10 мс, и дополнительно очень быстро планируются задачи *firm real-time* с разрешающей способностью 1 мкс. Побочным эффектом UTIME являются бóльшие накладные расходы – в среднем в 7 раз - на обработку прерываний, поскольку требуется дополнительно рассчитать загружаемое в аппаратный таймер значение счётчика для следующего прерывания.

Некоторые ОС используют дополнительные аппаратные таймеры, доступные на современных материнских платах персональных компьютеров, такие, как таймер высокого разрешения IA-PC HPET [18] и таймер управления питанием ACPI (расширенного интерфейса конфигурирования компьютера и управления питанием) [23]. Таймер ACPI при сравнимой с TSC разрешающей способностью (частота 3.579545 МГц) имеет то преимущество, что его частота неизменна и не зависит от режима энергопотребления компьютера.

Микроядро QNX/Neutrino, в отличие от других операционных систем, не умеет обращаться к таким таймерам. Однако, используя модульный принцип архитектуры QNX и стандартные средства взаимодействия с устройствами, программист самостоятельно может расширить применяемый в своих приложениях набор аппаратных таймеров. В частности, кроме упомянутых, в качестве источников периодических аппаратных прерываний с различными частотами могут быть использованы часы реального времени (обеспечивают частоты прерываний 2^n, n = 1,2,…,13) и микросхема UART (см. п. 1.9). При-

меняются также внешние источники периодических импульсов, например аппаратные таймеры, размещённые на платах расширения [24], или электронные схемы, подключённые к параллельному порту [25]. На современных процессорах (Pentium-3 500 МГц) QNX/Neutrino обеспечивает бесперебойную обработку прерываний от параллельного порта с частотой 10 кГц [25].

Для встроенных систем разработчики QNX/Neutrino формулируют несколько рекомендаций по использованию внешних источников периодических импульсов (осцилляторов) в качестве аппаратной основы системных часов (см. раздел справочной системы Building Embedded Systems/System Design Considerations):

- почти все современные процессоры способны обрабатывать прерывания внешних часов с частотой до 1 кГц, а Pentium и 300 Гц RISC процессоры – и с большей;
- используйте для внешних часов отдельную линию прерываний, чтобы не заставлять ядро тратить время на идентификацию устройства, вызвавшего прерывание;
- нет никаких требований на то, чтобы частота аппаратных часов выражалась “круглым” числом. ОС аккуратно промасштабирует исходную частоту внешних часов для использования во внутренних таймерах и часах, отсчитывающих время суток;
- временно́е разрешение программных таймеров будет не выше, чем у внешних часов.

При совместном использовании нескольких аппаратных источников периодических импульсов следует учитывать, что каждый из них образует свою собственную шкалу времени. Интервалы дискретности этих шкал скорее всего не совпадают и не кратны друг другу (типичный пример – таймер Intel 8254 и TSC микропроцессора). Поэтому время, отсчитываемое по каждой их этих шкал, будет своё, и разница между временами на достаточно длительных промежутках может быть значительна.

Стандартные средства программирования стартового кода QNX/Neutrino дают возможность настроить нужным образом внешний осциллятор на этапе загрузки ОС.

1.3. УСТАНОВКА И КОРРЕКТИРОВКА СИСТЕМНОГО ВРЕМЕНИ

Средства разработки программного обеспечения реального времени обычно работают в среде операционных систем общего назначения (Windows, Linux, Solaris и т.д.) на машинах разработки (*host computer*). Программный код операционных систем реального времени включается в загружаемый на целевую машину (*target computer*) программный образ, выполняющий задачи реального времени. ОС РВ, как правило, работает только на целевой машине. QNX является одним из немногих исключений из этого правила – помимо обычного варианта host – target, она существует также как самостоятельная операционная система и имеет в своём составе средства разработки. Описанные здесь способы установки и корректировки системного времени относятся как раз к самостоятельной ОС QNX, работающей на host компьютере.

Устанавливать системное время имеет право только системный администратор, зарегистрированный в системе как пользователь root. Чтобы при каждом включении компьютера не корректировать время в соответствии с локальной временной зоной, рекомендуется сразу установить аппаратные часы реального времени по UTC. Местное время рассчитывается операционной системой в соответствии с временной зоной (часовым поясом), задаваемой переменной окружения среды TZ. Значение TZ можно посмотреть, запустив утилиту `env` без параметров. QNX имеет также соответствующую конфигурационную системную переменную – _CS_TIMEZONE (значение смотреть командой getconf _CS_TIMEZONE или функцией `confstr`(_CS_TIMEZONE) в программе). Если переменная TZ не определена (так происходит после загрузки ОС), для всех экземпляров TZ используется значение переменной _CS_TIMEZONE. При загрузке ОС переменная _CS_TIMEZONE получает значение, хранящееся в файле /etc/TIMEZONE. Значение этой переменной можно ди-

намически изменять в процессе работы с помощью утилиты `setconf`.

1.3.1. Средства графической среды Photon

Используя утилиту `phlocale` (запускать командной строкой в терминальном окне или одной из кнопок Time & Date или Localization на полке “Configure” Photon’a) графической оболочки (рис.1.1, 1.2), установить:

На вкладке Time & Date

1. Местное время;
2. Флажок “Set Real Time Clock”

На вкладке Time Zone

1. Локальную временную зону. Для Брянска – Russian Federation (Moscow, St.Petersburg);
2. Флажок “The hardware clock uses UTC/GMT”;
3. Флажок “Use Daylight Savings Time”.

Установленные таким образом значения даты, времени и часового пояса сохраняются в конфигурационных файлах системы и действуют при последующих перезагрузках.

Если компьютер должен непрерывно выполнять в течение длительного времени какие-либо программы, содержащие таймеры, скачкообразная корректировка ушедших вперёд или отстающих системных часов приведёт к неправильной работе таймеров. В этом случае целесообразно использовать плавную корректировку системных часов, задав правильное время и интервал I , в течение которого время часы будут подстраиваться под правильное время. Для этого нужно установить флажок “Gradually adjust to new time over the next I minutes”. Текущее значение корректируемого времени отображается в виде текстовой строки вверху окна программы phlocale. Стрелочные часы и поля счётчика времени под ними показывают некорректированное время.

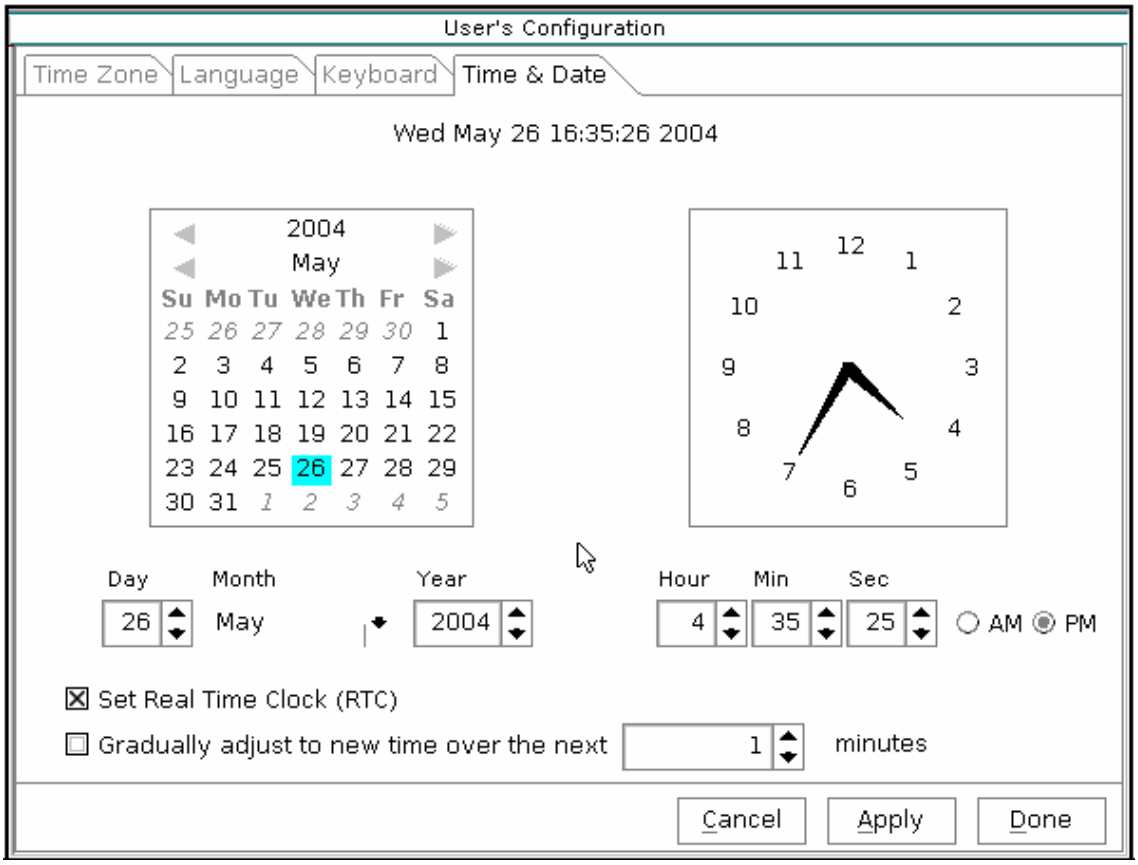


Рис 1.1. Установка даты и времени в среде Photon

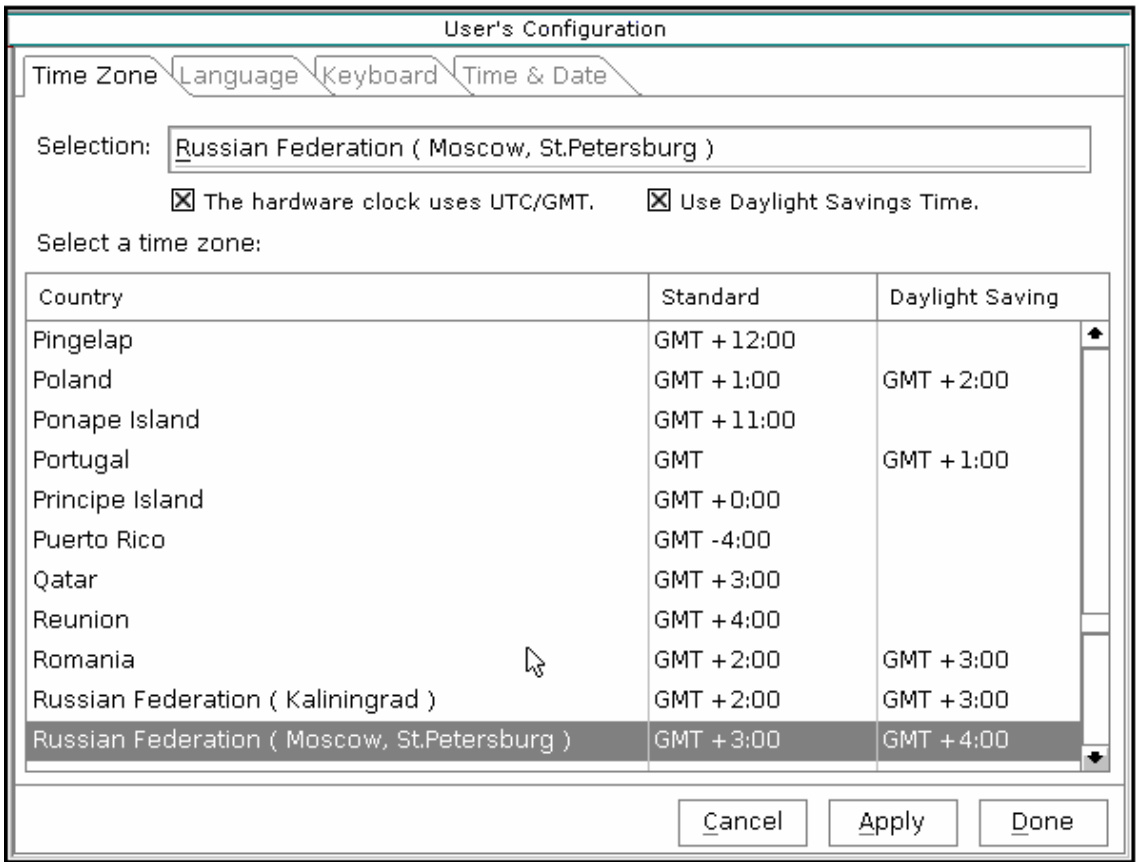


Рис 1.2. Установка временно́й зоны в среде Photon

1.3.2. Утилиты командной строки

1. Установить часы реального времени в соответствии с UTC. Это можно сделать:

а) или изменив время часов реального времени в настройках BIOS setup после включения компьютера, затем привести в соответствие системное время командой

```
#rtc hw;
```

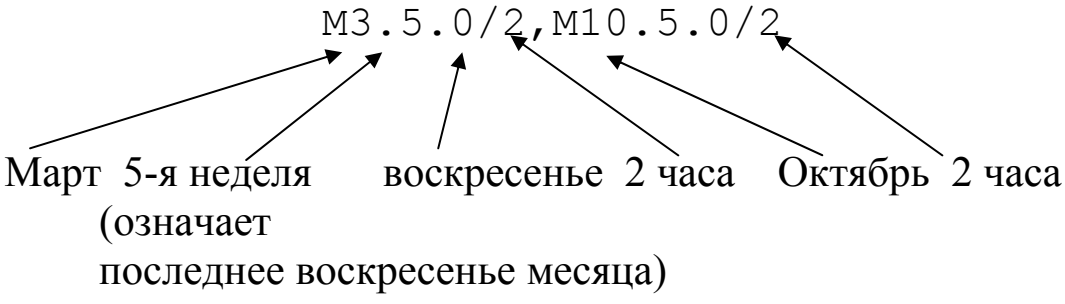
б) или при работающей ОС QNX утилитой date (например,

```
#date –u 31 may 2004 4 31 pm
```

для даты 31 мая 2004г. 4 часа 31 минута пополудни), затем утилитой RTC привести в соответствие показания часов реального времени

```
#rtc –s hw.
```

2. Установить значение переменной окружения **TZ**. Если нужно иметь для всех запускаемых программ в текущем сеансе работы ОС одно и то же значение **TZ**, можно динамически изменить значение конфигурационной переменной _CS_TIMEZONE, не задавая вовсе значения **TZ**. Как уже отмечалось, в этом случае любой экземпляр **TZ** будет принимать значение переменной _CS_TIMEZONE . Если для некоторых программ требуется рассчитывать время в разных часовых поясах, отличных от заданного в конфигурационной переменной _CS_TIMEZONE, значение **TZ** следует задать с помощью утилит `env` или `export` (`#env TZ=MSK-03MSKS-04`, `#export TZ=MSK-03MSKS-04`) или функции `setenv()`. Строка значения **TZ** при задании с помощью утилит `env` и `export` состоит из двух частей. Первая часть содержит две пары параметров для стандартного и летнего времени данного часового пояса. Каждая пара включает символическое обозначение зоны и количество часов, которое надо добавить к местному времени, чтобы получить UTC (пример для Москвы приведен выше). Вторая часть (необязательная) задаёт моменты времени перехода от зимнего к летнему времени и обратно. Принятые сейчас на всей территории РФ правила перехода задаются так:



Полная спецификация временной зоны выглядит так:

MSK-03MSKS-04, M3.5.0/2,M10.5.0/2 .

Более подробные сведения помещены в справочной системе в теме “The TZ environment variable”.

Символическое обозначение временной зоны может задаваться произвольно, но лучше придерживаться обозначений, принятых в самой операционной системе. Для часовых поясов в пределах Российской Федерации значения переменной TZ приведены в табл. 1.1 (информация взята из файла qnxbase.qfs).

Таблица 1.1

Значения переменной TZ для часовых поясов в пределах РФ

Город	Значение переменной TZ
Калининград	KALT-02KALST-03
Москва, С.-Петербург	MSK-03MSKS-04
Самара, Ижевск	IZHT-04IZHST-05
Екатеринбург, Пермь	YEKT-05YEKST-06
Новосибирск, Омск	NOVT-06NOVST-07
Красноярск, Томск	KRAT-07KRAST-08
Иркутск, Улан-Уде	IRKT-08IRKST-09
Чита, Якутск	YAKT-09YAKST-10
Владивосток, Хабаровск	VLAT-10VLAST-11
Магадан	MAGT-11MAGST-12
Петропавловск-Камчатка, Анадырь	ANAT-12ANAST-13

Следует иметь в виду, что установленное командой `env` значение переменной окружения TZ действует только при выполнении той команды/программы, которая при этом запускается. Проверьте, как ведёт себя абсолютный таймер (составленную вами программу назовём abs-timer_TZ.out), время срабатывания которого установлено, например, на 5 минут позже текущего времени. Первый запуск осуществите, использовав значение TZ по умолчанию (часовой пояс Москвы) командой ./abs-timer_TZ.out, сразу после чего во втором терминале запустите ту же программу, задав соседний восточный часовой пояс,например:

```
# env TZ= IZHT-04IZHST-05 ./abs-timer_TZ.out.
```

Поскольку заданное абсолютное время для второго запуска таймера уже истекло приблизительно 1 час назад, таймер сработает сразу же. Команда `export` позволяет установить значение TZ на весь текущий сеанс работы оболочки. В разных консолях и/или окнах псевдотерминала графической оболочки Photon переменные окружения TZ могут иметь разные значения;

3. Проверить правильность установки даты и времени:

- местного - `#date`

- UTC - `#date –u`.

Плавная корректировка системных часов осуществляется утилитой date по умолчанию в случаях, когда "новое время" T_{new} и "старое" T_{old} связаны соотношением

$$(-2.5 \text{ мин} + T_{old}) < T_{new} < 5 \text{ мин} + T_{old}.$$

Если разница между новым и старым временами больше указанной, требуется использовать ключ –S *секунд* утилиты `date`, который задаёт интервал времени в секундах, в течение которого системные часы подстраиваются. Этот интервал должен выбираться таким образом, чтобы при корректировке времени скорость хода часов не возрастала и не уменьшалась бы более чем вдвое. В противном случае системное время изменится скачком немедленно.

1.3.3. Установка и корректировка даты/времени из программы

Пример программы приведен в п. 3.1.

1.4. МОДЕЛЬ СИСТЕМНОГО ВРЕМЕНИ ОС РВ QNX/NEUTRINO

Системное время начинает отсчитываться после загрузки операционной системы. При загрузке инициализирующий модуль (*startup program*) среди прочего заполняет область `qtime` системной страницы данными, необходимыми для работы службы времени (см. справку по теме `qtime`). Исходная информация о текущем реальном времени в момент загрузки берётся по показаниям часов реального времени RTC.

Каждый момент времени, в который наступает обрабатываемое микроядром прерывание от таймера, называется **системным тактом**, или **тиком** (*tick*). Микроядро и вся операционная система существует в дискретной шкале времени с шагом, определяемым периодом хода

системных часов (*clock_period*). Любые два события, случившиеся в течение промежутка времени между двумя соседними тактами, рассматриваются микроядром с точки зрения дискретного компьютерного времени как одновременные. Любые формируемые микроядром интервалы времени для таймеров и любых других нужд формируются из целого числа системных тактов. Никакой заданный интервал не может быть короче системного такта. На машинах, оборудованных микропроцессором с тактовой частотой ≥ 40 МГц, по умолчанию *clock_period* = 999847 нс ≈ 1000000 нс = 1мс. Используя системный вызов `ClockPeriod()`, можно менять период хода системных часов в большую или меньшую стороны (гарантированный минимум составляет 10 мкс на любой машине). Вызов `ClockPeriod()` и POSIX функция `clock_getres()` позволяют также определить значение *clock_period*, не меняя его. Отметим следующие важные закономерности, касающиеся установки микроядром заданной величины системного такта:

1. Микроядро определяет допустимость заданного значения $clock\_period_з$. Если заданное значение на взгляд микроядра не допустимо, например разница между заданным и фактически установленным значением $clock\_period_ф$ велика, микроядро отказывается устанавливать заданное значение и возвращается к фактически установленному $clock\_period_ф$ Пример приведен в табл. 1.2.

Таблица 1.2

Последовательно задаваемые и формируемые микроядром значения *clock_period* (исходное значение по умолчанию 0.999847 мс)

$clock\_period_з$, мс	$clock\_period_ф$, мс
0.999847	0.999847
0.001000	0.999847
0.010000	0.009219
0.000100	0.009219
0.001000	0.009219

2. Если заданное значение $clock\_period_з$ допустимо, справедлива формула

$$clock\_period_з \approx clock\_period_ф = \left\lfloor \frac{T_{am} * D}{1000000} \right\rfloor \text{ нс},$$

где $T_{ат}$- частота внутреннего генератора импульсов аппаратного таймера (см. п.1.2), D – загружаемое микроядром в регистр-счётчик аппаратного таймера целочисленное значение “делителя частоты”, $\lfloor x \rfloor$ - обозначает ближайшее целое, не превосходящее x. Значение D, наряду с другой важнейшей информацией, касающейся службы времени операционной системы, можно получить чтением элементов структуры `qtime` системной страницы Neutrino, используя системный макрос SYSPAGE_ENTRY().

3. Уменьшение значения D приводит к увеличению нагрузки на процессор, связанной с обработкой аппаратных прерываний от таймера. Каждое прерывание сопровождается переключением процессора в пространство ядра и обратно. Хотя затраты времени на переключения контекста в QNX/Neutrino малы, сумма этих затрат, если переключения происходят часто, может вносить заметный вклад в суммарное время решения задачи. В одном из обсуждений в группах новостей *comp.os.qnx* сотрудник QSSL *David Gibbs* приводится пример влияния величины системного тика на время циклической пересылки порции данных 4 кбайта. Запуски тестовой программы, выполнявшейся на 850 мГц процессоре Pentium 3, дали следующие усреднённые длительности Tn пересылки:
   - системный тик 1 мс (устанавливается по умолчанию) – Tn=4850 циклов процессора,
   - системный тик 10 мкс – Tn=6140 циклов процессора, на 26% больше.

   В худшем случае затраты на на системные действия по перепланированию потоков, обновлению графического интерфейса и др. могут “съесть” львиную долю процессорного времени. Поэтому микроядро само определяет допустимую нижнюю границу clock_period. Существует также верхняя граница, определяемая разрядностью регистра-счётчика аппаратного таймера. Численные эксперименты показывают, что крайние значения D составляют 11 и 65535 (=216-1). Таблица 1.3 иллюстрирует сказанное.

Главное назначение ClockPeriod()заключается в изменении зависящих от системного тика параметров реактивности системы с возможностью подстраивания дискретности таймеров и периода перепланирования потоков дисциплины Round-Robin под требования конкретных задач.

Таблица 1.3

Некоторые значения заданного и фактического системного такта и “делителя частоты” D

***clock_period*$_з$**, мс	***clock_period*$_ф$**, мс	D
0.010047	0.009219	11
... ...		
0.009747	0.009219	11
0.009647	0.009219	11
0.010847	0.010057	12
... ...		
0.010247	0.010057	12
0.010147	0.010057	12
... ...		
0.011647	0.010895	13
0.011147	0.010895	13
0.011047	0.010895	13
0.010947	0.010895	13
0.011847	0.011733	14
0.011747	0.011733	14
... ...		
0.998947	0.998171	1191
... ...		
0.998747	0.998171	1191
0.998647	0.998171	1191
0.999747	0.999009	1192
... ...		
0.999147	0.999009	1192
0.999047	0.999009	1192
0.999847	**0.999847**	**1193**
... ...		
0.999847	**0.999847**	**1193**
1.999694	1.998857	2385
2.999541	2.998705	3578
3.999388	3.998552	4771
4.999235	4.998400	5964
5.999082	5.998248	7157

Окончание табл. 1.3

clock_period$_з$, мс	***clock_period***$_ф$, мс	D
6.998929	6.998096	8350
... ...		
47.992656	47.991853	57263
48.992503	48.991701	58456
49.992350	49.991549	59649
50.992197	50.991396	60842
51.992044	51.991244	62035
52.991891	52.991092	63228
53.991738	53.990940	64421
54.991585	54.924578	65535
55.991432	54.924578	65535
56.991279	54.924578	65535
57.991126	54.924578	65535

В обсуждении на форуме [3] отмечено ещё одно возможное применение средств изменения периода хода системных часов – синхронизация времени распределённой системы, в которой необходимо сделать минимальной разницу системных времён нескольких компьютеров с индивидуальными частотами аппаратных таймеров. В определённых пределах этого можно достичь небольшим изменением величины системного тика. Штатного средства корректировки системного времени `ClockAdjust()` здесь недостаточно, т.к. после выравнивания показаний системных часов отсчитываемое ими системное время опять начнёт расходиться из-за отличий в частотах аппаратных таймеров. По отмеченной на форуме [3] аналогии `ClockAdjust()` в этом случае выступает в качестве рукоятки механических часов, с помощью которой вручную переводятся стрелки, а `ClockPeriod()`- в качестве механизма “  ”, ускоряющего или замедляющего ход часов.

В ОС РВ QNX системные часы являются **глобальными для всех процессов** операционной системы. Узнать системное время можно с помощью системного вызова `ClockTime()` или POSIX-ф-ции `clock_gettime()`, переустановить время – используя `ClockTime()` и `clock_settime()`.

Отметим, что стандарты POSIX [26] и более поздний [27] в качестве опциональных вводят также часы процессорного времени для процессов (*Process CPU-time clocks*) и потоков (*Thread CPU-time clocks*). Назначение этих часов – отслеживание времени использования процессора отдельными процессами и потоками на этапах разработки программ (профилирование) и эксплуатации программ (недопущение чрезмерно высокого потребления процессорного времени отдельными потоками), а также возможность реализации специальных дисциплин планирования потоков, например при проведении итеративных вычислений с ограничениями по точности результатов и по времени [26]. Дополнительные часы поддерживаются, если в системе определены конфигурационные константы _POSIX_CPUTIME и _POSIX_THREAD_CPUTIME [28]. В этом случае с помощью POSIX-функций `clock_getcpuclockid()`, `pthread_getcpuclockid()` и системных вызовов `ClockId()`, `ClockId_r()` можно определить идентификатор часов процессорного времени для заданных процесса или потока. Этот идентификатор затем может быть передан в качестве аргумента системному вызову `ClockTime()` для получения процессорного времени. Хотя указанные функции и системные вызовы в библиотеке QNX/Neutrino существуют, их использование не позволяет получать процессорное время, т.к. упомянутые конфигурационные константы в системе не определены (QNX/Neutrino 6.3.0, январь 2006 г.), и сама реализация вызова `ClockTime()` предусматривает в качестве идентификатора часов задание только константы CLOCK_REALTIME. Это, однако, не означает, что операционная система QNX не ведёт статистику использования процессора каждым процессом и потоком. Доступ к этой информации иллюстрирует программа, приведенная в п.3.8.

1.5. КЛАССИФИКАЦИЯ ТАЙМЕРОВ

Существует несколько критериев классификации таймеров [8,29]:

1. **Периодические (интервальные, многразовые)** *(periodic, interval)* **и однократные,** или **разовые** *(one-shot)*.
   Первые многократно генерируют периодические события при истечении заданного интервала времени, вторые могут срабатывать только 1 раз и для нового срабатывания требуют повторного запуска.

2. **Задержки (ожидания)** *(delay)* **и уведомления**
   Таймеры задержки вызывают блокирование потока на заданный интервал времени. Таймеры уведомления генерируют событие по истечении заданного интервала времени, которое может как разблокировать заблокированный в ожидании события поток, так и переключить его с выполнения основных действий на выполнение функции-обработчика события.
3. **Относительные (***relative***) и абсолютные (***absolute***)**
   Абсолютные таймеры срабатывают по достижении заданного абсолютного момента времени, относительные отсчитывают заданный интервал срабатывания с момента своего запуска.
4. **Задержки в режиме занятого ожидания** (*busy waiting*)
   К ним относится семейство функций `nanospin*()`, предназначенных для задания коротких (порядка миллисекунд) временных интервалов без блокирования вызывающего потока и освобождения процессора. Основное их назначение – работа с аппаратными компонентами, требующими небольших задержек между обращениями к их регистрам.
5. Кроме таймеров, можно выделить ещё несколько групп функций API, которые позволяют задавать продолжительность ожидания тех или иных действий. Наряду с ожиданием и последующей обработкой некоторого не связанного со временем события, эти функции предполагают также задание временных границ ожидания (**таймаутов**). Если предполагаемое событие не произошло до истечения таймаута, функция прекращает ожидать его и выполняет другое действие.

Богатый API ОС РВ QNX/Neutrino включает широкий набор библиотечных функций и системных вызовов для реализации в программах различных видов таймеров (табл.1.4).

Все функции, не являющиеся непосредственно системными вызовами (имена этих функций начинаются со строчной буквы), реализованы с помощью соответствующих вызовов ядра. Так, `nanosleep()` является обёрткой для `TimerTimeout()`, `timer_create()` вызывает `TimerCreate()` и т.п. Для многих функций библиотеки QNX/Neutrino исходные коды приведены на сайте фирмы-разработчика QSSL [30].

Не все функции-таймеры можно использовать одновременно в одной программе. Так, разработчики ОС запрещают совмещать

`setitimer()` и `sleep()`, т.к. последняя из функций стирает информацию об пользовательском обработчике генерируемого функцией `setitimer()` сигнала SIGALRM. С другой стороны, `setitimer()` и `alarm()` независимы друг от друга (“The `setitimer()` function is independend of `alarm()`”), могут без проблем сосуществовать в одном процессе (программе).

1.6. КАК РАБОТАЮТ ТАЙМЕРЫ

Для правильного использования программных таймеров, входящих в библиотеку функций языка C QNX/Neutrino, необходимо отчётливо понимать принципы их функционирования. В основе работы любого вида таймеров лежат два положения :

1. Все POSIX функции, связанные с отработкой заданных интервалов времени, используют семантику “**не раньше, чем**”. Это значит, что таймер гарантированно выдержит задержку **не менее** заданного интервала. Никаких требований на верхнюю границу фактической длительности отрабатываемой таймером задержки POSIX не устанавливает. Вот как это требование формулируется в оригинальных документах:

- Раздел стандарта IEEE Std 1003.1, 2004 Edition, посвящённый функциям языка C `timer_gettime()`, `timer_settime()` [29]: “Time values that are between two consecutive non-negative integer multiples of the resolution of the specified timer shall be rounded up to the larger multiple of the resolution. Quantization error shall not cause the timer to expire earlier than the rounded time value”(Значения времени, находящиеся между двумя последовательными целыми, кратными разрешению заданного таймера, должны быть округлены до большего из целых. Ошибка квантования не должна привести к срабатыванию таймера раньше округлённого значения);[1]

[1] Подчёркнуто автором пособия.

Таблица 1.4

Некоторые функции API ОС РВ QNX/Neutrino для создания таймеров

Функция	Группа ф-ций (см. примечание)	Классификация	Единицы задания интервала времени	Возможность использования для таймеров					Дополнительные сведения
				Однократный	Периодический	Задержки	Уведомления	Абсолютный	
sleep()	C	POSIX 1003.1	с	+	-	+	-	-	Блокирует вызывающий поток либо до истечения заданного интервала времени, либо до получения не игнорируемого сигнала (в этом случае возвращает количество недоспанных с/мкс/нс)
usleep()	C	Standard Unix	мкс	+	-	+	-	-	
nanosleep()	A	POSIX 1003.1 (Realtime ext.)	нс	+	-	+	-	-	
clock_nanosleep()	A	POSIX 1003.1j (draft)	нс	+	-	+	-	+	Аналогично предыдущим. Дополнительно можно задать временной базис часов.
TimerAlarm()	B	QNX6	нс	+	+	+	+	-	Периодически посылает вызывающему потоку SIGALRM . Задаётся базис часов.
Alarm()	C	POSIX 1003.1	с	+	-	+	+	-	Аналогично предыдущему, но сигнал доставляется процессу

Окончание табл.1.4

Функция	Группа ф-ций (см. примечание)	Классификация	Единицы задания интервала времени	Возможность использования для таймеров					Дополнительные сведения
				Однократный	Периодический	Задержки	Уведомления	Абсолютный	
ualarm()	C	Standard Unix	мс	+	+	+	+	-	Аналогично предыдущему, но работает периодически
delay()	C	QNX4	мс	+	-	+	-	-	Блокирует поток на заданное количество миллисекунд
nap()	C	Unix	мс	+	-	+	-	-	
napms()	C	Unix	мс	+	-	+	-	-	
timer_settime()	A	POSIX 1003.1 (Realtime ext.)	нс	+	+	+	+	+	При срабатывании доставляют импульсы и др. события процессу или потоку
TimerSettime()	B	QNX6	нс	+	+	+	+	+	
setitimer()	C	Standard Unix	мкс	+	+	+	+	-	При срабатывании таймера доставляется сигнал SIGALRM

Примечания. 1. В табл. приведены только некоторые функции для реализации таймеров. Полный перечень см. в сводке библиотечных функций справочной системы (…/Library reference/Summary of Functions) в группах A)“Realtime timer functions”, B)“Time functions” и C)“Process manipulation functions”.

2. Категории функций расшифрованы там же.

- Раздел стандарта IEEE Std 1003.1, 2004 Edition, посвящённый функциям языка C, описывающим интервальные таймеры в стиле операционной системы UNIX `getitimer()`, `setitimer()` [31]: “ For each interval timer, if the requested timer value requires a finer granularity than the implementation supports, the actual timer value shall be rounded up to the next supported value ” (для каждого интервального таймера, если заданное время срабатывания требует более мелкой дискретизации, чем поддерживает конкретная реализация, действительное значение времени срабатывания должно быть округлено до ближайшего большего поддерживаемого значения);[2]
- Раздел справочной системы QNX/Neutrino по функции `nanosleep()`: The suspension time may be longer than requested because the argument value is rounded up to be a multiple of the system timer resolution or because of scheduling and other system activity (время задержки может быть длинней, чем заданное, т.к. значение аргумента округляется вверх до ближайшего целого, кратного разрешению системного таймера, или из-за диспетчеризации или других действий системы).[1]

Незнание указанных обстоятельств нередко является причиной неправильного понимания временно́го поведения программного обеспечения реального времени (ПО РВ), когда предполагается, что после истечения заданного интервала ОС РВ с минимальной задержкой выполнит заданное действие. Действительно, ОС РВ предоставляет программисту широкий набор средств для обеспечения такого временно́го поведения критических по времени участков кода, в том числе очень малые накладные расходы по времени на выполнение системных действий, например переключение контекста, обработку прерываний, передачу сообщений и т.д. Однако правильное применение этих средств является целиком обязанностью программиста и, отнюдь, не прикладывается ОС РВ к любому таймеру по умолчанию.

2. Как указывалось выше, любые связанные с отсчётом времени действия операционная система осуществляет в узлах своей дискретной шкалы системного времени, т.е. только в моменты “тиков” системных часов. Если пользовательская программа требует отсчёта интервалов времени, не кратных системному тику, а также вследствие конкурентной многозадачности и особенностей функционирования

[1] Подчёркнуто автором пособия.

аппаратной части компьютера, между заданными и фактическими интервалами возникает разница, которая называется флуктуациями отсчёта времени, или *timing jitter* (*jitter*-“дрожание”). Используются также термины “локальный временной дрейф” (*local timing drift*) и “ошибка квантования” (*quantization error*).Для разового таймера под этим понимается промежуток времени между моментом запуска таймера и ближайшим следующим системным тиком, начиная с которого ОС приступает к отсчёту заданного интервала, для периодического таймера дополнительно – разность фактического времени срабатывания таймера и заданного при каждом срабатывании.

Джиттер подразделяется на детерминированную (собственно ошибку квантования) и случайную составляющие. Детерминированная составляющая определяется характеристиками аппаратного таймера, алгоритмом работы программного таймера и при полностью известных характеристиках программного окружения может быть точно рассчитана для каждого срабатывания таймера. В натурных условиях работы системы реального времени это практически не осуществимо, главным образом по причине переменчивой внешней среды, с которой система взаимодействует, поэтому детерминированная составляющая фактически проявляет себя как псевдослучайная и выглядит как ошибки отрабатываемых таймером интервалов, хотя ошибкой в истинном смысле этого слова не является. Случайный компонент джиттера обусловлен случайными физическими флуктуациями в аппаратуре компьютера и взаимодействием внутренних неизвестных программисту механизмов ядра и аппаратуры, поэтому может быть проанализирован и предсказан только в вероятностно-статистическом смысле (см., например, статью [32]). Последнее, с учётом изложенного, относится и к джиттеру в целом как сумме детерминированной и случайной компонент. Имеются программные и аппаратные способы уменьшения джиттера, причём, поскольку запуск таймера в реальных условиях работы реактивных (по отношению к внешним событиям) систем асинхронен по отношению к системной шкале времени, локальный временной дрейф принципиально не устраним. Величина джиттера таймеров в наихудших условиях функционирования системы является одной из важных характеристик предсказуемости ОС РВ [33].

Рассмотрим, как указанные положения влияют на работу таймеров.

1.6.1. Интервальные таймеры уведомления

Разберём поведение периодического интервального таймера уведомления, задаваемого функциями `timer_create()` и `timer_settime()` с параметрами

```
timer_create(CLOCK_REALTIME, &event, &timer_id);
itime.it_value.tv_sec = 0;
itime.it_value.tv_nsec = start_interval;
itime.it_interval.tv_sec = 0;
itime.it_interval.tv_nsec = timer_period;
timer_settime(timer_id, 0, &itime, NULL);
```

Для дальнейшего анализа удобно представить величины `start_interval` и `timer_period` в долях системного тика.
Отметим следующие важные моменты работы таймера:

α) таймер начинает отсчёт и срабатывает только в моменты системных тиков;

β) промежуток времени от запуска до 1-го срабатывания равен

$$\lceil \text{start\_interval} \rceil + \text{jitter}.$$

($\lceil x \rceil$ обозначает ближайшее целое, не меньшее x,

$\lfloor x \rfloor$ - ближайшее целое, не превосходящее x)

Величина jitter находится в интервале $0 \le \text{jitter} < \text{tick}$;

γ) моменты последующих “периодических” срабатываний таймер выбирает таким образом, чтобы обеспечить в среднем соблюдение заданного timer_period. Этот период выдерживается точно при каждом срабатывании только если timer_period – целое число системных тактов. Если timer_period – не целый, таймер чередует последовательные серии интервалов срабатываний величиной $\lceil \text{timer\_period} \rceil$ и $\lfloor \text{timer\_period} \rfloor$;

δ) последовательности чередующихся в определённом порядке серий интервалов срабатываний таймера строго периодически повторяются. Период повторений (далее “большой период”, или big_period) равен наименьшему общему целому кратному (НОК) величин tick и timer_period (предполагается, что timer_period выражен в долях tick) и может быть вычислен по формуле

$big\_period = n * \frac{1}{timer\_period - 1} * timer\_period$ системных тактов.

Целое *n* подбирается таким, чтобы значение *big_period* было точно целым или достаточно близким к целому. Примеры приведены в табл. 1.5.

Примеры временных последовательностей срабатываний таймера приведены на рис.1.3. Аналогично ведёт себя интервальный таймер в стиле ОС UNIX, создаваемый и запускаемый функциями `getitimer()`, `setitimer()`, хотя последовательность чередования интервалов срабатываний может быть иной.

Приведенные соображения можно сформулировать в виде практически важных выводов:

а) интервальный таймер работает как строго периодический только если величина timer_period равна величине tick.

Таблица 1.5

Соотношение заданного периода срабатывания таймера, “большого” периода и множителя `n` интервального таймера (все периоды в долях системного тика)

Timer_period	n	Big_period
1.4	2	7
2.3	13	23
$\frac{1000000нс}{999847нс} = 1.000153...$	1	6535.94≈6536

б) В остальных случаях наблюдается джиттер времени срабатывания относительно заданных моментов. Фактический строго повторяемый период срабатываний здесь равен

big_period = целоеНОК(tick, timer_period).

Интервалы промежуточных срабатываний в пределах “большого периода” чередуются по величине таким образом, что усреднённое их значение равно timer_period. Количество этих интервалов равно big_period / timer_period. В тех случаях, когда tick и timer_period -

взаимно простые числа, big_period может достигать больших значений (отметим, что величина системного такта по умолчанию 999847 нс имеет два простых делителя : 467 и 2141). Разумеется, значимая разница в фактических интервалах срабатывания имеет место только когда tick и timer_period имеют один порядок. Если timer_period >> tick, этой разницей можно пренебречь.

Критически важные случаи возможны при близких значениях timer_period и tick [34]. Рассмотрим это более подробно на воображаемом примере. Пусть управляющий компонент системы реального времени должен отслеживать изменение некоторой физической величины, например давления в химическом реакторе, и по определённому количеству последовательных отсчётов рассчитывать и вырабатывать управляющее воздействие, поддерживающее давление в допустимых пределах. Скорость изменения давления может быть достаточно большой, так что интервал отсчёта должен составлять одну миллисекунду. В программном обеспечении управ ляющей ЭВМ период интервального таймера установлен величиной 1000000 нс = 1 мс, 1 системный тик равен 999847 нс [34]. В соответствии с рассмотренным алгоритмом работы таймер определённое время будет строго отсчитывать близкие к заданному интервалы, равные 1 такту, причём **i**-тое по счёту срабатывание происходит при условиях

$$\begin{aligned} &i^* \text{timer\_period} > i^*\text{tick}, \\ &i^* \text{timer\_period} < (i+1)^*\text{tick} \end{aligned} \qquad (*)$$

Поскольку интервал отсчётов выдерживается строго периодично, управляющая ЭВМ будет успевать отслеживать все изменения давления и вовремя выдавать исполнительным устройствам команды на его регулирование. При этом погрешность между реальным и заданным временами срабатывания таймера на каждом цикле будет увеличиваться на 1000000 – 999847 = 153 нс.

Опасная ситуация возникнет на шаге i_{big_period}, когда за счёт накопления погрешности сменится знак второго из неравенств (*)

$$i_{big\_period}*\text{timer\_period} > (i_{big\_period}+1)*\text{tick}$$

что соответствует $i_{big\_period} = \left\lceil \frac{tick}{timer\_period - tick} \right\rceil$.

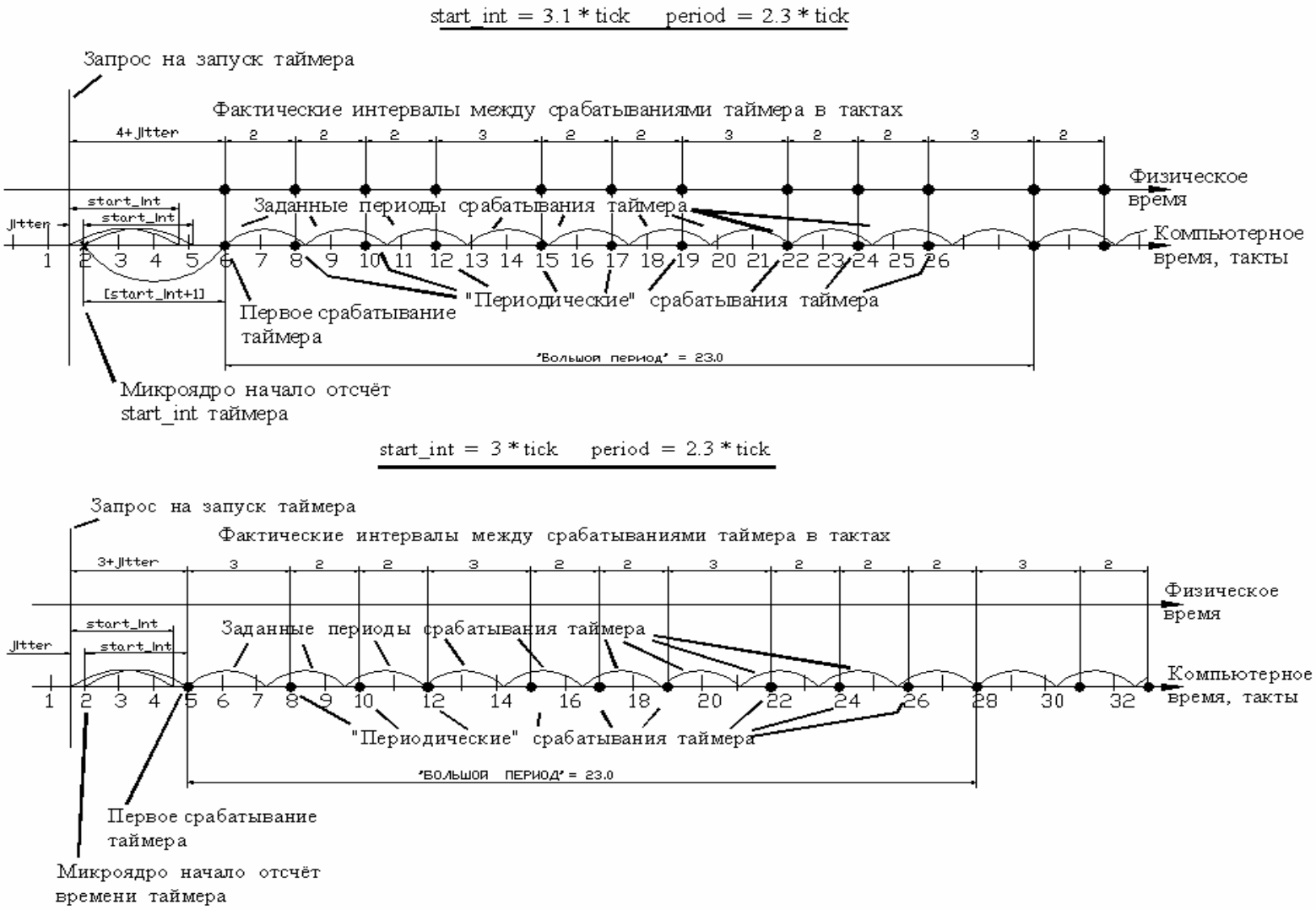


Рис. 1.3. Временные диаграммы работы интервальных таймеров

Для заданных параметров это значение составляет i_{big_period}=65535. При этом значении i срабатывания таймера не произойдёт (отметка B на рис. 1.4) и два соседних срабатывания (метки A и C) окажутся разделены интервалом 2*timer_period в полном соответствии с требованиями γ) и δ) алгоритма таймера. Такие пропуски срабатываний будут повторяться периодически с большим периодом i_{big_period} системных тактов, каждый период будет заканчиваться сменой знака второго из неравенств (*). Автор статьи [34] сравнивает это с биениями суммы двух гармоник с близкими частотами, при котором амплитуда периодически уменьшается до нуля.

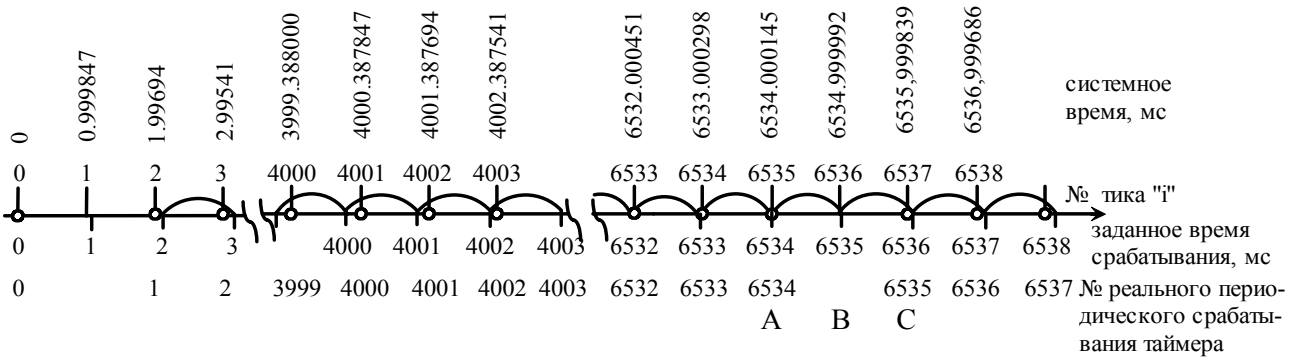


Рис 1.4. Временная диаграмма работы интервального таймера при близких значениях периода таймера и системного тика (первое срабатывание таймера совмещено со вторым тиком, хотя в реальности может быть больше)

Потеря одного из очередных значений критически важной с точки зрения безопасности величины (давления) может привести к выработке управляющей ЭВМ неправильной команды для регулятора давления и, как следствие, катастрофическим последствиям – взрыву. Единственный способ избежать этого – задать timer_period = tick. С другой стороны, для каких-то практических задач может быть полезным периодическое повторение двойных интервалов между генерируемыми таймером событиями.

1.6.2. Таймеры задержки

Чтобы таймер задержки удовлетворял требованию стандарта POSIX “**не раньше, чем**”, заданный интервал срабатывания таймера timer_interval, если он не кратен системному тику, округляется в большую сторону до ближайшего целого числа тиков. Таким образом, без учёта возможных дополнительных затрат времени на ожида-

ние в очереди к процессору в рамках заданной дисциплины планирования промежуток времени от запуска до срабатывания составляет

$$\text{jitter} + \lceil \text{timer\_interval} \rceil .$$

Дополнительное увеличение интервала может вызвать:

- Вре́менная блокировка прерываний одной из пользовательских программ или ядром, в результате чего очередной тик системных часов может быть потерян.
- Занятость процессора другими потоками с тем же или более высоким приоритетом в рамках дисциплины планирования FIFO (потенциально не ограниченная по времени) или Round Robin (ограниченная 4-мя тиками при одинаковом или потенциально неограниченная при большем приоритете конкурирующих за процессор потоков).

Использование таймера задержки в циклах приводит к появлению накопленного временного дрейфа (*cumulative drift*), в результате чего реальное время ожидания может значительно отличаться в большую сторону от предполагаемого. Общий случай реализации периодических действиий путём циклического вызова однократного таймера и возникающий при этом накопленный временной дрейф проиллюстрированы рис.1.5 [35].

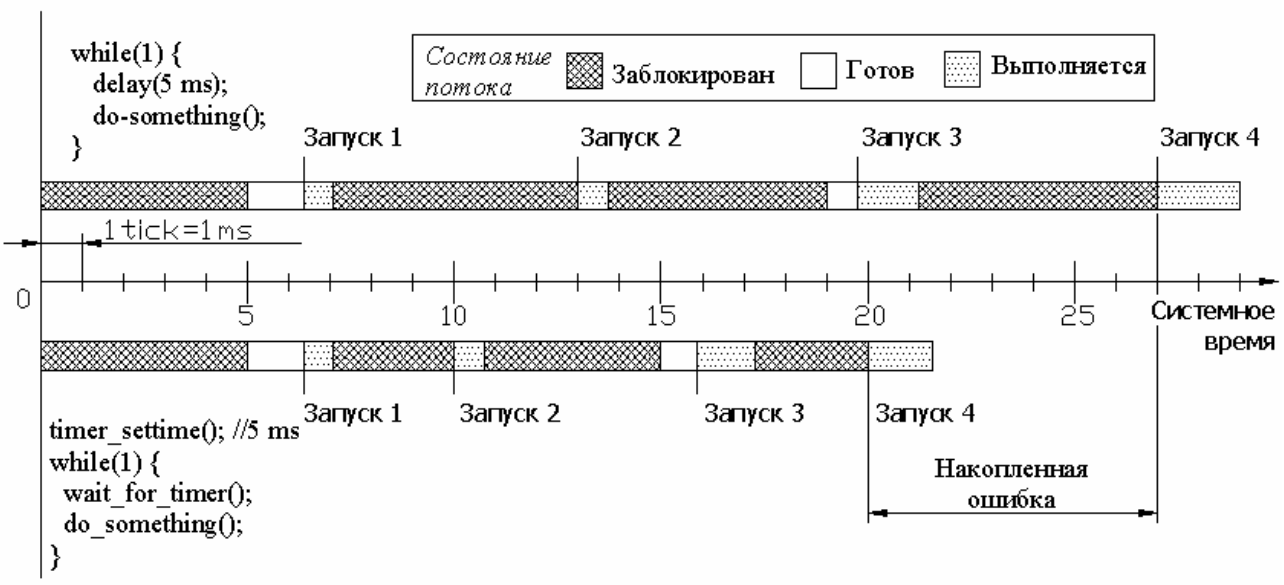


Рис. 1.5. Накопленный дрейф таймера при задании периода с помощью таймера задержки

Особенно большие относительные погрешности возникают при заданных малых значениях задержки, близких к значению системного тика [36]. Так, время выполнения следующей программы с аргументом 1000000=10^6нс вместо ожидаемого в 1 секунду = 1000раз*10^6нс/(10^9нс/с) займёт приблизительно 3 секунды, может быть, и больше. Объясняется это так. Заданная задержка находится в пределах

(999847нс = 1 тик) < (1000000нс = 1.000153 тика) < 2 тик.

Поэтому функция nanosleep() будет отрабатывать задержку $\lceil 1.000153 \text{ тика} \rceil$ = 2 тика = 1.999694 мс,т.е. на каждом шаге цикла фактически будем иметь ≈2 мс вместо заданной 1 мс.

```
#include <time.h>
#include <stdlib.h>
#include <sys/neutrino.h>
#include <inttypes.h>
#include <sys/syspage.h>

#define BILLION 1000000000L

int main( int argc, char **argv )
{
    struct timespec rqtp, res;
    _uint64 nanoseconds;
    uint64_t cps, cycle1, cycle2, ncycles;
    float sec, tick_elapsed;
    int i;

    nanoseconds = strtol( argv[1], NULL, 0 );
    nsec2timespec( &rqtp, nanoseconds);
    cycle1=ClockCycles( );
    for (i=0; i<1000; i++) nanosleep( &rqtp,  NULL );
    cycle2=ClockCycles( );
    ncycles=cycle2-cycle1;
    cps = SYSPAGE_ENTRY(qtime)->cycles_per_sec;
    sec=(float)ncycles/cps;
    clock_getres( CLOCK_REALTIME, &res);
    tick_elapsed = sec*BILLION / res.tv_nsec;
    printf("Elapsed %f s  %f tick\n",sec,
           tick_elapsed);
```

```
    return EXIT_SUCCESS;
}
```

Дополнительную добавку даёт локальный временной дрейф (*jitter*). На первом шаге он составляет случайную величину в интервале от 0 до одного такта. Начиная со второго шага запуск функции nanosleep() синхронизирован с системным тиком, поскольку погружение потока в сон и его пробуждение, т.е. операции перепланирования, осуществляются ядром только в моменты времени, совпадающие с тиками, поэтому jitter составляет ≈1 такт. Временная диаграмма работы программы приведена на рис. 1.6.

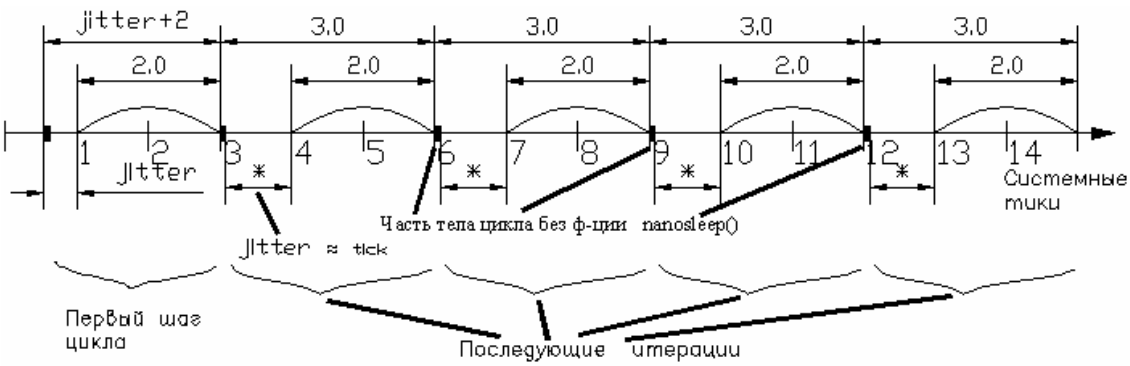


Рис 1.6. Временна́я диаграмма работы таймера задержки при близких значениях задержки и системного тика

Накопленный временной дрейф за 1000 шагов цикла составит [(1.999694-1.0) мс + 0.999847 мс] * 1000 ≈ 2 с,что с добавлением “полезной” заданной задержки даст общее время выполнения ≈3 с.

Реализация периодического многоразового таймера путём циклического вызова однократной задержки всегда сопряжена с накоплением ошибок (*cumulative drift*). Это, на первый взгляд, очевидное обстоятельство было далеко не сразу должным образом оценено и использовано разработчиками программного обеспечения. Например, первый стандарт языка Ада (Ада-83) в качестве таймера включал только оператор `delay` (аналог POSIX-функций `delay()`, `sleep()` и т.п.). И только в языке Ада-95 была введена конструкция `delay until` (аналог абсолютного POSIX-таймера) [10]. Отметим, что одна из целей создателей языка Ада состояла в возможности его применения для программирования систем реального времени. Поэтому стандарт языка содержит целый раздел – приложение D (см., например [37], посвященный системам реального времени, а таймеры (`delay, delay until`) и другие необходимые средства явля-

ются непосредственно операторами языка [38] в отличие от языков C/C++, где они реализованы в виде библиотечных функций. То же самое относится к средствам реализации многозадачности – аналог POSIX-потока в языке Ада называется задачей (task), создание, синхронизация и коммуникация задач осуществляются операторами самого языка. Существует компилятор GNU Ada (GNAT), который можно запускать в QNX/Neutrino (подробности – на форумах [2] [3], сайтах [39] и [40]).

Более подробные результаты исследования фактических интервалов блокирования потоков на заданное время с помощью таймеров задержки с учётом конкурентного выполнения нескольких потоков на однопроцессорной машине приведены в [41].

Зависимость времени блокирования потока таймером от его приоритета иллюстрируется рис. 1.7 и 1.8 [35]. Рассматривается чередование на процессоре двух потоков – t1 и t2 с карусельной (*Round Robin, RR*) дисциплиной диспетчеризации, при которой каждому потоку отводится на процессоре квант времени (*time slice*), равный 4 тикам. Таймер, реализуемый функцией `nanosleep()`, должен блокировать поток t2 на 10.5 тиков. В первом случае (см. рис. 1.7) оба потока имеют одинаковый приоритет 10. После срабатывания таймера по прошествии $\lceil 10.5 \rceil = 11$ тиков от его запуска поток t2 ставится ядром в хвост очереди потоков с приоритетом 10 и ожидает, пока поток t1 выработает предоставленный ему квант времени, пробуждаясь через 12.5 единиц системного времени от момента запуска таймера. “Ошибка” (как указывалось выше, на самом деле не ошибка, а неотъемлемое свойство системы) времени блокирования составляет в этом случае 12.5-10.5=2 тика. Во втором случае (см. рис. 1.8) поток t2 после срабатывания таймера не ждёт, пока t1 исчерпает свой квант, а сразу вытесняет его, поскольку имеет более высокий приоритет. “Ошибка” времени блокирования составляет здесь 1 тик. Рассмотренная на рис. 1.7 и 1.8 идеализированная ситуация учитывает конкуренцию только двух потоков. В реальных системах их, как правило, больше, и приоритеты у каждого могут быть свои. Поэтому реальная “ошибка” задержки может быть значительно больше, если поток t2 будет ждать завершения потока с приоритетом >11 или дисциплиной диспетчеризации FIFO. Ещё увеличит “ошибку” обработка ядром ап-

паратного прерывания или сигнала, которые “сверхприоритетны”. Самые большие задержки могут возникать при совместном использовании несколькими потоками общих ресурсов помимо процессора. Возникающее при этом явление инверсии приоритетов и способы уменьшения его негативного влияния рассмотрены в частях 3,4,5 настоящего пособия.

1.7. ИСПОЛЬЗОВАНИЕ ТАЙМЕРОВ В ПРОГРАММАХ

Таймеры являются объектами микроядра. Это значит, что независимо от состояния, в котором находится поток, ожидающий уведомления от таймера, отсчёт заданного интервала времени микроядром ведётся (“Солдат спит – служба идёт”). Поток будет готов принять уведомление и выполнить необходимые по времени действия лишь в том случае, когда к моменту срабатывания таймера не существует конкурентов за обладание процессором и сам он не заблокирован каким-либо не связанным со временем системным вызовом, в противном случае его реакция может запоздать, несмотря на своевременное срабатывание таймера и весьма малые накладные расходы ОС на системные операции. Именно поэтому в сложном многопоточном реактивном приложении с несколькими таймерами, обслуживающем случайные внешние события, нельзя быть уверенным в точном выдерживании системой заданных временнЫх интервалов.

1.7.1. Таймеры задержки

Наиболее просто использовать таймеры задержки (функции `sleep()`, `delay()` и им подобные), блокирующие процесс или поток. Необходимо только помнить, что все функции с фрагментом имени “sleep” могут разблокировать поток не только по истечении заданного интервала времени, но и при доставке сигнала, если предусмотрена обработка сигнала специальной функцией-обработчиком или сигнал должен завершить процесс. Пример программы с использованием nanosleep() приведен в пп. 1.6.2. Второй пример использует более богатую возможностями функцию `clock_nanosleep()` (п.3.2.).

Рис. 1.7. Зависимость времени блокирования потока таймером от приоритета потока (t2 = t1)

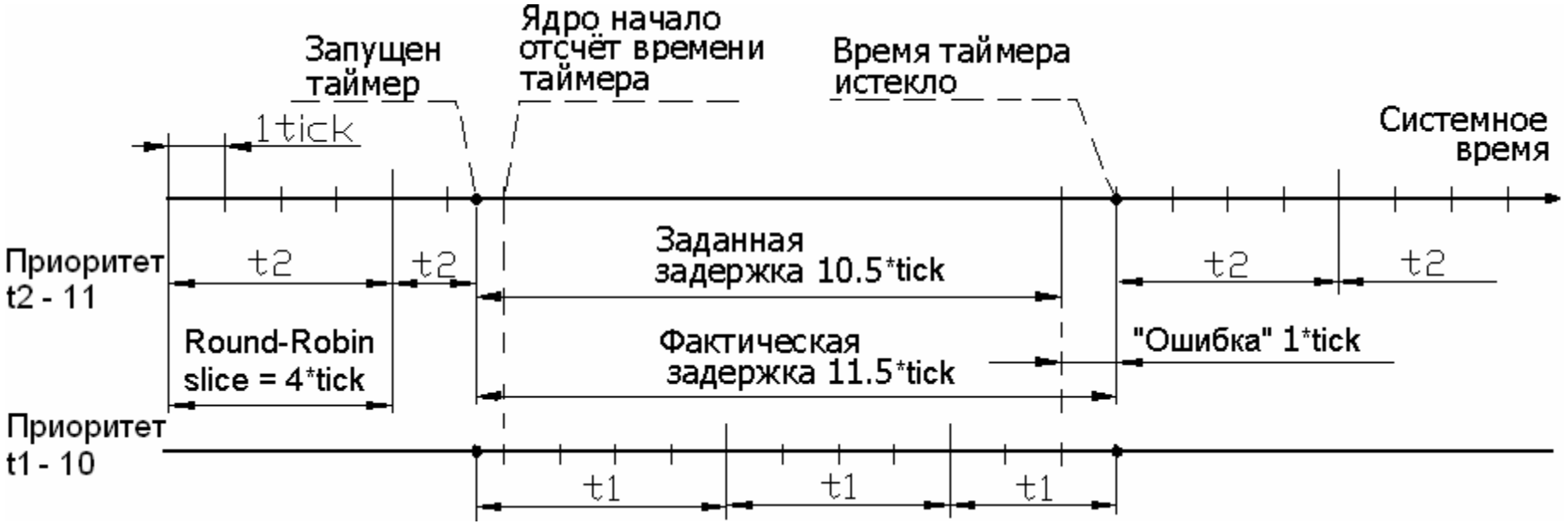


Рис. 1.8. Зависимость времени блокирования потока таймером от приоритета потока (t2 > t1)

1.7.2. Короткие неблокирующие задержки

Основное их назначение – работа с аппаратными компонентами, требующими небольших задержек – порядка единиц миллисекунд - между обращениями к регистрам аппаратуры. Реализованы как циклы `do...while` занятого ожидания (*Busy-wait*) с заданной продолжительностью выполнения.

Семейство включает группу функций `nanospin*()`. Перед первым обращением к функциям занятого ожидания необходимо откалибровать по времени величину задержки одного шага цикла. Осуществляется калибровка функцией `nanospin_calibrate(disable)`. Величина параметра `disable`, равная 1, задаёт режим врéменного блокирования аппаратных прерываний, интервалы обработки которых влияют на точность калибровки. Значение параметра `disable`, равное 0, требует явным образом заблокировать прерывания перед вызовом ф-ции `nanospin_calibrate()`.

В качестве примера использования этих библиотечных функций рассмотрим фрагмент программы-драйвера встроенного 2D/3D графического контроллера чипсета i810 (исходный код функций приведен в комплекте разработки драйверов (DDK) QNX/Neutrino). Функции этого фрагмента реализуют передачу данных по шине I^2C.

Функция калибровки:

```
Void disp_usecspin_init()
{
    #ifdef __QNXNTO__
        nanospin_calibrate(1);
    #else
        disp_usecspin(0);
    #endif
}
```

Функция реализации задержки:

```
void disp_usecspin(unsigned usecs)
{
    #ifdef __QNXNTO__
        nanospin_ns(usecs*1000);
    #else
    if (loops_per_usec == -1)
```

```
        disp_init_delayloop();
        spin(loops_per_usec * usecs);
    #endif
}
```

Фрагмент функции пересылки данных - передача одного байта:

```
for (i = 0; i < 8; i++) {
/* send each bit, MSB first */
    if (data[byte] & (0x80 >> i))
        I810_DATA_HIGH;
    else
        I810_DATA_LOW;
    I810_CLOCK_HIGH;
    disp_usecspin(I2C_WAIT);
    I810_CLOCK_LOW;
    disp_usecspin(I2C_WAIT);
}
```

1.7.3. Таймеры уведомления

Среди интервальных таймеров наибольшими возможностями обладают таймеры, входящие в группу “Realtime timer functions” (см. табл. 1.4), создаваемые с помощью функции `timer_create()` или системного вызова `TimerCreate()`. Эти таймеры лежат в основе периодических временнЫх программ-серверов (см., например [13] и [42]). Ввиду практической важности и широкой применимости этих таймеров рассмотрим их более подробно. За основу возьмём POSIX-функцию `timer_create()` и связанные с ней.

Общая схема использования таймеров следующая:

1. **Создать таймер.** Синтаксис:

```
int timer_create( clockid_t clock_id,
                  struct sigevent * evp,
                  timer_t * timerid ).
```

Функция возвращает идентификатор созданного таймера по адресу *`*timerid`*. Этот идентификатор используется затем всеми другими функциями, работающими с таймером.

Задаваемые аргументы (в порядке их расположения в вызове функции):

– Временна́я база *clock_id* используемых таймером системных часов (см. п. 1.1).

– Событие, генерируемое ядром при срабатывании таймера. Если оно не задано (значение NULL), по умолчанию используется сигнал SIGALRM. Явное задание события предусматривает заполнение POSIX-структуры `sigevent`. Разработчики ОС QNX/Neutrino рекомендуют задавать уведомление одного из следующих типов:

- SIGEV_SIGNAL - сигнал;
- SIGEV_SIGNAL_CODE – сигнал с заданным 8- битным кодом и 32-битным значением;
- SIGEV_SIGNAL_THREAD - сигнал с заданным 8-ми битным кодом и 32-х битным значением, посланный определённому потоку (тому, в котором тип уведомления задаётся);
- SIGEV_PULSE – импульс с заданным приоритетом, 8-ми битным кодом и 32-х битным значением;
- SIGEV_UNBLOCK – разблокирование потока;
- SIGEV_INTR – генерирование прерывания.

Для удобства заполнения структуры `sigevent` удобно использовать предопределённые макросы, например, `SIGEV_SIGNAL_INIT()`. Более подробная информация приведена в справочной системе по структуре sigevent. Созданный таймер находится в неактивном состоянии до тех пор, пока не будет запущен.

Очень важно чётко представлять себе варианты взаимодействия целевого потока, который должен реагировать на срабатывание таймера, и генерируемого таймером события. В зависимости от способа обработки целевым потоком уведомления от таймера последний может выступать как синхронный или асинхронный по отношению к целевому потоку. Если целевой поток блокирован в ожидании уведомления и таймер разблокирует поток, такой таймер может рассматриваться как синхронный по отношению к потоку (например, поток блокирован на функции `MsgReceive()` в ожидании импульса от таймера). Если выполнение целевого потока прерывается уведомлением от таймера и запускается специально предназначенный для моментов срабатывания код, такой таймер является асинхронным по отношению к потоку (напри-

мер, уведомление – сигнал и обработчик сигнала в целевом потоке).

2. **Задать временные параметры работы таймера и запустить таймер**. Синтаксис:

```
int timer_settime( timer_t timerid,
                   int flags,
                   struct itimerspec * value,
                   struct itimerspec * ovalue );
```

– Параметр flags задаёт тип таймера: значение 0 означает относительный таймер, TIMER_ABSTIME – абсолютный.

– Аргумент value устанавливает время срабатывания таймера. Структура itimerspec содержит два члена: `struct timespec it_value` - задаёт время первого срабатывания (относительное или абсолютное). Если задано значение 0, таймер не запускается;

– `struct timespec it_interval` – задаёт интервал последующих срабатываний. Если задано значение 0, таймер срабатывает однократно.

Структура `struct timespec` определена так:

```
struct timespec  {
 time_t    tv_sec;   // полные секунды
 long      tv_nsec;  /* часть последней неполной секунды
                        в наносекундах */
}
```

3. **Получить сведения о состоянии таймера**, если они нужны. Могут быть использованы функции:

- `timer_gettime()` - возвращает промежуток времени, оставшийся до срабатывания таймера;
- `timer_getoverrun`()- возвращает количество “необработанных” срабатываний таймера, возникших в промежутке между моментом генерирования таймером сигнала и моментом доставки (delivery) этого сигнала, называемое “overrun”. Таймер является объектом микроядра и, будучи запущенным, генерирует уведомления независимо от состояния того “целевого” потока, который должен эти уведомления принимать. В очереди может находиться только один уведомляющий сигнал от таймера, все последующие теряются. Такая ситуа-

ция возникает, когда функционирование “целевого” потока препятствует своевременной доставке или принятию (*acceptance*) сигнала до следующего срабатывания таймера. Отличное от нуля значение overrun свидетельствует о неадекватной работе программы как приложения реального времени .

4. **Удалить таймер с освобождением всех занятых им ресусов**, когда необходимость в нём отпала. Синтаксис:
`int timer_delete( timer_t timerid );`

Примеры программ, содержащие интервальный таймер с уведомлением импульсом, использующий POSIX API, помещёны в справке по функции `timer_create()` и в книге [13]. Интервальный таймер с уведомлением сигналом, использующий собственый QNX/Neutrino API, приведен в п.3.3.

1.7.3.1. Задание времени первого срабатывания абсолютного таймера

1. Установить правильное системное время (см. п.1.3). Аппаратные часы PB должны быть установлены по UTC, местное время операционная система рассчитывает в соответствии со значением переменной окружения TZ ;

2. Задать в программе местное время срабатывания таймера в нужном часовом поясе, записав его в структуру *tm* . Члены этой структуры содержат годы, месяцы, часы, минуты и секунды. С помощью функции `time_t mktime(&tm)` получить календарное время (UTC) срабатывания таймера;

3. Пересчитать возвращённое `mktime()` время в требуемые для установки таймера секунды и/или наносекунды и записать в соответствующие структуры.

Использование абсолютного таймера иллюстрируется программами, приведенными в п. 3.2 и п. 3.4.

1.7.3.2. Получение информации об интервальных таймерах

Данные о таймерах, созданных в системных и пользовательских процессах, так же, как и вся остальная информация о процессах и потоках, хранится в виртуальной файловой системе `/proc` менеджера процессов. Доступ к этой информации иллюстрирует программа п. 3.5 [43]. Интересно, что одинаковые по своему действию таймеры,

заданные с помощью различных функций API, менеджером процессов воспринимаются по-разному. В простейшей тестовой программе (см. п.3.5) используются одновременно две информирующие об истечении периодических интервалов времени функции категории Standard Unix: интервальный таймер на основе `setitimer()` и периодический “будильник ” `ualarm()` (см. табл. 1.4). Анализ вывода программы получения информации о таймерах (см. п. 3.5) показывает, что файловая система менеджера процессов содержит только данные о таймере `setitimer()` и ничего об `ualarm()`.

1.7.4. Таймауты

Под таймаутом понимается заданный отрезок времени или момент абсолютного времени, до окончания которого разрешено ожидать определённое событие. Таймауты могут быть реализованы с помощью таймеров общего назначения. Более удобно использовать специализированные средства API.

Микроядро QNX/Neutrino поддерживает функции с таймаутами нескольких видов. Первая группа функций держит вызывающий поток в заблокированном состоянии, ожидая заданного события, затем выполняет определённые связанные с этим событием действия. Если событие не наступило до истечения таймаута, функция разблокирует ожидающий поток и возвращает ошибку ETIMEDOUT. К этой группе относятся, например, функции с абсолютным (привязанным к календарному времени) таймаутом (подробности см. в справочной системе QNX/Neutrino)

```
mq_timedreceive(),
mq_timedsend(),
pthread_cond_timedwait(),
pthread_mutex_timedlock(),
pthread_rwlock_timedrdlock(),
pthread_rwlock_timedwrlock().
```

и относительным (отсчитываемым от момента обращения к функции)

```
sem_timedwait(),
sigtimedwait(),
_sleepon_wait().
```

Более развитые действия по таймаутам, включая режим опроса готовности устройств, содержат функции `readcond()` и `select()`.

Перечисленные группы функций обрабатывают таймауты по отдельным возможным в системе состояниям блокировки. QNX содержит также общий универсальный механизм разблокирования потоков по таймауту для любых блокированных в ядре состояний. Этот механизм реализуется третьей группой функций, которые предназначены исключительно для отслеживания времени пребывания потока в заблокированом в ожидании события состоянии. Поток попадает в блокированное состояние в результате обращения к любому из блокирующих системных вызовов и может быть выведен из этого состояния по таймауту, задаваемому в самих функциях этой группы. Ввиду универсальных возможностей использования этих функций рассмотрим их более подробно. К таким функциям относятся POSIX функция `timer_timeout()` и системный вызов `TimerTimeout()`. Различие между ними заключается в единицах времени для задания таймаута – в первом случае это члены структуры struct timespec в целых секундах и наносекундах последней неполной секунды, во втором – наносекунды.

Синтаксис:

```
extern int timer_timeout(
              clockid_t id,
              int flags,
              const struct sigevent* notify,
              const struct timespec* ntime,
              struct timespec* otime );
```

Обращаться к ф-ции *`timer_timeout()`* следует непосредственно перед вызовом той функции, на которую таймаут устанавливается. Задаваемые аргументы:

– ВременнАя база *`id`* используемых системных часов (см. п. 1.1);

– *`flags`* – битовая маска, определяющая, на какие блокированные состояния таймаут распространяется. Задаётся одной или несколькими объединёнными операцией ИЛИ константами. Символические имена констант и соответствующие состояния приведены в табл. 1.6.

Дополнительно к перечисленным значение TIMER_ABSTIME аргумента flags означает, что таймаут задаётся как абсолютный.

– *notify* указывает на структуру *sigevent*, содержащую генерируемое при срабатывании таймера событие. Из всех предусмотренных вариантов событий только SIGEV_UNBLOCK гарантирует разблокирование потока.

– *ntime* задаёт значение таймаута аналогично тому, как это сделано для интервального таймера. Как и для других видов таймеров, событие об истечении таймаута генерируется в момент системного тика. Микроядро начинает отсчитывать таймаут в момент вызова `timer_timeout()`, а не в момент входа в блокированное состояние. Вследствие этого возможна ситуация, когда поток вытесняется после вызова `timer_timeout()` до обращения к заданному блокирующему системному вызову. Если такая неприятная ситуация не возникает, фактическое значение таймаута есть

$$\lceil (ntime.tv\_sec * 1000000000 + ntime.tv\_nsec) / clock\_period \rceil + jitter / clock\_period$$

системных тактов. В противном случае фактическая длительность состояние заданной блокировки может быть совсем малой, в предельном случае нулевой.

Значение NULL аргумента *ntime* означает отсутствие блокировки. В этом случае попытка потока войти в заблокированное состояние немедленно заканчивается возвратом с ошибкой ETIMEDOUT из соответствующего системного вызова. Это можно использовать для опроса потенциально блокирующих системных вызовов, например, `MsgReceivev()`, с немедленным возвратом по таймауту.

– *otime*. Если значение этого аргумента не NULL, он содержит неиспользованное время “сна” микроядра, находившегося в состоянии STATE_NANOSLEEP и “разбуженного” сигналом.

Функция `timer_timeout()` работает как однократный таймер: для каждого потока в любой момент времени может отслеживаться только один таймаут. Задание нового таймаута до того, как:

- один из заданных блокирующих системных вызовов (см. табл.1.6) завершился без блокирования,

Таблица 1.6

Условные обозначения констант блокированных состояний

Константа	**Таймаут на состояние**	**Блокирующий системный вызов**	**Что ожидает поток?**
_NTO_TIMEOUT_CONDVAR	STATE_CONDVAR	*SyncCondvarWait()* *SyncMutexLock()*	Разблокирования условной переменной по SyncCondvarSignal() или освобождения мутекса по SyncMutexUnlock()
_NTO_TIMEOUT_MUTEX	STATE_MUTEX		
_NTO_TIMEOUT_JOIN	STATE_JOIN	*ThreadJoin()*	Завершения присоединённого (joined) потока
_NTO_TIMEOUT_INTR	STATE_INTR	*InterruptWait()*	Аппаратного прерывания
_NTO_TIMEOUT_RECEIVE	STATE_RECEIVE	*MsgReceivev()*	Сообщения или импульса
_NTO_TIMEOUT_SEND	STATE_SEND	*MsgSendv()*	Приёма сообщения или импульса адресатом или ответа от адресата
_NTO_TIMEOUT_REPLY	STATE_REPLY		
_NTO_TIMEOUT_SEM	STATE_SEM	*SyncSemWait()*	Возможности инкременировать семафор
_NTO_TIMEOUT_SIGSUSPEND	STATE_SIGSUSPEND	*SignalSuspend()*	Получения сигнала
_NTO_TIMEOUT_SIGWAITINFO	STATE_SIGWAITINFO	*SignalWaitinfo()*	Получения сигнала
_NTO_TIMEOUT_NANOSLEEP	STATE_NANOSLEEP	*TimerTimeout()*	Истечения таймаута или разблокирующего сигнала

Примечание. Значение _NTO_TIMEOUT_NANOSLEEP аргумента *flags* блокирует вызывающий TimerTimeout() поток до истечения таймаута или получения сигнала. Это можно использовать для задания интервалов “отдыха” ядра (kernel sleep).

- или был заблокирован и разблокировался до истечения таймаута,
- или истёк таймаут

приводит к замене действующего таймаута новым для данного потока. Пример программы, использующей таймауты ядра, приведен в [13] и в п. 3.6.

Одним из средств реализации таймаутов являются сторожевые (“watchdog”, "keep alive","deadman") таймеры. Эти таймеры срабатывают, если критическое по времени действие не выполнено вовремя. На отдельных материнских платах управляющих контроллеров они реализованы аппаратно. Существуют также программные варианты реализации сторожевых таймеров с помощью стандартных средств POSIX. Один из вариантов рассмотрен в [10, гл.12]. Продвинутым вариантом программного “сторожа” ("smart watchdog"), предназначенного для оперативного восстановления функционирования системы в QNX/Neutrino является менеджер высокой готовности HAM (High Availability Manager).

1.8. ИЗМЕРЕНИЕ ВРЕМЕННЫХ ИНТЕРВАЛОВ

API QNX/Neutrino включает богатый набор функций для измерения временнЫх интервалов, которые могут использоваться как для замера “полного” физического времени между определёнными событиями, так и для оценки “чистого” времени, в течение которого приложение занимало процессор для выполнением своего кода и необходимых системных вызовов. Сводка функций приведена в табл.1.7. Кроме использования функций в табл. 1.7, интервалы могут быть определены:

1) подсчётом количества срабатываний интервального таймера заданной разрешающей способности;

2) извлечением информации о времени использования процессора процессом в целом и отдельными его потоками из системной области;

3) с помощью системной программы-профайлера (более подробно см. в книгах [6,7]);

4) с помощью утилиты `time`, определяющей (“грубо”) полное время выполнения команды, в том числе пользовательской программы.

В качестве примера использования функций из табл.1.7 рассмотрим программу time_interval_measure_3.c (п.3.6). В ней замеряется разными способами интервал времени, в течение которого поток заблокирован в ожидании мутекса, захваченного другим потоком. Опрос состояния потока проводится с помощью функции *`timer_timeout()`* с нулевым значением таймаута. Для уменьшения влияния других потоков на измеряемые интервалы запускать программу лучше с высоким приоритетом 63. Командная строка запуска

```
$ time on -p63 ./time_interval_measure_3.out.
```

Результаты работы программы следующие:

```
Mutex locked. Elapsed time measured with clock-function was 2.000000 seconds
Mutex locked. Elapsed time measured with difftime-function: 2.000000 seconds
Mutex locked. Elapsed time measured with ClockCycles-function  is 2.500713
Mutex locked. Elapsed time measured with clock_gettime-function: 2.502617
Mutex locked. Elapsed time measured with ClockTime-function: 2.502617
Mutex locked. times-function results: System time = 0.0000 s user time=2.50000 s
  10.03s real     2.50s user     0.00s system
```

Последняя строка выдаётся утилитой time. Обратите внимание на “грубое” округление интервалов до целых секунд функциями `clock()` и `time()+difftime()`. Некоторые запуски программы выдают значение, замеренное с помощью `difftime()`, равное 3 с. Предпочтительнее пользоваться функциями стандартов POSIX и собственными системными вызовами QNX/Neutrino.

Наличие нескольких функций API для измерения интервалов позволяет провести их взаимопроверку и выбрать результаты, подтверждённые несколькими способами замера. В [41] приведен пример, когда замеры интервалов с помощью разных функций могут дать существенно отличающиеся результаты.

Занятое процессами и потоками время процессора может быть получено из сответствующих информационных структур. Способ доступа к этим структурам иллюстрирует программа пп. 3.4.1. Достататочно подробная информация содержится в приложении “B” книги [43].

Таблица 1.7

Сводка функций, применимых для измерения временнЫх интервалов

Имя функции	Единицы измерения	Что замеряется	Стандарт
difftime()	с	Разница между двумя моментами календарного времени, определяемого функцией time()	ANSI
ClockTime()	нс	Системное время с начала эпохи UNIX	QNX 6
clock_gettime()	Полные секунды и наносекунды последней неполной секунды	Системное время с начала эпохи UNIX	POSIX 1003.1 (RT Extensions)
ClockCycles() Более поробно см. п. 3.5.1	Перевод в секунды-делением на *SYSPAGE_ENTRY(qtime)->cycles_per_sec*	Количество циклов работы процессора с начала очередной серии отсчётов	QNX 6. Макрос описан в /usr/include/x86/neutrino.h
clock()	Для перевода в секунды делить на CLOCKS_PER_SEC	Время с начала запуска процесса	ANSI
times()	Для перевода в секунды делить на CLK_TCK	Процессорное время родительского и дочерних потоков	POSIX 1003.1
time (утилита командной строки)	мс	Общее время, затраченное на выполнение программы, и время использования процессора для системных нужд	UNIX

Если программа замеряет время выполнения некоторого участка кода, повторяющегося циклически, следует иметь в виду, что первый шаг цикла выполняется почти всегда дольше, чем остальные. Это связано с особенностями архитектуры современных микропроцессоров. В заметке [44] приводятся следующие объяснения:

- на первом шаге цикла повторяющийся код загружается из оперативной памяти в кэш и только затем выполняется. Затраты времени на загрузку кэша отсутствуют на последующих шагах цикла, если объём кода целиком помещается в кэше;
- то же самое относится к обрабатываемым в цикле данным. Выборка данных из ОЗУ значительно более длительная операция, чем выполнение команд по их обработке;
- инструкции jump при первом выполнении цикла отсутствуют в таблице предсказания переходов, и поэтому едва ли будут предсказаны правильно;
- для многих процессоров декодирование длины команды является узким местом. Процессор Pentium решает эту проблему, запоминая длину любой команды, которая осталась в кэше со времени последнего её выполнения. Как следствие этого, набор команд не будет распариваться на Pentium при первом их выполнении, за исключением случая, когда первая из команд имеет длину 1 байт. Процессоры Pentium MMX, Pentium Pro, Pentium 2 и Pentium 3 не имеют дополнительных затрат времени на первое декодирование. На Pentium 4 команды при первом своём выполнении идут прямо с декодера на устройство выполнения. При последующих выполнениях они выбираются из кэша микрокоманд с частотой 3 микрооперации за такт.

1.9. ИСПОЛЬЗОВАНИЕ ДОПОЛНИТЕЛЬНЫХ ИСТОЧНИКОВ ПЕРИОДИЧЕСКИХ АППАРАТНЫХ ПРЕРЫВАНИЙ В КАЧЕСТВЕ ТАЙМЕРОВ

Любое устройство, генерирующее периодические аппаратные прерывания, может быть использовано в качестве источника периодических импульсов, которые могут быть основой для программных таймеров. Пример использования аппаратных часов реального времени (RTC) приведён в [45]. Микросхема RTC способна обеспечить следующие периоды аппаратных прерываний на линии IRQ8 (частоты RTC составляют ряд чисел 2^n Гц):

- 3600, 60, 1 секунда,
- 500,250,125,62.5,31.25,15.625,7.8125, 3.90625,1.953125 миллисекунд,
- 976.5625,488.281,244.141,122.07 микросекунд.

Дополнительно можно задать время прерывание в часах.минутах.секундах текущих суток.

Рассмотрим в качестве второго примера как для этих целей использовать микросхему UART. Программа приведена в приложении п. 3.7. На компьютере должна быть установлена заглушка, соединяющая выход UART с его входом (см. приложение 1 части 2 настоящего пособия).

РЕЗЮМЕ

Операционная система QNX/Neutrino имеет богатый API для работы со временем, включающий прежде всего функции-таймеры расширений реального времени стандартов POSIX и спецификации UNIX. Наличие функций стандарта ANSI и других обеспечивает возможность использовать фрагменты кода программ, разработанных для других операционных систем, в том числе ОС общего назначения. Особую ценность для программ реального времени имеют собственные системные вызовы микроядра QNX/Neutrino, обогащающие и расширяющие возможности, заложенные в стандартах POSIX.

Независимо от разделения функций API по группам в пределах разных стандартов в основе их всех лежит отсчёт времени аппаратными таймерами (для платформы x86 это таймер по типу Intel 8254). Важное свойство этого таймера состоит в том, что период отсчёта им времени не выражается через целое число микросекунд. Не сказываясь заметным образом на отсчёте системного времени (микроядро знает, на сколько реальных наносекунд оно увеличивает структуру системного времени при каждом тике системных часов), указанное свойство приводит к “неприятным” особенностям реализации ядром системных однократных и периодических таймеров. Периодический таймер может поддерживать произвольный заданный период отсчёта времени только “в среднем”. Однократный таймер всегда отработает интервал времени больше заданного, причём здесь микроядром дополнительно учитывается требование стандарта POSIX “не раньше заданного срока”.

Для любого таймера на поведение потока, ожидающего срабатывания таймера, может существенно влиять программное окружение. Постановка микроядром на выполнение потоков-конкурентов с более высоким приоритетом и/или более требовательной к ресурсам дисциплиной планирования (FIFO вместо Round-Robin) приведёт к запоздалой реакции потока на таймер. Учёт этих обстоятельств, включая проектирование правильной архитектуры для программных модулей, отвечающих за управление критичными по времени задачами, позволяет уменьшить “неточности” таймеров в реальных программах.

2. ПОГРЕШНОСТИ ЧАСОВ, ТАЙМЕРОВ И СТАТИСТИЧЕСКАЯ ОБРАБОТКА РЕЗУЛЬТАТОВ ИЗМЕРЕНИЙ ВРЕМЕННЫХ ИНТЕРВАЛОВ

Соотношение непрерывного физического и отсчитываемого компьютером дискретного времени можно графически представить следующим образом (рис. 2.1).

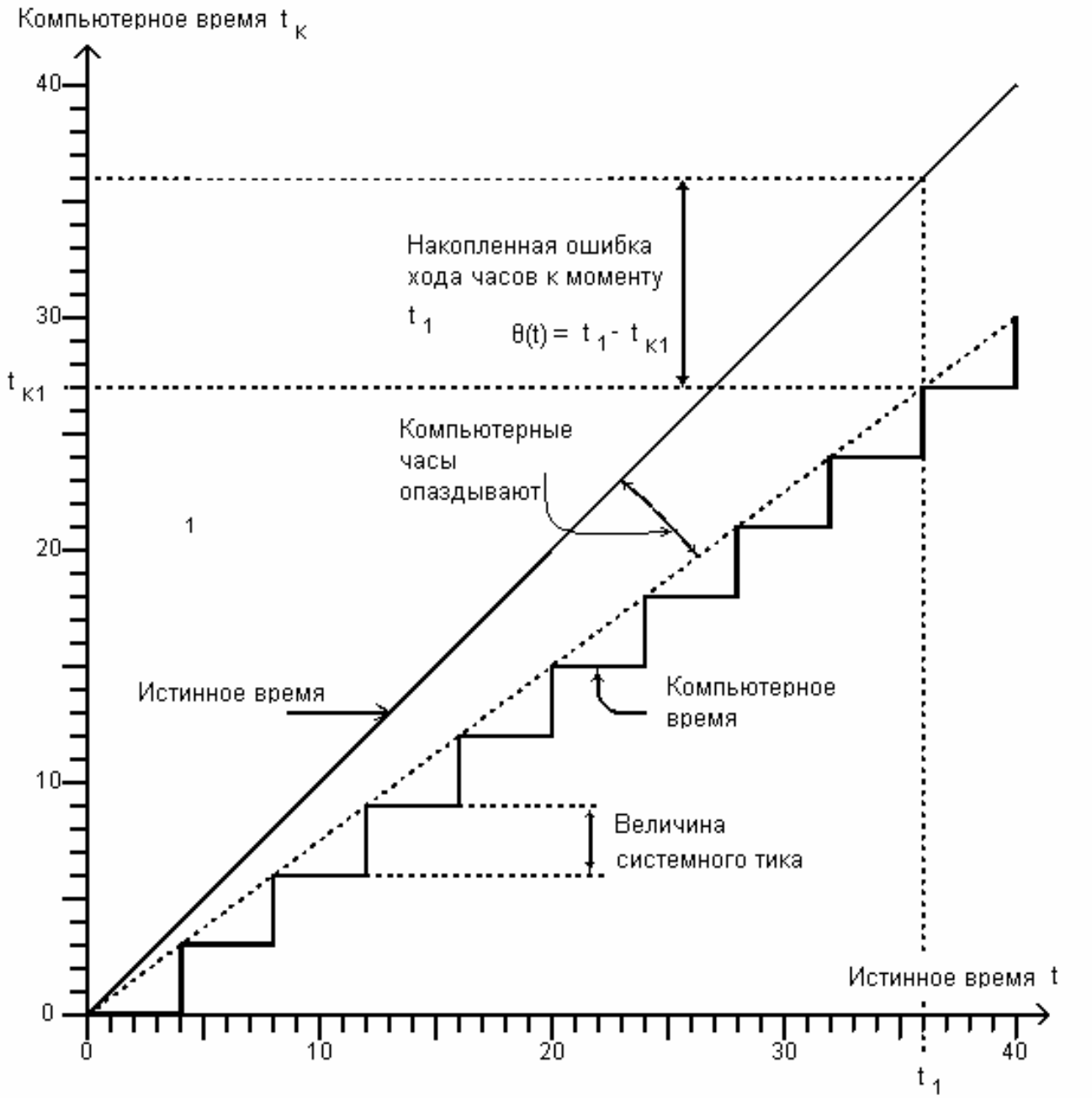


Рис. 2.1. Соотношение непрерывного физического и дискретного компьютерного времён

Операционная система считает, что каждый системный тик имеет величину 3 единицы. Реальная величина периода инкременирования значения системной переменной, хранящей системное время, несколько больше 3 единиц за счёт неточного определения системой частоты аппаратного таймера и задержки обработки прерывания таймера. Это приводит к рассогласованию показаний компьютерных часов по отношению к физическому времени и к накоплению растущей ошибки системного времени $\theta(t)$. Исследования UNIX-платформ показали [46], что дрейф системного компьютерного времени по отношению к истинному на протяжении интервала времени 1000 с. описывается зависимостью

$$\theta(t) = \theta(0) + \gamma * t + \eta(t) . \qquad (2.1)$$

В формуле (2.1) $\theta(0)$ – рассогласование в момент запуска системных часов (определяется точностью хода часов реального времени). На рис. 2.1 $\theta(0)=0$. Член $\gamma * t$ характеризует систематическую ошибку частоты аппаратного таймера i8254. Величина коэффициента γ составляет приблизительно $50*10^{-6}$. Третье слагаемое $\eta(t)$ представляет собой аддитивный стационарный случайный “шум” (по терминологии, принятой в радиоэлектронике) с нулевым средним и среднеквадратическим отклонением ~5 мкс, искажающий системное время за счёт других причин. К ним, в частности, относятся задержка обработки операционной системой прерываний аппаратного таймера и температурные флуктуации кварцевого осциллятора, которые находятся в пределах нескольких миллионных от номинальной величины. Как видно из формулы (2.1.), отклонение системного времени от истинного случайно. По прошествии 1000 с. начинает проявляться нестабильность частоты аппаратного таймера и приведенная модель становится непригодной.

Результаты нескольких замеров заданного времеНного интервала компьютерного времени также всегда имеют разброс, представляя случайную величину. Причиной этого является “шум”, накладывающийся на периодический временной сигнал работой аппаратуры компьютера и программного обеспечения, главным образом системного. К источникам шума можно отнести:

- неточность отработки операционной системой величины заданного интервала за счёт:
  - дискретности системного времени,
  - соблюдения требований стандартов POSIX (см. п. 1.6),

- конкурентного исполнения нескольких потоков, диспетчируемых по алгоритмам вытесняющей многозадачности на основе приоритетов,
- флуктуаций периода отсчётов аппаратных источников периодических импульсов,
- задержки в обработке прерываний аппаратного таймера при большой нагрузке на контроллер прерываний или полное запрещение прерываний пользовательскими программами;

- малая разрешающая способность и/или принципиально неустранимая “плавающая” неточность используемых программных средств измерения интервалов. Последнее относится также к использованию машинно-специфичного регистра TSC, лежащего в основе системного макроса `ClockCycles()`.

При отсутствии внешних по отношению к аппаратной части компьютера средств измерений дополнительным источником случайности является совместное параллельное использование программами одних и тех же ресурсов компьютера для задания и измерения времени, что потенциально искажает как задаваемые, так и измеряемые значения. Здесь уместна аналогия с принципом неопределённости квантовой физики: точности определения положения элементарной частицы и её энергии взаимосвязаны, если измерения проводятся одновременно - чем точнее определяется значение одного из параметров, тем менее точно значение другого. Вот как характеризует эту ситуацию участник форума [2] Evgeniy, касаясь методики и результатов замера накладных расходов ОС QNX на перепланирование потоков и задержку обработки прерываний (*scheduling* и *interrupt latency*): “Я не очень понимаю, чем эти данные вам сильно помогут - я имею в виду наихудшие значения без развернутого статистического анализа применительно к условиям вашей конкретной системы РВ - количество и размеры буферов ввода/вывода, ваши потоки данных и т.п. … 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

измерение временных характеристик системы, если их проводить корректно, является очень серьезным самостоятельным исследованием. И оценивать результаты отдельных измерений практически не имеет смысла”.

Аналогичные соображения справедливы и для точного задания временнЫх интервалов средствами операционной системы. Вот мнение участника форума [2] Цилюрика: «Вообще, в дополнение ed1k и в результате многочисленных экспериментов со временем в QNX, общее правило, наверное, должно формулироваться так: и не пытайтесь с помощью всех врЕменных функций OS задать **фиксированный** временной интервал в многозадачной (многопотоковой) OS - ничем хорошим это не закончится, а о точности здесь рассуждать просто смешно... Для всех этих служб времени руководствуйтесь положением POSIX 1003b: "временной интервал **ни в коем случае** не может быть меньше заданного, но может быть **сколь угодно** большим". Кстати, и в HELP при функциях "активного ожидания" (захватывающих циклы процессора) оговаривается, что они "предназначены для выдержки временных задержек, **требуемых аппаратурой**". Т.е. - для определения точных временнЫх интервалов, думаю, можете использовать только "внешние" по отношению к службе времени источники, вызывающие прерывания "со стороны"».

Разумеется, высказанное опытными практическими программистами мнение касается особых требований к функционированиию систем реального времени – или очень малого интервала таймера (конкретно 5 мкс для последнего цитированного фрагмента), или очень короткого замеряемого интервала (порядка десятков микросекунд), или сложной многозадачной структуры. В этих случаях могут понадобиться внешние аппаратные таймеры и высокоточные устройства замера коротких интервалов времени (см., например, методику тестирования ОС РВ, применяемую экспертами [24]). Для большого класса СРВ не требуется столь высокой точности и дискретности отсчёта времени. Чтобы разработчик мог правильно оценить соответствие требованиям к СРВ имеющихся в его распоряжении программно-аппаратных средств, он должен чётко представлять себе основные факторы, влияющие на стабильность задания и измерения временНых интервалов. Рассмотрим более подробно некоторые из этих факторов.

2.1. ДИСКРЕТНОСТЬ КОМПЬЮТЕРНОГО ВРЕМЕНИ

На рис.2.2 показаны временНые ошибки, возникающие за счёт дискретной природы системных компьютерных часов.

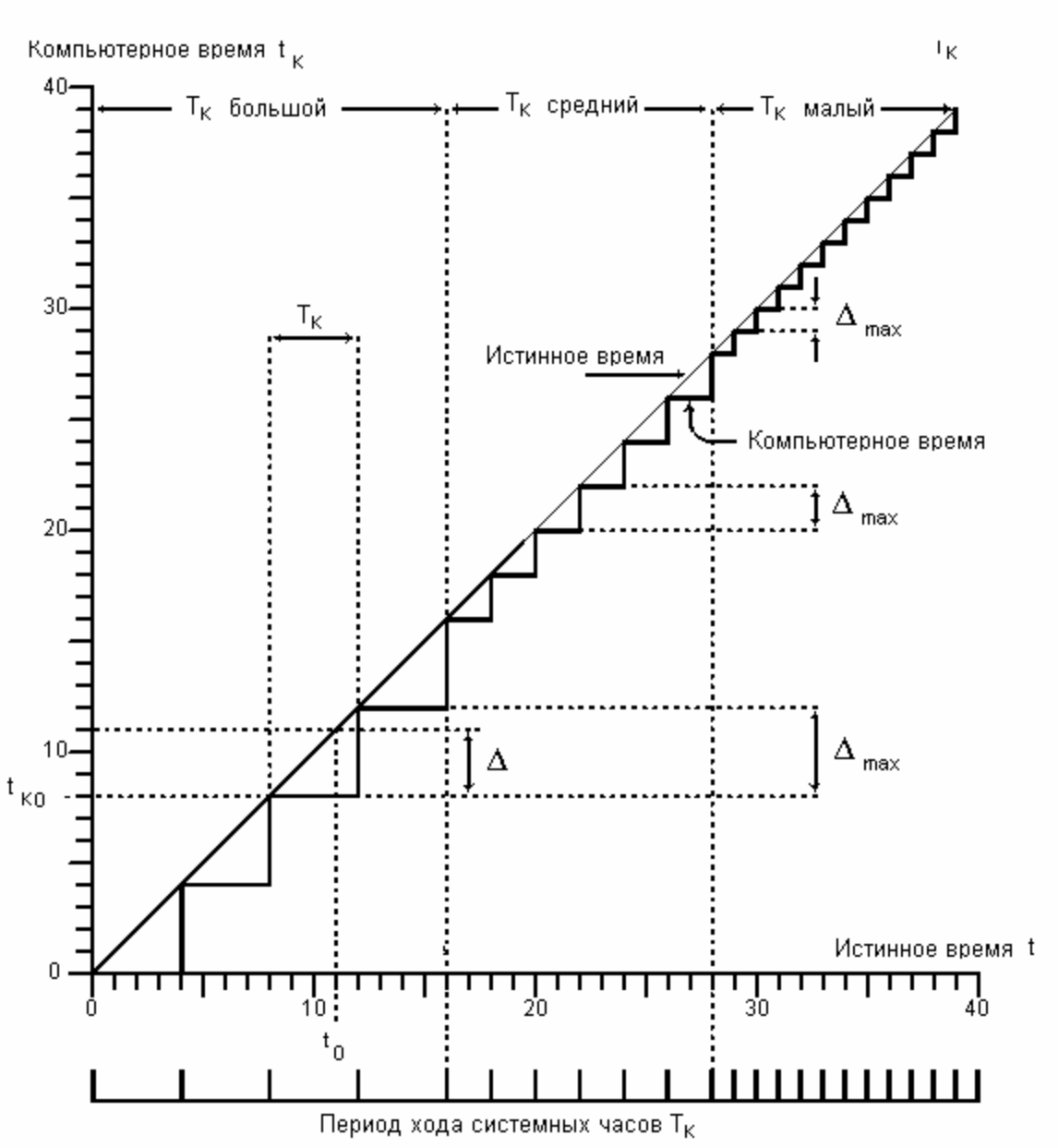


Рис. 2.2. Зависимость ошибки Δ компьютерного времни от периода хода системных часов

В момент времени t_o приложение запрашивает текущее время у операционной системы, которая возвращает значение $t_{ко}$. Величина наибольшей погрешности Δ равна периоду хода системных часов $T_к$.

Чем меньше период $T_к$, тем меньше наибольшее значение ошибки Δ_{max}.

2.2. АППАРАТНЫЕ ПРЕРЫВАНИЯ

Одной из функций микроядра является отсчёт системного времени на основе обработки прерываний от аппаратного таймера. Обработчик прерываний таймера вытесняет выполняющийся поток каждый раз, когда это прерывание наступает. При формировании системного времени отставание от реального может возникнуть, если в момент появления прерывания таймера операционная система занята обработкой прерываний от других устройств – сетевого адаптера, аудио карты и т.д. Хотя таймер генерирует прерывание IRQ0 наивысшего приоритета, понадобится некоторое время Δt, чтобы вытеснить обработчики более низкоприоритетных прерываний. Δt может быть значительным в тех случаях, когда в обработчике прерываний приложения вызвана функция запрета прерываний.

Аналогичным образом аппаратные прерывания могут исказить замеряемое время выполнения участка программы. Если длительность выполнения участка кода близка к величине системного тика и код выполняется циклически, наложение обработчика прерываний таймера может существенно исказить временное поведение пользовательской программы. Для более точного замера времени выполнения можно кратковременно запрещать обработку аппаратных прерываний, однако при этом вносится погрешность в отсчёт системного времени.

2.3. ПОГРЕШНОСТИ ПЕРИОДИЧЕСКИХ ИМПУЛЬСОВ АППАРАТУРЫ

Аппаратные таймеры используют задающие генераторы импульсов на основе кварцевых осцилляторов. Точность и стабильность поддержания частоты этими устройствами зависит от многих факторов внешней среды – напряжения питания, температуры, влажности, атмосферного давления и даже высоты над уровнем моря. Для нетермостабилизированных компьютеров, к которым относится большинство промышленных управляющих машин и обычных персоналок, определяющим фактором является температура. Так, часы реального

времени имеют относительную погрешность хода не менее $1*10^{-6}$, что в лучшем случае даёт ошибку хода 0.1 секунды за сутки. Типичной является погрешность хода 5-6 секунд в сутки [47].

Базовый аппаратный таймер i8254 способствует накоплению ошибки системного времени по отношению к эталонному за счёт джиттера периода прерываний и долговременного дрейфа этого периода от изменений температуры и напряжения питания (см. формулу (2.1)). Подробное описание детерминированных и случайных отклонений частоты кварцевого осциллятора от номинального значения дано в [48]. Интересно, что даже после корректировки несовпадения частоты осциллятора с эталоном и долговременного дрейфа частоты время, отсчитываемое осциллятором, “плавает” относительно эталонного в пределах ±150 мкс на протяжении часов наблюдений по причине случайного шума, который принципиально не устраним, но может быть скомпенсирован путём периодической синхронизации осциллятора с эталонными часами. Изучение статистических закономерностей этого шума позволило смоделировать его и на этой основе разработать эффективный алгоритм корректировки хода часов по показаниям эталонных. Разработанный метод и устройство корректировки названо “*smart clock*” и запатентовано (US Patent 5274545). Некоторые авторы [51] объясняют кратковременную нестабильность кварцевого генератора неэлектромагнитным информационным влиянием.

Специальные методы позволяют достичь высокой аппаратной стабильности кварцевых осцилляторов как на протяжении секунд, так и лет (см., например, [49]).

Программный метод компенсации температурного дрейфа частоты кварцевого осциллятора NTP-серверов предложен в [50]. Метод основан на измерении температуры вблизи кристалла, расчёте добавки и программной корректировке частоты. На аппаратной платформе Compaq AlphaPC 164 432 под управлением ОС Tru64 Unix 4.0B с модулем MICRO_TIME среднеквадратическое отклонение флуктуаций системного времени сервера относительно точного времени, получаемого от GPS-приёмника, уменьшено в 3.54 раза, полуразмах – в 4 раза.

2.4. АРХИТЕКТУРНЫЕ ОСОБЕННОСТИ МИКРОПРОЦЕССОРОВ

Архитектура современных микропроцессоров семейства x86 содержит элементы – конвейеры, кэши кода и данных нескольких уровней, блок предсказания переходов и ряд других - делающие выполнение инструкций не вполне предсказуемым по времени. Сами команды микропроцессора в зависимости от архитектуры процессора, предыстории, места размещения операндов и др. причин могут занимать различное количество процессорных тактов от начала выполнения до завершения. В контексте рассматриваемой темы это проявляется в том, что время (количество процессорных тактов) выполнения участков кода, а часто и порядок выполнения инструкций для одного и того же участка кода (для Pentium II, Pentium Pro и более молодых процессоров), а также кода функций – измерителей времени их выполнения может быть разным в зависимости от программного окружения и/или состава аппаратной части компьютера. В строгом смысле слова такое поведение нельзя рассматривать как погрешность, аналогичную той, которую вносит систематическое или случайное отклонение от номинала частоты кварцевого осциллятора аппаратного таймера. Однако в системах реального времени с заранее непредсказуемым порядком внешних событий указанные особенности программно-аппаратного поведения компьютера не могут быть заранее известны и выглядят как погрешности отработки/замера времени.

2.4.1. Макрос ClockCycles()

Рассмотрим более подробно этот вопрос применительно к инструкции RDTSC, лежащей в основе системного макроса `ClockCycles()`. Системный вызов `ClockCycles()` может использоваться как для замера интервала времени между некоторыми внешними событиями, так и для определения времени выполнения отдельных фрагментов программ.

Анализ использования TSC для профилирования программ выполнен в части I книги [52]. В тех случаях, когда замеряемый промежуток времени имеет порядок десяток-сотен процессорных тактов, важно учитывать продолжительность выполнения самой инструкции RDTSC, которая согласно [44] (см. также [53]) составляет (табл.2.1)

Таблица. 2.1

Время выполнения инструкции RDTSC в процессорных тактах

Процессор	**Потребное количество тактов для выполнения**
Pentium	6 в привилегированном или реальном режимах, 11 - в непривилегированном режиме
Pentium MMX	8 в привилегированном или реальном режимах, 13 - в непривилегированном режиме
Pentium Pro, Petntium 2, Pentium 3	31
Pentium 4	11

Следуя [54], можно достаточно точно оценить число тактов, занимаемых RDTSC, на конкретной машине. В программе п.3.8 подряд размещаются 2 вызова `ClockCycles()` и находится разность между возвращаемыми значениями.Фрагменты программы, относящиеся к теме данного пункта, выделены полужирным начертанием. Дополнительно в программе иллюстрируется способ получения информации о процессе и потоках, включая время использования процессора процессом и потоками (см. п. 1.8). Приведём данные для двух лабораторных машин со следующими характеристиками (табл. 2.2).

Таблица 2.2

Характеристики лабораторных компьютеров

Характеристика	**ПК №1**	**ПК №2**
Процессор	Celeron 635 МГц, кэш 1-го уровня 32 Кбайт	Celeron 668 МГц, кэш 1-го уровня 32 Кбайт
Материнская плата	Lucky Star 693A-686	Gygabyte 6VTXE
Ёмкость ОЗУ	2*64 = 128 Мбайт	128+64 = 192 Мбайт
Чипсет	VIA VT82C693A Apollo Pro 133 System Controller	Gygabyte VT82C694X Apollo Pro 133A System Controller
Частота шины FSB	67 МГц	67 МГц
Частота шины памяти	67 МГц	100 МГц

Несколько запусков программы п.3.8 дали следующие результаты (табл. 2.3).

Таблица 2.3

Результаты “калибровки” ClockCycles()

№ шага цикла	ПК №1						ПК №2 в любых условиях
	В псевдотерминале Photon с printf() / В “голом” терминале с printf()					Без printf() с записью в массив	
	Запуск программы						
	1	2	3	4	5		
0	102 / 102	102 / 102	102 / 102	102 / 102	102 / 102	33	33
1	121 / 443	33 / 443	33 / 583	121 / 443	33 / 187	33	33
2	33 / 33	458 / 33	177 / 33	33 / 33	33 / 33	33	33
3	121 / 33	122 / 33	122 / 33	33 / 435	122 / 33	33	33
4	33 / 33	33 / 33	33 / 33	122 / 33	33 / 33	33	33
5	33 / 33	33 / 33	33 / 33	33 / 33	33 / 33	33	33

Из табл. 2.3 можно заключить, что на обоих лабораторных машинах выполнение инструкции RDTSC занимает 33 такта, что хорошо согласуется с данными Intel в табл. 2.1. Вместе с тем небольшие различия аппаратной части сказываются на стабильности отработки `ClockCycles()`. Дополнительные инструкции в теле цикла, реализующие `printf()`, на ПК №1 иногда вызывают “задержку”. На более производительном железе ПК №2 такого никогда не наблюдается. Аналогичное “аппаратное непостоянство” описано в [52]. Отметим, что на многопроцессорных архитектурах счётчики циклов отдельных процессоров также не синхронизированы между собой. Разработчики QNX/Neutrino особо указывают на некорректность использования разности показаний двух TSC-счётчиков различных процессоров, когда эта величина используется в качестве оценки интервала времени

какого-либо действия распределённого между процессорами потока. Корректное время получается только когда замеряемый поток привязывается к одному конкретному процессору.

Проанализировав время выполнения инструкций на процессоре с точки зрения профилирования программ, автор [52] приходит к выводам:

1. *Не всякая система (имеется в виду аппаратура компьютера) пригодна для профилировки и оптимизации приложений*.
2. *Если последовательные замеры дают значительный временной разброс, просто смените систему*.

По мнению автора [52], “минимальный промежуток времени, которому еще можно верить при замере с помощью time-stamp counter, составляет, как показывает практика и личный опыт автора, по меньшей мере пятьдесят–сто тактов”.

Для “сложных” участков кода, работающих с большим количеством данных, погрешность замера может возникнуть из-за способа реализация макроса `ClockCycles()` в QNX/Neutrino. Заголовочный файл /usr/include/x86/neutrino.h содержит следующее определение

```
#elif defined(__GNUC__)
#define ClockCycles()   ({ register _Uint64t __cycles; __asm__
__volatile__( \
                "rdtsc" \
                 : "=A" (__cycles)); __cycles; })
```

Это определение и пример его использования в справке по теме `SYSPAGE_ENTRY()` библиотеки QNX/Neutrino не соответствует рекомендациям фирмы Intel, изложенным в документе [55]. Согласно этим рекомендациям, в процессорах Pentium Pro, Pentium II (а также более молодых) RDTSC — это *неупорядоченная* команда, которая может завершаться даже раньше предшествующих ей команд, если, например, предшествующая команда простаивает в ожидании операндов. Чтобы RDTSC начала выполняться после завершения всех предшествующих ей инструкций, перед ней нужно поместить одну из инструкций упорядоченного выполнения, в качестве которой Intel рекомендует CPUID. Из общего количества тактов, затраченных на выполнение участка кода, надо затем вычесть время, потраченное на CPUID (совместно с MOVE). Особенность CPUID состоит в том, что

первые два последовательных её “прогона” могут потребовать более длительного времени, чем все последующие. Поэтому Intel рекомендует три раза повторить вспомогательный фрагмент кода, содержащий RDTSC, с тем, чтобы вычесть из общего времени выполнения правильное значение накладных расходов. По-видимому, тот факт, что в QNX/Neutrino при обращении к счётчику TSC рекомендации Intel не реализованы, объясняет некоторые “странные” значения, возвращаемые макросом `ClockCycles()` при измерении времён выполнения отдельных участков кода программ. В любом случае получаемые с помощью `ClockCycles()` значения процессорных тактов “загрязнены” дополнительными затратами, удельный вес которых особенно велик при малых значениях порядка десятков и сотен. Для более “чистых” замеров интервалов времени с высокой разрешающей способностью представляется целесообразным использовать регистр-таймер системы ACPI, имеющийся в чипсете современных системных плат.

2.5. ДОСТИЖИМАЯ НА КОМПЬЮТЕРАХ ТОЧНОСТЬ ХОДА ЧАСОВ И ИЗМЕРЕНИЯ ВРЕМЕНИ

При рассмотрении этого вопроса не следует путать понятия точности (accuracy) хода компьютерных часов, характеризующей степень близости компьютерного и физического времён, и их разрешающей способности (precision, resolution). Часы с высоким разрешением могут идти сколь угодно неточно, точные часы в произвольные моменты времени могут иметь погрешность не выше разрешающей способности (см рис.2.2).

По результатам обследования около 20000 подключённых к сети Интернет компьютеров [47] было обнаружено, что усреднённая по ансамблю относительная ошибка частоты хода системных часов, определяемая разбросом тактовой частоты аппаратного таймера, составила $78*10^{-6}$. Для отдельных машин она составляла $500*10^{-6}$. Возникающее за счёт этого рассогласование системного времени с эталонным велико. Многие задачи, решаемые с помощью вычислительной техники, требуют значительно более точного соответствия компьютерного времени и UTC. Сюда, в частности, относятся и управляющие системы реального времени, особенно в тех случаях, когда для каких-то действий установлены абсолютные таймеры. Для ряда задач столь же важным является возможность очень точной привязки

внешних событий к шкале реального времени независимо от того, с какой точностью отсчитывается системное время. Рассмотрим несколько примеров того, как эти задачи решаются на практике.

2.5.1. Синхронизация системных часов

Ручная синхронизация на любом компьютере, в том числе на платформе QNX (см. п.1.3.1), может быть осуществлена по радио/телевизионным сигналам точного времени. Точность синхронизации порядка 1 с.

Для автоматической синхронизации используются следующие источники эталонного времени и способы передачи временного сигнала [47,56]:

- серверы точного времени, передающие значения времени компьютерам-клиентам по протоколу NTP или его упрощённой версии SNTP (*Network Time Protocol*, *NTP-серверы*). Серверы образуют иерархическую структуру из нескольких ярусов (стратумов). Самые точные серверы 1-го стратума отсчитывают время с погрешностью 10 мкс. За счёт задержек передачи сигнала в сетях точность хода серверов более низкого уровня хуже. Синхронизируемый компьютер связан с NTP-сервером сетью, в наилучшем случае точность установки системного времени может достичь 1 мс. Специальные программно-аппаратные решения позволяют компенсировать недетерменированное прохождение сигнала по коммутируемой сети Ethernet [57], позволяя достичь точности синхронизации порядка 1 мкс. Помимо сетевого интерфейса, передача точного времени с серверов на компьютеры-клиенты осуществляется также по телефонным линиям через аналоговые модемы. По этому принципу работает автоматизированная служба компьютерного времени (ACST) Национального института стандартов США. Наибольшая точность ±5 мс получается, когда калибруется линия связи между компьютером-клиентом и сервером путём прямой (от сервера) и обратной (от клиента) посылки сигнала [58];
- приёмники системы глобального позиционирования GPS (аналогичная Российская глобальная навигационная система - ГЛОНАС). Временная погрешность выходных сигналов этих приёмников порядка 1 мкс для доступных всем стандартных сервисов, а при условии калибровки антенны и исключении по-

мех выборочного доступа к GPS достигает ±100 нс [58]. Приёмник передаёт ежесекундные сигналы точного времени синхронизируемому компьютеру через последовательный порт или шину расширения. Точность синхронизации достигает 10 мкс. Обычно GPS-приёмники являются эталонными источниками времени для NTP-серверов 1-го стратума. В качестве примера такого устройства можно привести одноплатный компьютер фирмы Soekris [59], обеспечивающий стабильный период 1с ±120нс, что позволяет использовать их в качестве NTP серверов 1-го стратума.

Отметим, что для QNX/Neutrino 6 портирован NTP-демон (*ntpd*), позволяющий корректировать время системных часов по протоколу NTP с тайм-сервера. Более подробная информация имеется на форуме [2].

Отдельный вопрос, связанный с возможной корректировкой хода часов, – какой длины переменную-аккумулятор взять для хранения системного времени? Разработчики ОС РВ EYRX для платформ ПК [60] подошли к решению этого вопроса следующим образом. Дискретность внутреннего представления системного времени принята 1 нс. Для хранения с дискретностью 1 нс системного времени, прошедшего с начала эпохи UNIX, достаточно 64-разрядной переменной. В этом случае счётчика времени хватит на

$$\frac{2^{64}\,нс}{364*24*60*60*10^{9}\,\frac{нс}{год}} = 586 лет \approx 584 года$$

с учётом високосных лет. Счётчик заполнится в 1970 + 584 = 2554 году, что практически достаточно. Если компьютерные часы идут неправильно из-за того, что во время каждого системного тика счётчик системного времени увеличивается на неточную величину (см. рис. 2.2), эту величину можно скорректировать добавлением при каждом тике поправки Δ. Поправка Δ, очевидно, не превосходит величины тика. Наибольшее значение тика в ОС EYRX наблюдается при наименьшей допустимой частоте 18 Гц, откуда $\Delta_{\max} = \frac{10^9\,нс}{18} = 55555555.(5)нс$. Такие значения помещаются в 32 двоичных разряда (2^{32} = 4294967296), т.е. достаточно иметь 32-разрядную переменную, чтобы можно было корректировать системное время с дискретностью 1 нс. Однако наименьшее значение Δ в 1 нс не удовлетворило разработчиков ОС EYRX, поскольку

1 нс не удовлетворило разработчиков ОС EYRX, поскольку добавка 1 нс к каждому тику с частотой, например, 100 кГц (тик = 1/100000 с = 10000 нс) даёт

$$\frac{1нс}{тик} = \frac{1нс}{10000нс} = \frac{8.64с}{день} = \frac{52.596мин}{год}$$

, т.е. в лучшем случае на протяжении года корректировка позволит уточнить системные часы на 52.596 мин, что для систем реального времени очень грубо. Поэтому разрядность хранящей системное время переменной была увеличена до 96, а разрядность системного тика – до 64 бит. Для приложений видны только 64 разряда системного аккумулятора времени, обеспечивающие дискретность 1 нс, а дополнительные 32 разряда скрыты и используются только ядром ОС для размещения дробных долей корректирующих наносекунд. Теперь каждый разряд двоичного представления аккумулятора системного времени содержит $2^{-32} \approx 2.3283*10^{-19}$ с. При частоте тиков 100 кГц наименьшая одноразрядная корректировка обеспечивает

$$\frac{2^{-32}нс}{тик} = \frac{2^{-32}нс}{10000нс} = \frac{0.735мкс}{год}$$

, что более чем достаточно для практических нужд.

ОС QNX/Neutrino является коммерческой системой с закрытым кодом микроядра. В документации по Neutrino разработчики не сообщают подробно о том, как реализовано внутреннее представление системного времени. Некоторая информация доступна в описании структуры `qtime` системной страницы, которая вместе с другими данными содержит числовые параметры, определяющие внутреннее представление дискретности отсчёта системного времени с заданной точностью. Соотношение задающей частоты аппаратного таймера $f_{ат}$ и параметров расчёта значений системного времени *timer_rate* (“периода”) и *timer_scale* (“масштабного коэффициента”) задаётся формулой

$$f_{am} = \frac{1}{timer\_rate*10^{timer\_scale}}\ Гц\ . \qquad (2.2)$$

Для систем на аппаратной платформе x86 используются следующие значения: *timer_rate* = 838095345UL фемтосекунд, *timer_scale* = -15, что даёт по формуле (2.2) $f_{ат}$= 1193181,6 Гц – тактовую частоту таймера i8254. Эти числовые значения можно увидеть в выводе программы `startup-bios`, запущенной из уже загруженной ОС

QNX/Neutrino. Программа `startup-bios` аварийно заканчивает работу с дампом памяти (сообщение ОС “memory fault core dump”), просмотр дампа возможен обычным текстовым редактором.

2.5.2. Точная привязка внешних событий к реальному времени

Все эти решения опираются на использование дополнительных аппаратных источников периодических импульсов. Сюда относится прежде всего регистр TSC микропроцессора.

Авторы [46] разработали сверхточные часы на основе TSC. Эти часы служат для определения календарного времени наступления внешних событий и работают параллельно с основными системными. В [46] отмечаются следующие достоинства TSC: погрешность частоты порядка $1*10^{-7}$, стабильность частоты на уровне генерирующих электронных схем, очень малые накладные расходы на считывание показаний этого счётчика - <50 нс на компьютере с частотой микропроцессора 600 МГц. Ключевым моментом “сверхточных” компьютерных часов [46] является знание точного значения (с относительной погрешностью не выше $1*10^{-7}$) тактовой частоты микропроцессора и соответственно TSC-регистра. Эта частота определялась по специальной методике либо с помощью временных меток NTP, либо путём посылки пакетов по сети от компьютера с эталонными часами вне зависимости от загрузки сети. Использования GPS не требовалось. Было установлено, что частота остаётся стабильной на протяжении недель и даже месяцев. Используя счётчик TSC как высокостабильный и высоточный источник временных отсчётов с известной частотой, авторы разработали для ОС PB RT-Linux программу-монитор сетевого трафика “TSC-RT-Linux monitor”. Монитор имеет погрешность отсчёта абсолютного времени 1 мкс и относительную стабильность частоты порядка $1*10^{-7}$. Помимо использования TSC, точность монитора достигается путём устранения “программного шума”, присущего стандартным программным системным часам компьютера. В мониторе минимизированы все системные программные действия от момента наступления события до получения его временной метки.

Аналогичные результаты получили разработчики службы времени UTIME ОС KURT Linux [20] (см. п. 1.2). Точная калибровка с помощью XNTP в момент запуска ОС используемого в качестве источника меток системного времени счётчика TSC процессора снижа-

ет погрешность хода системных часов до 235 мс за 14 дней и 0.5 с в течение месяца.

2.6. СТАТИСТИЧЕСКАЯ ОБРАБОТКА РЕЗУЛЬТАТОВ ИЗМЕРЕНИЙ ПЕРИОДИЧЕСКИХ ВРЕМЕННЫХ ИНТЕРВАЛОВ

При программировании приложений реального времени разработчику нужно знать, насколько точно предоставляемые операционной системой таймеры, заложенные в проект, отрабатывают заданные временнЫе интервалы, а также насколько стабильно эти интервалы поддерживаются на протяжении определённого отрезка времени. В частности, это относится к периодическим таймерам. Как указано выше, все временНые интервалы имеют случайные отклонения от средних значений, а сами средние в ряде случаев отличаются от заданных и могут быть непостоянны . Это касается как таймеров, так и продолжительности разного рода системных действий. Иллюстрацией, например, является распределение частот времён переключения нитей в ОС QNX/Neutrino 6.2 (рис.2.3, по данным группы экспертов [61]).

Для оценивания характеристик случайных величин требуется применение методов математической статистики. Существуют статистические методы анализа временных рядов, в частности разработаны специальные статистические характеристики – дисперсия Аллана и т.п. – для оценки погрешности частоты периодических таймеров, основанные на обработке различий частот анализируемого и эталонного источника частоты на протяжении определённого промежутка времени [9,62], и соответствующее программное обеспечение – в специализированных пакетах [63] и многих универсальных статистических. Часть методов предусматривает анализ стабильности осцилляторов в частотной области [64]. Эти методы применяются при проектировании серверов точного времени. В частности, они позволяют определить интервал, по прошествии которого частота стабилизируется, и тип “шума”, искажающего частоту, на основе чего можно внести коррективы в архитектуру устройства. Обязательным условием такого статистического анализа является наличие частотного эталона, с которым исследуемое устройство сравнивается, а также длительные замеры в течение десятков часов. В отсутствии эталонного

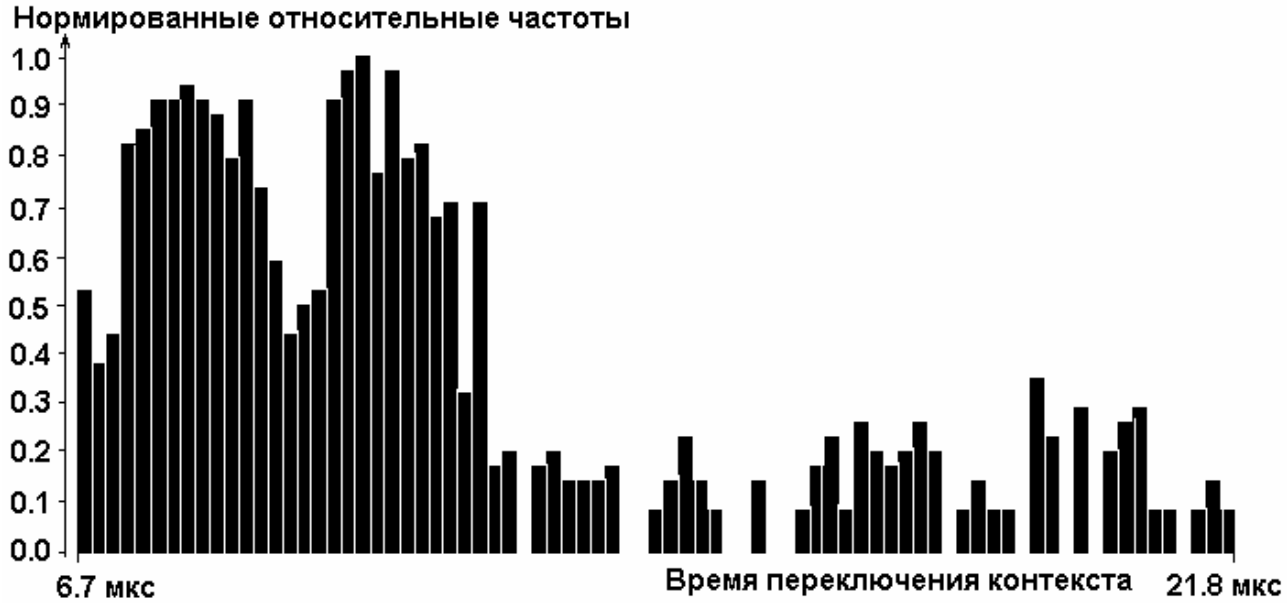


Рис. 2.3. Частоты случайного разброса времён переключения потоков в QNX Neutrino 6.2

временно́го устройства приходится ограничиваться более простым анализом, используя предоставляемое ОС QNX средство замера времени с высокой разрешающей способностью – `ClockCycles()`.О возможных “погрешностях” при использовании этого макроса говорилось выше, однако если сделать эксперимент “чистым”, замеренные с помощью `ClockCycles()` интервалы можно с большой долей уверенности считать их истинными значениями. Если замеры проводятся для сравнения между собой интервалов, соответствующих различным таймерам и/или условиям работы программ (приоритеты, окружение и т.д.), когда важны не абсолютные величины интервалов, а соотношения “больше-меньше” между ними, длительность выполнения `ClockCycles()` может не учитываться. Сама задача выбора адекватных статистических методов для оценки погрешностей системных таймеров, анализ их применимости с учётом лежащих в основе этих методов допущений нетривиальна и лежит за пределами настоящего пособия. Здесь ограничимся простейшей статистической обработкой, подходя к ней с позиций неискушённого в математической статистике программиста, который знает, что повторяющийся интервал срабатываний таймера – величина случайная, причём лежащие в основе случайности механизмы работы вычислительного устройства не известны, и устройство с этой точки зрения представляет собой “чёрный” или “серый” ящик. Разработчика интересует:

- насколько случайна величина отрабатываемого таймером интервала и в каких пределах она может изменяться;
- каково её среднее значение;
- насколько стабильно она ведёт себя по времени работы таймера;
- все ли предоставляемые операционной системой функции-таймеры и функции для измерения интервалов равнозначны по точности и/или стабильности или среди них есть более “хорошие”.

Ответы на эти вопросы можно получить, не прибегая к развитым методам анализа временных рядов, а используя классические методы математической статистики для одномерных случайных величин. Простейшая статистическая обработка включает:

- вычисление первых моментов - характеристик положения (математического ожидания) и изменчивости (среднеквадратического отклонения и коэффициента вариации) случайных величин;
- оценку степени случайности;
- сравнение моментов и распределений двух выборок одной/разных случайных величин. По результатам сравнения может быть сделан вывод о том, относятся ли эти величины к одной или разным генеральным совокупностям и можно ли порождающие случайные величины процессы считать одинаковыми или различающимися по своей природе и/или характеристикам. В частности, такое сравнение для нескольких выборок случайного интервала срабатываний периодического таймера позволяет сделать заключение о стабильности его работы.

Отметим, что при анализе работы периодических таймеров следует задавать повторяющийся интервал срабатывания кратным системному тику, в противном случае к случайным составляюшим примешивается неслучайная изменчивость интервала за счёт округления до целого числа тиков (см. п.1.6.1). Основными средствами математической статистики для решения перечисленных задач являются методы оценивания моментов, а также критерии согласия для проверки однородности функций распределения вероятности, средних и дисперсий двух случайных величин. Под однородностью в математической статистике понимается отсутствие значимых различий при сравнении тех или иных характеристик нескольких случайных величин.

Начальный статистический анализ однократных таймеров проведён в [41].

2.6.1. Краткое описание используемых статистических методов и критериев

Существующие статистические методы оценивания характеристик и сравнения случайных величин делятся на две большие группы. Первая группа относится к параметрическим методам – она предполагает известными законы распределения вероятности случайных величин, причём каждое распределение относится к определённому семейству функций, зависящих от небольшого числа параметров. С математической точки зрения это очень удобно, так как для полного описания распределения достаточно узнать или оценить одно, два или три числа. Типичным примером такого распределения является нормальное. “Распределения реальных данных практически никогда не входят в какое-либо конкретное параметрическое семейство. Реальные распределения всегда отличаются от тех, что включены в параметрические семейства. Отличия могут быть большие или маленькие, но они всегда есть. … К сожалению, параметрические семейства существуют лишь в головах авторов учебников по теории вероятностей и математической статистике. В реальной жизни их нет. Поэтому прикладная статистика использует в основном **непараметрические методы, в которых распределения результатов наблюдений могут иметь произвольный вид**”. Эта цитата из книги [65] объясняет, почему вопреки широко представленным в учебной и справочной литературе параметрическим методам статистики (критерии Стьюдента, Фишера и т.д.) анализ реальных статистических данных целесообразно проводить с использованием методов непараметрической статистики. В нашем случае наибольший интерес представляют непараметрические статистические методы проверки гипотезы однородности, т.е. совпадения функций распределения двух случайных величин, по выборкам ограниченного размера. Однозначно определенный способ проверки статистических гипотез называется статистическим критерием.

Общая схема применения статистического критерия такова [66].

1. Критерий предназначен для проверки некоторого предположения о свойствах случайной величины (или совокупности величин), называемого нулевой гипотезой $\mathbf{H_0}$. Если гипотеза $\mathbf{H_0}$ не верна, считается, что анализируемые данные удовлетворяют другому предположению – альтернативной гипотезе $\mathbf{H_1}$. При сравнении двух выборок гипотезы формулируются так:

- $\mathbf{H_0}$ - выборки взяты из одной генеральной совокупности, функции распределения обеих случайных величин совпадают. Совпадение распределений автоматически влечёт за собой совпадение средних и дисперсий.

  $\mathbf{H_1}$ - функции распределения обеих случайных величин не совпадают;

Или для менее строгих условий совпадения:

- $\mathbf{H_0}$ - средние значения (математические ожидания) двух случайных величин совпадают;

  $\mathbf{H_1}$ - средние значения не совпадают:
  - *двусторонняя* альтернативная гипотеза: несовпадение может заключаться в том, что среднее первой величины как больше – “одна сторона”, так и меньше – “вторая сторона”, чем у второй;
  - или *односторонняя* альтернативная гипотеза: среднее первой величины только больше или только меньше, чем у второй;
- $\mathbf{H_0}$ - дисперсии двух случайных величин совпадают;

  $\mathbf{H_1}$ - *двусторонняя* или *односторонняя* альтернативная гипотеза, аналогичная несовпадению средних.

2. На основе выборочных (полученных экспериментально) данных вычисляется **К -** значение *статистики* критерия [66]– случайной величины, для которой, если нуль-гипотеза справедлива , известно теоретическое распределение вероятности при определённых объёмах выборок, чаще всего, бесконечно больших. *Статистики* обычно называются по именам математиков, предложивших их использование.
3. Задают уровень значимости критерия α [66] – вероятность отвергнуть нулевую гипотезу в том случае, когда она верна (α может также выражаться в %). Отвержение правильной гипотезы называется **ошибкой первого рода**. Такую ошибку желательно совершать как можно реже, поэтому величина α берётся малой. Выбор α не связан непосредственно с математической статистикой, а определяется тем, **что** исследователь, проводящий статистическое тестирование, считает маловероятным событием, т.е. при проведении одного опыта, исход которого случаен, какие шансы (вероятность) произойти, на его взгляд, имеет маловероятное событие. Чаще всего в статистике используются значения α = 0.01, 0.05, 0.1, и,

как правило, статистические таблицы критических значений статистик построены для этих значений.

4. Определяют критическое значение критерия $\mathbf{K_{кр}}$ для принятого уровня значимости α по известной функции плотности вероятности критерия, соответствующей нулевой гипотезе. Значения $\mathbf{K_{кр}}$ затабулированы для наиболее употребимых уровней значимости и объёмов выборок. Вся совокупность возможных значений критерия в зависимости от принятого уровня значимости разбивается на две области: область принятия гипотезы и критическую область [66]. Области разграничивает $\mathbf{K_{кр}}$ - такое значение критерия, для которого вероятность событий

$\mathbf{K > K_{кр}}$ **(**правосторонний критерий),

$\mathbf{K < K_{кр}}$ **(**левосторонний критерий),

и (($\mathbf{K < K_{кр}}$**)** $\cap$**(**$\mathbf{K > K_{кр}}$**)) (**двусторонний критерий),

равна α. (рис.2.4, 2.5, 2.6). $P_k(x)$ обозначает функцию плотности распределения вероятности критерия при справедливости гипотезы $\mathbf{H_0}$). Для двустороннего критерия $\mathbf{K_{кр1}}$ и $\mathbf{K_{кр2}}$ выбираются так, чтобы вероятности попаданий в правую и левую критические области равнялись α/2.

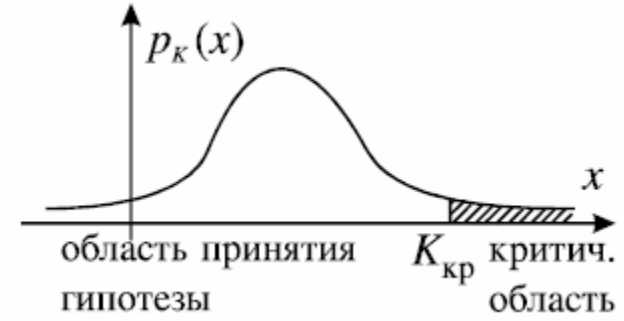


Рис.2.4. Правосторонняя критическая область

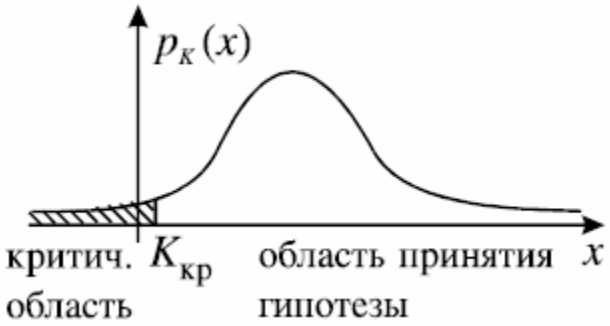


Рис.2.5. Левосторонняя критическая область

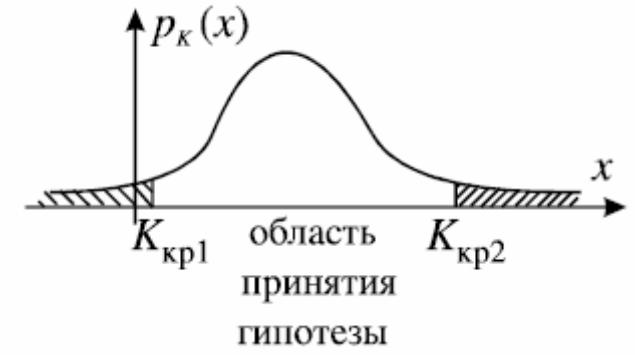


Рис.2.6. Двусторонняя критическая область

Имея значения К и Ккр, выносят заключение о принятии и отбрасывании нуль-гипотезы. Если гипотеза $\mathbf{H_0}$ справедлива, то вероятность того, что вычисленный по экспериментальным данным критерий **К** превзойдет значение $\mathbf{K_{кр}}$ очень мала – 0,05 , 0,01 или еще меньше, в зависимости от выбора α. Если **К** превзошло значение $\mathbf{K_{кр}}$, это означает, что выборочные данные не дают основания для принятия нулевой гипотезы $\mathbf{H_0}$ (например, если α=0,01, то можно сказать, что произошло событие, которое при справедливости гипотезы $\mathbf{H_0}$ встречается в среднем не чаще, чем в одной из ста выборок). В этом случае говорят, что **гипотеза $\mathbf{H_0}$ не согласуется с выборочными данными и должна быть отвергнута**. Ложность $\mathbf{H_0}$ доказана. Если **К** не превосходит $\mathbf{K_{кр}}$, то говорят, что **выборочные данные не противоречат гипотезе $\mathbf{H_0}$**, и нет оснований отвергать эту гипотезу. Очень важно правильно понимать результат теста. Если нуль-гипотеза не отвергается, это отнюдь не доказывает её истинность. Это свидетельствует лишь о том, что по имеющимся экспериментальным данным нет оснований считать гипотезу $\mathbf{H_0}$ ложной. Справочник [67] рекомендует такой подход к выбору нуль- и альтернативной гипотез: “Если мы хотим доказать, что на основании экспериментальных данных имеет место что-то новое, отличное от того, что принималось ранее, мы формулируем это новое как альтернативную гипотезу. Если мы хотим продолжать считать, что имеет место традиционное состояние вещей до тех пор, пока нет очевидных свидетельств против этого, мы формулируем это состояние как нуль-гипотезу”. Один из авторов [68] приводит такую образную трактовку результатов статистического теста: “Инженеры (и я в том числе) ненавидят этот вид статистической двусмысленности. Но факт остаётся фактом: любой статистический

тест строится чтобы доказать, что что-то неверно. Сухой тротуар с очевидностью доказывает, что дождя нет (альтернативная гипотеза), а причиной мокрого тротуара мог быть как дождь (нулевая гипотеза), так и проехавшая мимо поливальная машина. Т.е. мокрый тротуар не может доказать, что идёт дождь, тогда как сухой делает очевидным отсутствие дождя". С развитием вычислительной техники наряду с классическим, когда по заданному уровню α рассчитывается $\mathbf{K_{кр}}$, появился другой подход – вычисляется по выборке значение статистики **К** и затем значение доверительной вероятности (называемое p-уровень, p-value), соответствующее **К**. p-уровень - это вероятность получить значение тестовой статистики равным или большим чем то, которое рассчитано по экспериментальным данным. p-уровень представляет статистическую значимость результата, числовую меру уверенности в соответствия выборочных данных нуль-гипотезе. Если величина p-value > α , нуль-гипотеза не отвергается.

5. С уменьшением уровня значимости расширяется область принятия гипотезы $\mathbf{H_0}$ и увеличивается риск принятия проверяемой гипотезы, когда она неверна, т.е. когда предпочтение должно быть отдано конкурирующей гипотезе. Пусть при справедливости гипотезы $\mathbf{H_0}$ статистический критерий **К** имеет плотность распределения $\boldsymbol{p_0(x)}$, а при справедливости альтернативной гипотезы $\mathbf{H_1}$ – плотность распределения $\boldsymbol{p_1(x)}$. Графики этих функций приведены на рис. 2.7. При заданном уровне значимости находится критическое значение $\mathbf{K_{кр}}$ и правостороняя критическая область. Если значение **К**, определенное по выборочным данным, оказывается меньше, чем $\mathbf{K_{кр}}$, то гипотеза $\mathbf{H_0}$ принимается. Предположим, что справедлива на самом деле конкурирующая гипотеза $\mathbf{H_1}$.

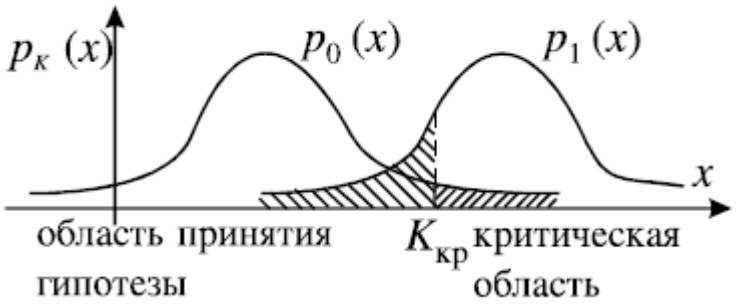


Рис. 2.7. Вероятность ошибки второго рода и мощность критерия

Тогда вероятность попадания критерия в область принятия гипотезы $\mathbf{H_0}$ есть некоторое число $\boldsymbol{\beta}$, равное площади фигуры, образованной графиком функции $p_1(x)$ и полубесконечной частью горизонтальной координатной оси, лежащей слева от точки **К**. Очевидно, что $\boldsymbol{\beta}$ – это вероятность того, что будет принята неверная гипотеза $\mathbf{H_0}$. Принятие неверной гипотезы называется ошибкой второго рода, $\boldsymbol{\beta}$ – вероятность ошибки второго рода. Число $\mathbf{1–\beta}$, равное вероятности того, что не совершается ошибка второго рода, называется мощностью критерия. На рис.2.7 мощность критерия равна площади фигуры, образованной графиком функции $p_1(x)$ и полубесконечной частью горизонтальной координатной оси, лежащей справа от точки **К**. Если имеется несколько критериев для проверки статистической гипотезы, следует предпочесть тот, который обладает большей мощностью. Для некоторых критериев мощность затабулирована, для других, более сложных, имеются сравнительные оценки мощности “в среднем”.

6. Среди всех возможных критериев следует пользоваться состоятельными. Применительно к задаче сравнения распределений случайных величин по их выборкам состоятельный критерий – это такой, что для любых отличных друг от друга функций распределения первой и второй величины (другими словами, при справедливости альтернативной гипотезы $\mathbf{H_1}$) вероятность отклонения гипотезы $\mathbf{H_0}$ должна стремиться к 1 ($\boldsymbol{\beta \to 0}$) при увеличении объемов выборок.

С точки зрения математической статистики при определении однородности двух выборок важно учитывать условия проведения эксперимента. Различают два случая – независимые выборки и связанные, или парные выборки. В каждом из случаев применяют свои критерии проверки однородности. Применительно к сравнению результатов замеров случайных интервалов срабатывания периодических таймеров независимые выборки получаются, например, когда одним способом замеряются интервалы, отрабатываемые таймером в разных прогонах одной и той же программы. Суть различий в том, что ни одно значение первой выборки не связано с каким-либо значением второй выборки в момент его получения. В случае связанных выборок значения из обеих выборок связаны попарно в момент их получения, например, когда двумя разными способами измеряется один и тот же отрабатываемый таймером интервал.

В математической статистике разработано большое количество различных критериев согласия, каждый из которых имеет своё назначение, условия и правила применения. Конкретные критерии, использованные для анализа работы таймеров, описаны ниже в соответствующих параграфах.

2.6.2. Периодические таймеры реального времени

В тестовой программе (см. п.3.9) заданное количество раз в цикле периодический таймер отрабатывает интервал 5*tick = 4999235 нс. Считываются и сохраняются в массиве показания счётчика TSC процессора на границах интервалов. После завершения работы таймера рассчитывается и записывается в файл массив интервалов между срабатываниями таймера в единицах “количество циклов процессора”. Для минимизации влияния других программ тестовая программа работает с наибольшим приоритетом (63 в QNX 6.2.*, 255 в QNX 6.3.*). Пересчёт полученных интервалов с единиц измерения “циклы процессора” в единицы “системный тик” осуществляется путём деления на

$$\frac{величина\_тика\_в\_нс * такт\_частота\_процессора\_Гц}{10^9\_наносекунд} = 668129\_\frac{циклов\_процессора}{тик}.$$

Напомним, что разрешающую способность системных часов в нс возвращает функция `clock_getres()`, тактовая частота процессора содержится в структуре `qtime` системной страницы, возвращается макросом SYSPAGE_ENTRY().

Записанный в файл массив интервалов можно проанализировать с двух точек зрения, рассматривая длительности интервалов либо как не упорядоченные по времени и не связанные друг с другом значения случайной величины, либо как реализацию случайного процесса – дискретного временного ряда. В первом случае, если прослежена работа на достаточно продолжительном промежутке времени, можно сделать выводы о точности (сравнив заданный и усреднённый фактический интервалы) и стабильности таймера, включая оценку допустимости для конкретной системы реального времени крайних снизу и сверху значений интервала. Во втором случае можно получить дополнительную информацию о стабильности работы таймера на коротких и длинных отрезках его работы и о некоторых внешне проявляющихся скрытых механизмах работы таймера как продукта взаимодействия микроядра и аппаратуры компьютера. Для анализа

удобно использовать специализированные пакеты программы статистической обработки. На наш взгляд, одним из наиболее пригодных для подобных задач пакетов является DataPlot, разработанный и распространяемый бесплатно отделом инженерной статистики Национального института стандартов и технологий (NIST) США [69] (для студентов-программистов будет интересно узнать, что вычислительные модули пакета написаны на языке FORTRAN и построены компилятором Intel 8.1, а графический интерфейс – на скриптовом языке Tcl/Tk). Справочные материалы по математической статистике в применении к инженерным задачам и пакету DataPlot подробно изложены в электронном справочнике [67]. Можно пользоваться также программой Statistica+ [70] с удобным Excel-подобным русскоязычным интерфейсом. Возможности Statistica+ (на момент написания пособия версия 2005 [3.5.0 RC9]) по сравнению с Dataplot весьма ограничены, однако безусловным достоинством является наличие критериев сравнения независимых и парных выборок, а также очень полезные справочные материалы, включающие статьи на русском языке ведущих специалистов-статистиков по правильному использованию статистических методов.

Рассмотрим вначале некоторые характеристики массива интервалов как временного ряда, используя технику графического анализа и представления данных программы DataPlot. С внешней точки зрения таймер представляет собой “серый ящик”, подробности алгоритма работы которого, в особенности касающиеся случайной составляющей времени срабатывания, скрыты от программиста, однако статистический анализ его работы даёт возможность выявить некоторые закономерности. Одна из реализаций ряда интервалов как функция порядкового номера интервала приведена на рис. 2.8 с разной разрешающей способностью по горизонтальной оси. После затухания на протяжении нескольких десятков срабатываний “переходных процессов”, связанных с заполнением кэша процессора и другими причинами, таймер выходит на установившийся режим работы. Внешний вид графика всей реализации напоминает биения с примесью случайного шума. Отчётливо видно, что периодически значения интервалов имеют выбросы, причём система вслед за выбросом в положительную сторону почти сразу корректирует работу таймера, уменьшив величину интервала до выброса в отрицательную сторону. На рис.2.9 приведен график зависимости величины следующего по порядку

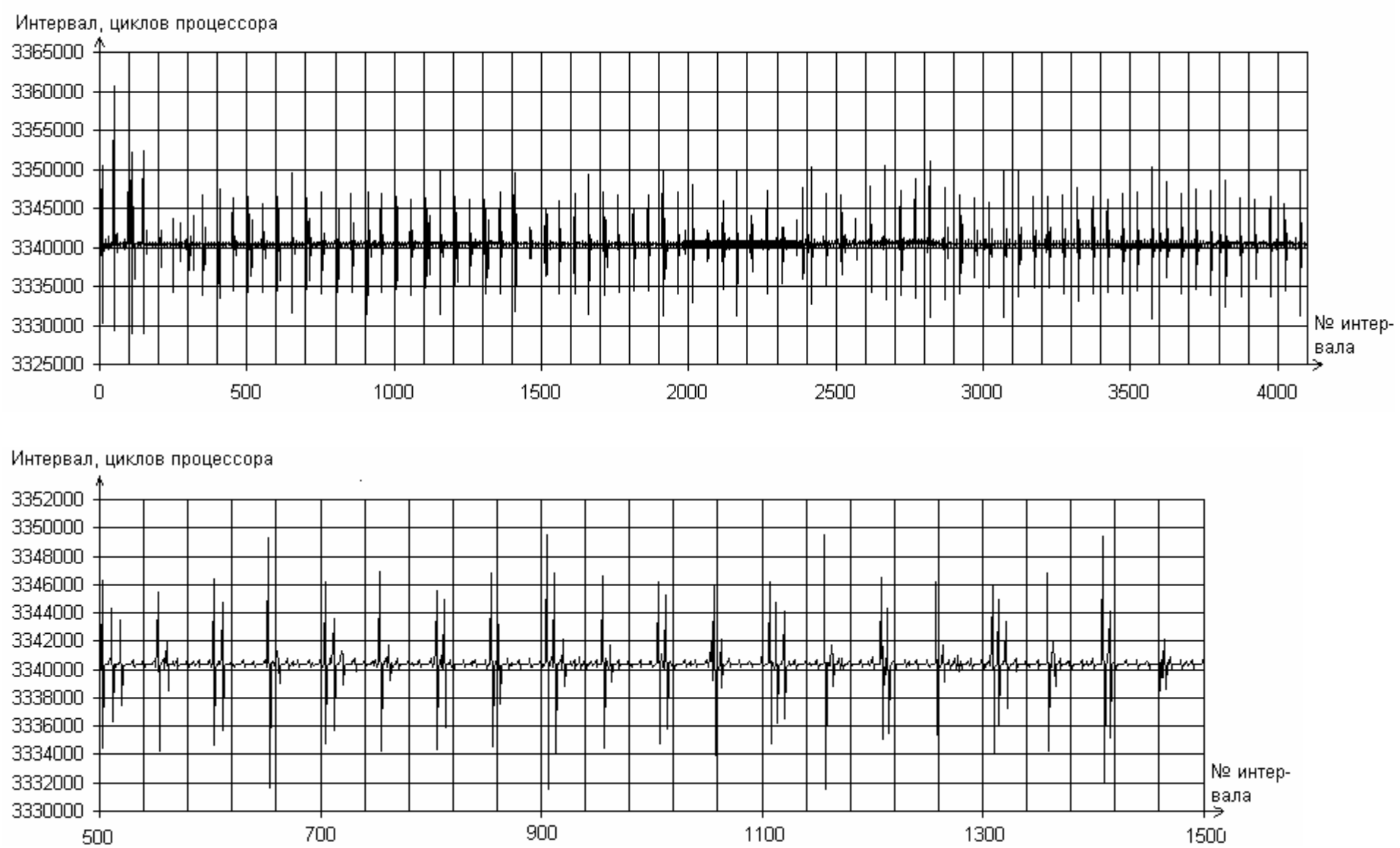


Рис. 2.8. Реализации случайного процесса последовательных интервалов периодического таймера (начало)

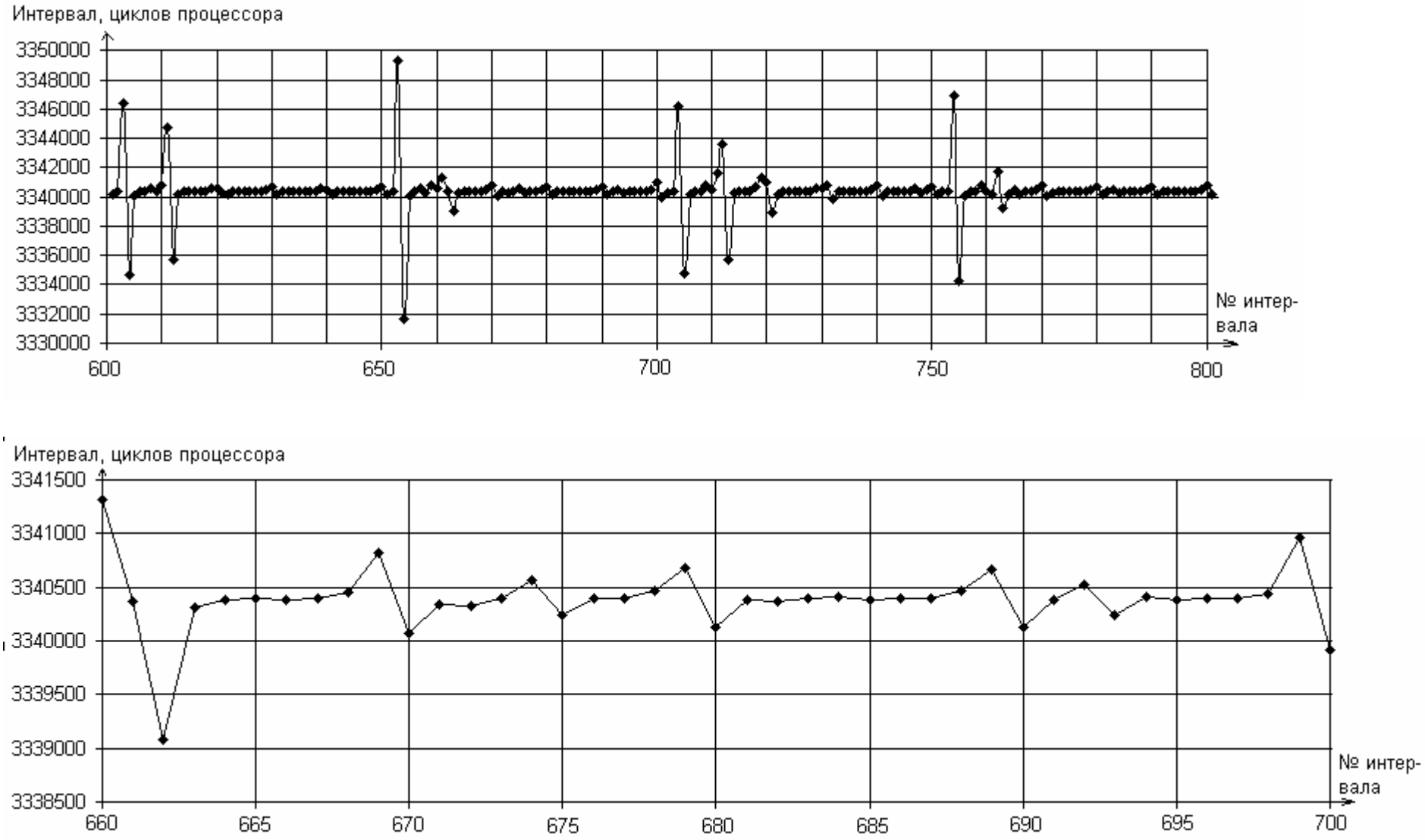


Рис. 2.8. Реализации случайного процесса последовательных интервалов периодического таймера (окончание)

интервала от предыдущего (LAG PLOT с запаздыванием 1 по терминологии DataPlot) для фрагмента реализации с номерами интервалов 2400…3200. Большинство точек группируются вблизи среднего значения ≈4.99 тика, близкого к заданному. Типичными для этих графиков являются 3 “языка”: горизонтальный вправо, вертикальный вниз и наклонный под углом 135°. После интервалов, больших среднего (горизонтальный “язык”), система отрабатывает значение, близкое к среднему. После интервалов, приблизительно равных заданному (вертикальный “язык”), система имеет тенденцию отрабатывать интервалы с меньшими значениями. Интервалы с меньшими чем заданное значениями система компенсирует последующими большими значениями (наклонный “язык”).

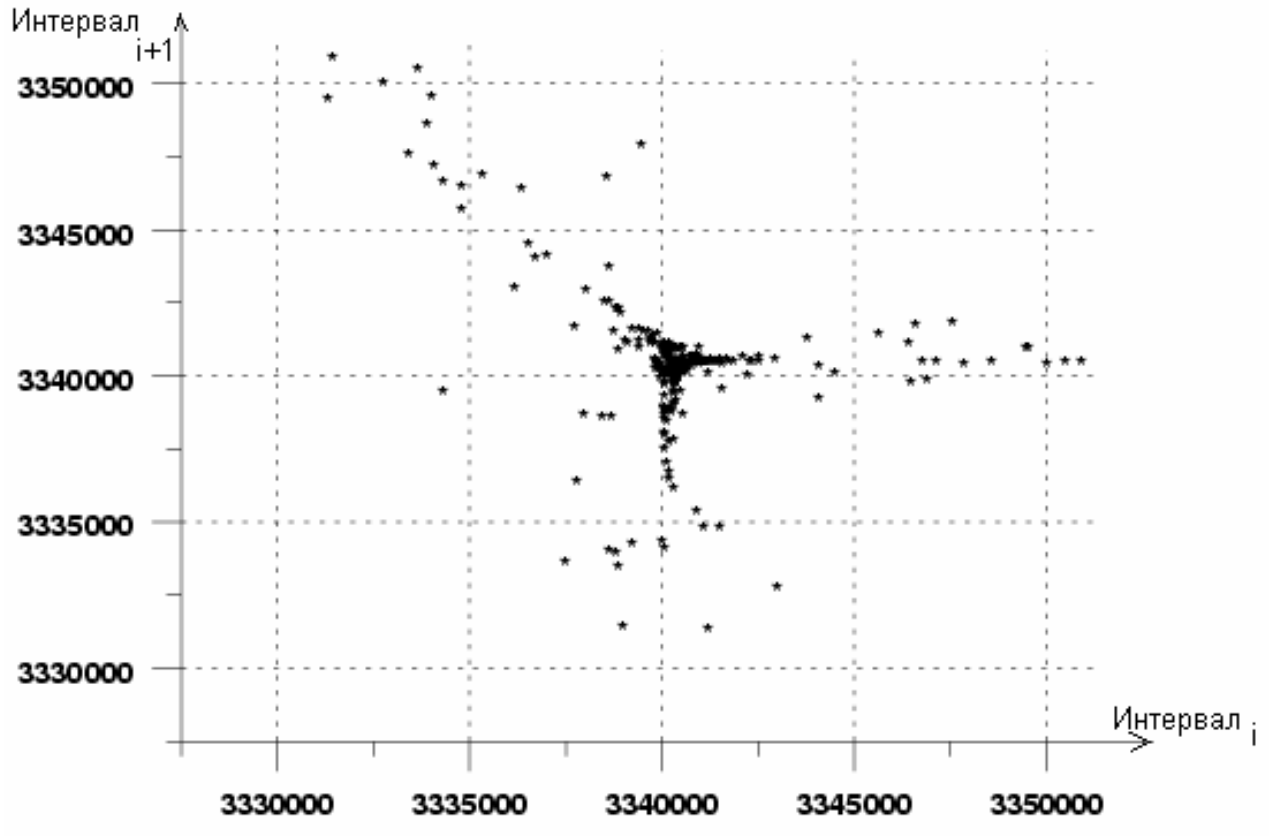


Рис. 2.9. Зависимость между i-м и i+1-м интервалами периодического таймера (LAG PLOT с запаздыванием 1)

Периодичность повторения значений интервалов видна на автокорреляционной функции (рис.2.10). Период составляет ≈50 интервалов, или 50*(5*999847нс) / 1000000000 нс/с ≈ 0,25 секунды. В частотной области периодичность хорошо видна на графике выборочного спектра мощности (рис. 2.11).

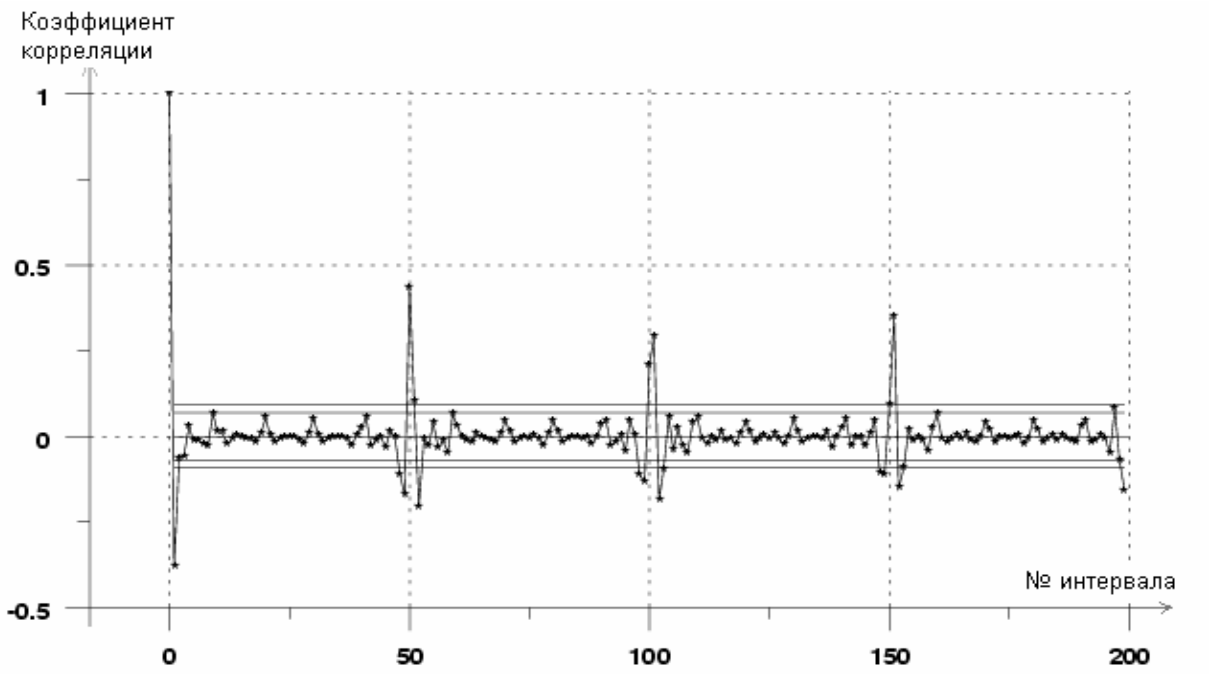


Рис. 2.10. Выборочная автокорреляционная функция случайного процесса последовательных интервалов периодического таймера

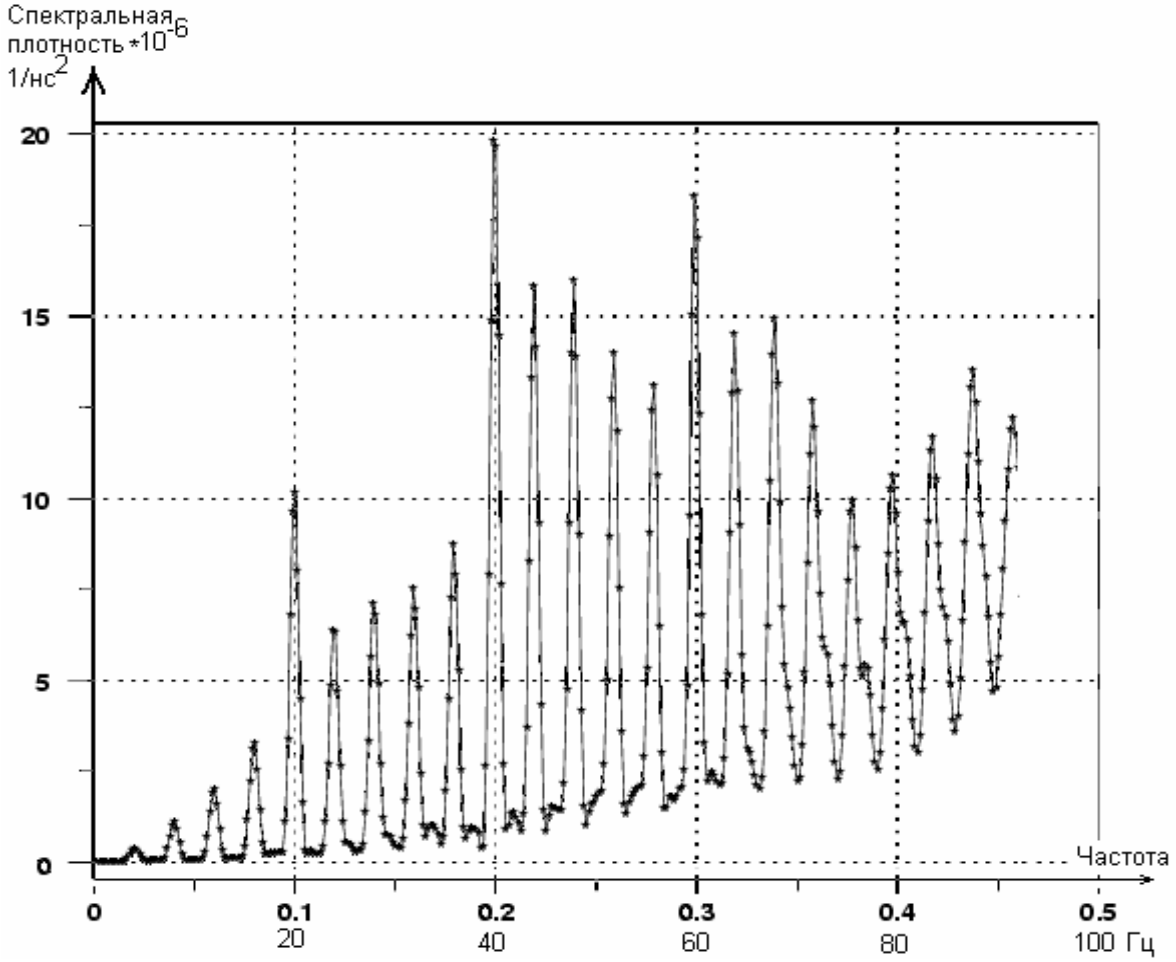


Рис. 2.11. Выборочный спектр мощности случайного процесса последовательных интервалов периодического таймера

Интересно также отметить устойчивый рост амплитуды малых колебаний значений интервала, начиная с №1988 (рис.2.12). Какие внутренние механизмы ответственны за это, не ясно, однако такой рост присутствует в любой реализации, и не обязательно в этом месте.

Стабильность работы таймера можно оценить, сопоставив две выборки значений интервалов, отстоящих друг от друга на некотором расстоянии по времени. Сравнение может производиться как по точечным оценкам практически важных параметров работы таймера – среднему значению интервала и его среднеквадратическому отклонению, так и по выборочным функциям распределения вероятности. Совпадение в статистическом смысле распределений вероятности обеих выборок можно считать достаточным признаком стабильности работы таймера. С практической точки зрения в качестве показателя стабильности можно ограничиться равенством средних значений и, возможно, дисперсий, хотя другие характеристики, в частности выбросы, могут отличаться. Несмотря на то, что каждая из выборок интервалов представляет собой реализацию случайного процесса, при сравнении их можно рассматривать как множество независимых (или почти независимых) значений одномерной случайной величины без учёта связи последовательных значений процесса между собой. Это допущение позволяет применить к сравнению критерии (тесты) однородности (статистически значимого совпадения тех или иных характеристик) несвязанных выборок классической математической статистики. Наибольший интерес представляют тесты однородности (*Goodness of Fit*) законов распределения вероятности двух или нескольких выборок, поскольку для одномерных случайных величин функция распределения является исчерпывающей характеристикой, и совпадение в статистическом смысле функций распределения двух случайных величин означает совпадение средних значений и всех других характеристик. Следует иметь в виду, что критерии однородности распределения вероятности применяются для:

а) сравнения двух случайных выборок между собой (двухвыборочные критерии);
б) сравнения эмпирического распределения случайной величины с теоретическим.

В обоих случаях алгоритмы расчёта статистик критериев одинаковы, но критические значения – разные. Критические значения и p-значения для одного случая не годятся для другого и наоборот.

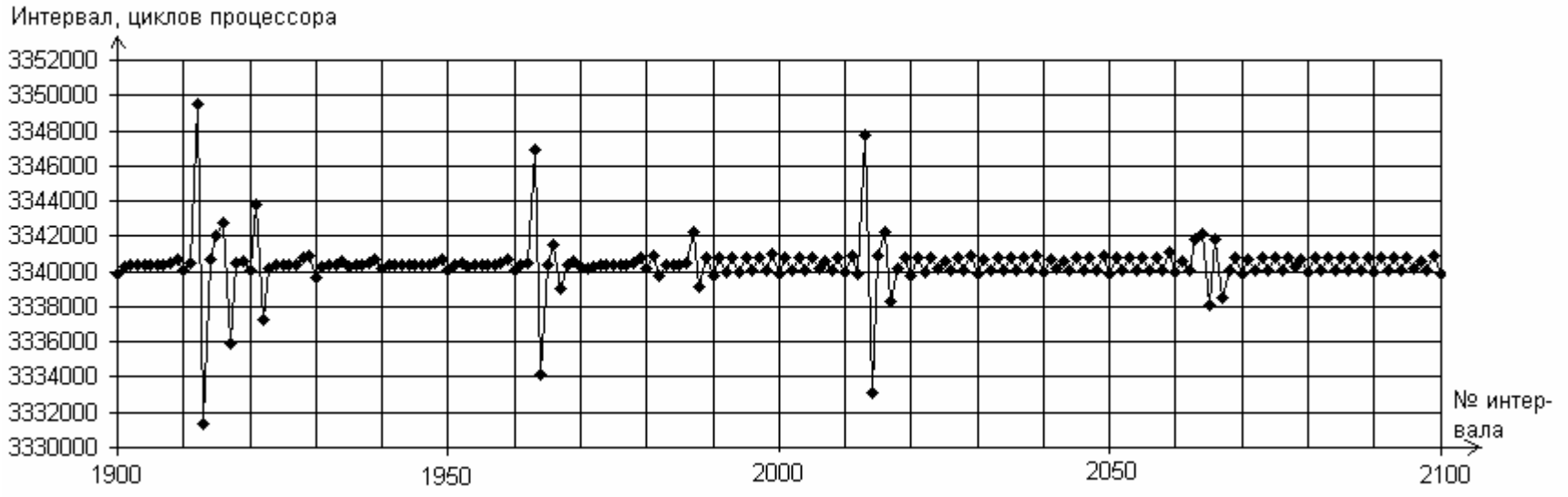


Рис. 2.12. Фрагмент реализации случайного процесса последовательных интервалов периодического таймера

В литературе описано большое количество критериев с ещё бóльшим количеством их модификаций, условий и правил их применения. Выбрать подходящий критерий однородности, не имея соответствующей практики, сложно. Здесь может быть полезен опыт применения критериев в различных сферах науки и инженерии. Подробная сводка критериев с условиями их применимости, используемых в статистическом модуле Statistical Toolkit пакета программ для моделирования взаимодействия элементарных частиц с веществом, разработанном Итальянским национальным институтом ядерной физики, приведена в [71]. Один из самых мощных критериев – критерий Андерсона-Дарлинга однородности k выборок – подробно описан в справочнике [72]. В русскоязычной статистической литературе подробное описание некоторых тестов сравнения двух выборок приведено в [65]. Полезными могут быть раздел VI справочника [73] и книга [74]. Два самых мощных критерия — Крамера - фон Мизеса (ω^2) и Андерсона-Дарлинга реализованы в виде подпрограмм-функций в библиотеке nonpaired_stat_lib.a. Однако их использование ограничивается трудоёмкостью нахождения критических значений. Практически нами использовались критерии однородности, реализованные в пакете DataPlot, описание их содержится в [67]. В табл. 2.4 приведены использованные тесты.

Таблица 2.4

Статистические тесты для сравнения независимых выборок

Назначение критерия	Название критерия	Ссылка на подробное описание
Проверка однородности математических ожиданий (1) и медиан (2) двух выборок	(1) Крамера-Уэлча	[65] разд. 3.1
	(2) Манна-Уитни	[67] n. 7.3.5
Проверка однородности дисперсий двух выборок	Levene	[67] n. 1.3.5.10

Перед использованием критериев полезно сравнить выборки на качественном уровне и определить их основные статистические характеристики. Все приведённые ниже результаты относятся к двум выборкам с номерами отсчётов 2400…3200 (выборка 1) и 3300…4100 (выборка 2) из общей реализации, обработка данных проводилась с помощью программы DataPlot. На рис.2.13 приведена бигистограмма обеих выборок.

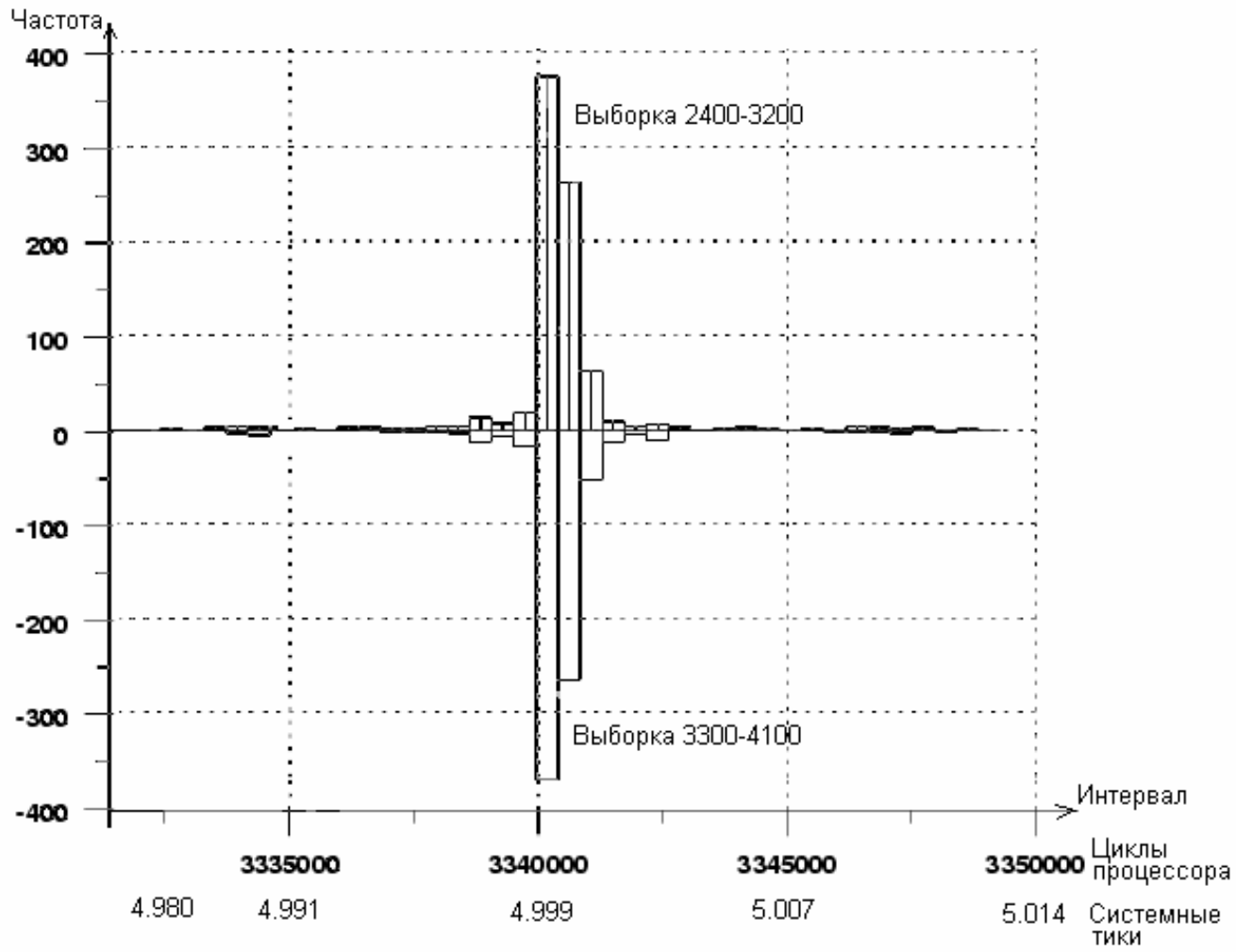


Рис.2.13. Бигистограмма двух выборок интервалов таймера

Характеристики положения и разброса выборок приведены в табл. 2.5.

Таблица 2.5

Точечные оценки характеристик положения и разброса
(все размерные величины в циклах процессора)

Характеристика	Выборка 1	Выборка 2
Количество элементов	801	801
Среднее арифметическое	0.3340400E+07	0.3340400E+07
Центр размаха (midrange = (Max+Min)/2)	0.3341110E+07	0.3340546E+07
Медиана	0.3340398E+07	0.3340398E+07
Размах = Max - Min	0.1959600E+05	0.1902500E+05
Среднеквадратическое откло-нение	0.1509211E+04	0.1458502E+04

Окончание табл. 2.5

Характеристика	Выборка 1	Выборка 2
Коэффициент вариации	4.52E-04	4.37E-04
Наибольшее значение Max	0.3350908E+07	0.3350058E+07
Наименьшее значение Min	0.3331312E+07	0.3331033E+07
Асимметрия	1.13	0.215
Эксцесс	23.96	21.87

Внешний вид бигистограммы и данные таблицы позволяют сделать следующие выводы:

- таймер точно выдерживает заданный интервал: средние значения составляют 4.999 тика для обеих выборок, коэффициенты вариации малы;

- распределения вероятностей обеих выборок близки между собой, симметричны относительно среднего значения (малые значения асимметрии) и заметно более островершинны, чем нормальное распределение за счёт более тесной группировки значений относительно средних (эксцесс >> 0-значения для стандартного нормального распределения). Степень отличия от нормального распределения следует проверить отдельно (см. табл. 2.6);

- если ненормальность обеих подтвердится, проверку их однородности следует проводить с использованием непараметрических критериев.

Проверка нормальности по двум критериям (см. табл. 2.6) подтвердила справедливость альтернативной гипотезы об отличии распределения вероятности обеих выборок от нормального.

Результаты проверки однородности выборок приведены в табл.2.7.

По результатам тестов нет оснований считать математические ожидания, дисперсии и законы распределения двух выборок различными. В пределах рассмотренного короткого промежутка времени таймер работает стабильно. Стабильность для продолжительных периодов работы должна оцениваться отдельно.

.

Таблица 2.6

Проверка нормальности Выборка1 / Выборка 2

Тест	Значение тестовой статистики	Критические значения
Вилк-Шапиро	0.4841332 / 0.5088028	Вероятность выборочного значения статистики за счёт случайных причин p-value 0.512E-42/0.267E-41
Андерсон-Дарлинг	156.9 / 147.5	α = 10 % 0.6560000 α = 5% 0.7870000 α = 2.5% 0.9180000 α = 1% 1.092000

Таблица 2.7

Результаты проверки однородности несвязанных выборок

Критерий		Значение статистики K	Критические значения Kкр	Условие принятия нуль-гипотезы об однородности
Равенства математических ожиданий (1) и медиан (2)	(1)Крамера-Уэлча	0	См. табл. для норм. распределения	K< Kкр
	(2)Манна-Уитни	0.7002702 (нормальная аппроксимация)	Интервалы принятия альтернативной гипотезы (0.,0.025) (0.975,1)	Значение K вне интервалов принятия альтернативной гипотезы
Однородности дисперсий Levene		0.4779765	α=10 % 1.137847 α=5 % 1.180328 α= 1 % 1.264568	K < Kкр
Однородности распределений	Хи-квадрат	34.21 (35 степеней свободы)	α= 10% 46.05879 α= 5% 49.80185 α= 1% 57.34208	
	Колмогорова-Смирнова	0.0362	α= 10% 0.06113 α= 5% 0.06784 α= 1% 0.08127	

2.6.3. Измерение интервалов срабатывания однократных таймеров

Рассмотрим соответствие друг другу двух способов измерения интервала срабатывания однократного таймера, основанных на:

1) разности отсчётов системного времени, возвращаемого функцией ClockTime() и
2) разности отсчётов счётчика TSC процессора, возвращаемых вызовом ClockCycles().

Тестовая программа (см. п. 3.10) задаёт случайный интервал срабатывания однократного таймера и замеряет этот интервал двумя способами. Запись замеряемых значений в файл vivod.txt осуществляется многократным выполнением команды paired_gen.out >> vivod.txt. С точки зрения математической статистики накапливаемые в файле vivod.txt значения двух измерений каждого интервала представляют собой парные (paired), или связанные (tied), или коррелированные (correlated) выборки. Для их сравнения разработаны критерии однородности, учитывающие взаимную связь каждой пары элементов двух выборок через независимую измеряемую величину. Статистической проверке подвергаются не обе исходные выборки, а новая выборка **Z**, каждый элемент которой есть разность между парами элементов исходных выборок. Нуль-гипотеза формулируется как однородность (отсутствие различий в статистическом смысле) между математическими ожиданиями и распределениями вероятности обеих выборок, альтернативная гипотеза – как различие этих характеристик случайных величин. В книге [65] в зависимости от вида альтернативной гипотезы предлагается использовать критерии, приведённые в табл. 2.8.

При справедливости нуль-гипотез статистики критериев 1 и 2 из табл. 2.8 распределены асимптотически нормально с математическим ожиданием 0 и дисперсией 1. Нормальный закон распределения Ф(x) с такими параметрами называется стандартным, соответствующая плотность вероятности обозначена φ(x):

$$\varphi(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}, \Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{t^2}{2}} dt.$$

Практически распределение статистик критериев 1 и 2 можно считать нормальным уже при объёмах выборок $n(n+1)/2 > 20$. Критическое значение $К_{кр}$, задающее область принятия нуль-гипотезы, отсекает на

Таблица 2.8

Критерии проверки однородности связанных выборок

№ критерия	Альтернативная гипотеза для исходных выборок	Альтернативная гипотеза для выборки **Z**	Тест
1	мат.ожидание выборки 1 ≠ мат.ожиданию выборки 2	Мат.ожидание выборки **Z** ≠ 0;	$Q = \sqrt{n}\,\frac{\overline{Z}}{s(Z)}$, где $\overline{Z} = \frac{Z_1 + Z_2 + ... + Z_n}{n}$ – мат.ожидание **Z**, $s(Z) = \sqrt{\frac{1}{n-1}\sum_{j=1}^{n}(Z_j - \overline{Z})^2}$ – выборочное среднее квадратическое отклонение **Z**, проверка $\lvert Q\rvert \le Q_{кр}$
2	Функция распределения вероятности выборки 2 G(x) равна сдвинутой на величину α функции распределения выборки 1 F(x) $G(x) = F(x + a)$	Функция распределения величины **Z** не симметрична за счёт сдвига относительно 0	Знаковый ранговый тест Вилкоксона $W^{++} = \frac{W^{+} - \frac{n(n+1)}{4}}{\sqrt{\frac{n(n+1)(2n+1)}{24}}}$, где $W^{+} = \sum_{j=1}^{n} R(Z_j)Q(Z_j)$ - сумма рангов положительных значений **Z** в вариационном ряду, построенном по абсолютным величинам всех ненулевых значений **Z,** для равных значений ранги усредняются, $Q(Z_j) = \begin{cases} 1, Z_j > 0, \\ 0, Z_j < 0 \end{cases}$, проверка $\lvert W^{++}\rvert \le W_{кр}$

Окончание табл. 2.8

№ критерия	Альтернативная гипотеза для исходных выборок	Альтернативная гипотеза для выборки **Z**	Тест
3	Функции распределения вероятности обеих выборок различны (не только сдвигом)	Функция распределения H(**Z**) величины **Z** не симметрична относительно 0 $H_n(Z)+H_n(-Z)-1\neq 0$.	$\omega_n^2=\sum_{j=1}^{n}(H_n(Z_j)+H_n(-Z_j)-1)^2$, проверка $\omega_n^2\le(\omega_n^2)_{кр}$

Таблица 2.9

Критические значения $K_{кр}$ нормально распределённой статистики

Уровень значимости α, %	0.1	0.5	1	2.5	5	10
Односторонний критерий	3.090	2.576	2.326	1.960	1.645	1.282
Двусторонний критерий	3.291	2.807	2.576	2.241	1.960	1.645

Таблица 2.10

Критические значения статистики ω_n^2 для проверки симметрии распределения

Значение функции распределения $S_0(x)$	Уровень значимости $\alpha=(1-S_0(x))*100$, %	Критическое значение *x* статистики ω_n^2
0,90	10	1,20
0,95	5	1,66
0,99	1	2,80

оси x часть функции плотности вероятности φ(x) статистики, площадь под которой равна $\alpha\%/100$ для одностороннего критерия и $\alpha\%/200$ для двустороннего критерия (см.рис.2.5 и 2.6). Соответствующие вероятностям α значения $К_{кр}$ рассчитываются по значениям так называемых квантилей u(α). Квантиль (существительное женского рода), или процентная точка, есть такое значение аргумента x, при котором Φ(x)=α (рис. 2.14).

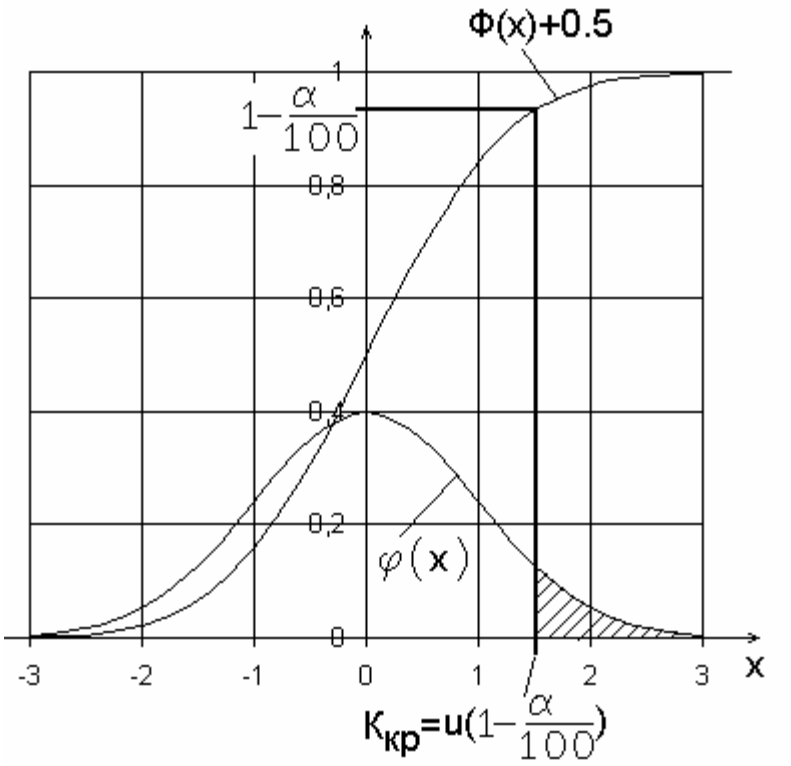


Рис. 2.14. Квантиль и критическое значение критерия для стандартного нормального распределения

Значения квантилей стандартного нормального распределения, приведены в многочисленных таблицах (см., например, [73, табл. 1.3, с.136]) или рассчитываются с помощью специализированных подпрограмм-функций математической статистики (см., например, функцию НОРМОБР() MS Excel). Для одностороннего критерия $К_{кр}=u(1-\frac{\alpha}{100})$, для двустороннего критерия $К_{кр}=u(1-\frac{\alpha}{200})$ (табл. 2.9).

Для критерия 3 из табл. 2.7 критические значения [65] задаются табл. 2.10.

Многие пакеты статистических программ, включая DataPlot, содержат знаковый ранговый тест Вилкоксона (*Wilcoxon signed rank test*), однако практически нигде нет теста типа омега-квадрат для парных выборок. Для практического использования этот тест реализован в виде функции на языке С и включён вместе с другими функциями-тестами в библиотеку paired_stat_lib.a. Дополнительно библиотека содержит подпрограмму-функцию изображения на экране гистограмм двух сравниваемых (не обязательно парных) выборок. Ниже приведен заголовочный файл paired_stat.h, содержащий прототипы библиотечных функций, а также текст функции для построения гистограммы histogram.c с иллюстрацией использования расширенного текстового режима работы терминала `ncurses`. Возможности `ncurses` будут использованы для визуализации работы компьютерной модели системы управления, являющейся конечной целью настоящего пособия. Функция histogram.c иллюстрирует использование некоторых функций этого `ncurses`.

Тестовая программа paired_gen.c запускалась в псевдотерминале графической среды Photon с приоритетом по умолчанию (10). Объём каждой из выборок m=320. На рис.2.15 приведены бигистограмма интервалов, замеренных с помощью `ClockCycles()` – переменная Y1 и `ClockTime()` - переменная Y2, а также гистограмма их разности **Z** (графики построены пакетом DataPlot).

Математическое ожидание $\overline{Z}$ =-0.00038, среднеквадратическое отклонение S(Z)=0.00056, медиана **Z**= − 0.000406. Результаты статистической обработки сведены в табл. 2.11. Данные табл.2.11 позволяют сделать следующий вывод: отклонение распределения величины Z от симметричного (см. рис. 2.15) не является случайным, нульгипотеза об однородности средних и распределений вероятности двух сравниваемых парных выборок отвергается.

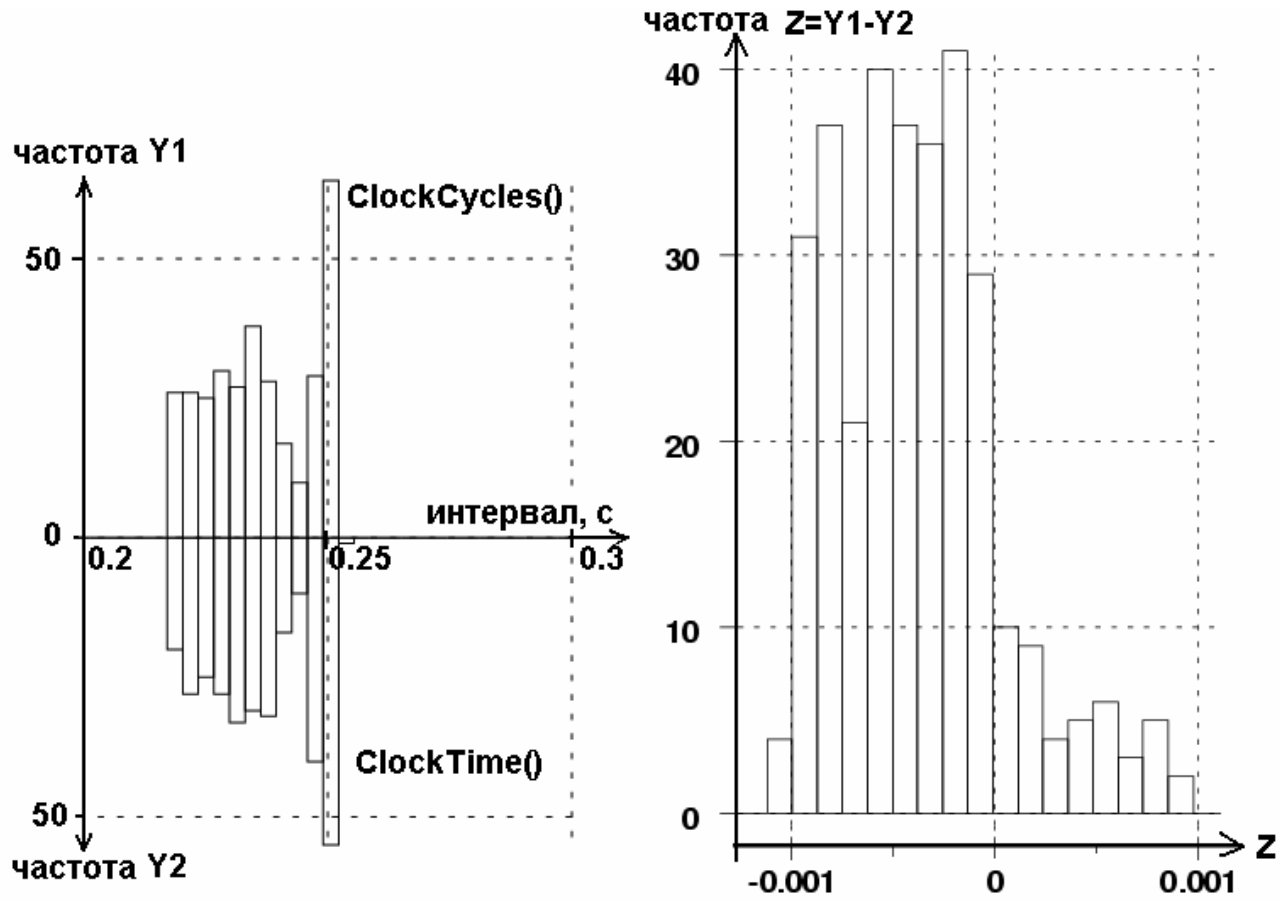


Рис.2.15 Бигистограмма случайных интервалов одноразового таймера, замеренных двумя способами, и гистограмма разностей парных замеров

Таблица 2.11

Результаты проверки однородности парных выборок

Критерий	Значение статистики	Сравнение с критическими значениями
Равенства математических ожиданий Q	12.1	Превышают критические значения для всех приведенных в табл. 2.8 и 2.9 уровней значимости
Вилкоксона W^{++}	4.72	
Типа омега-квадрат ω_n^2	64.9	

РЕЗЮМЕ

Любой программист хочет получить от операционной системы полностью предсказуемую работу предоставляемых ею системных средств. Для систем реального времени, кроме прочего, важна временна́я предсказуемость поведения системных функций. Как показано выше в этой главе, предсказуемость ведения системного времени и отсчитываемых различными видами таймеров временнЫх интервалов не всегда может считаться удовлетворительной. Строго говоря, вести речь о предсказуемости можно только в статистическом смысле, т.к. все временнЫе интервалы в системе являются псевдослучайными величинами, содержащими неслучайную переменную составляющую и чисто случайную компоненту. Более точно замеренные значения временнЫх интервалов образуют реализации случайных процессов.

Причины непостоянства интервалов времени многообразны. Здесь вносят свой вклад:

- принципиально дискретный ход системных часов, отсчитывающих непрерывное время, причём величина системного тика не выражается точно целым числом микросекунд;
- флуктуации частоты кварцевых осцилляторов аппаратных таймеров, лежащих в основе всех средств службы времени;
- не вполне предсказуемое и зависящее от конкретного программного окружения поведение других компонентов аппаратуры, например системы аппаратных прерываний, кэшей, конвейеров процессора;
- механизмы реализации операционной системой таймеров с учётом требований стандарта POSIX “не раньше чем”.

Определённую погрешность в оценку точности и стабильности временных интервалов могут вносить способы замера, использующие те же аппаратные таймеры и работающие в системе параллельно с программными единицами, реализующими сами таймеры. Для более точной оценки величины временных интервалов следует применять внешние измерительные аппаратные устройства.

Разработаны и в различной аппаратуре с помощью специальных программных модулей реализованы способы корректировки системного времени компьютерных устройств, позволяющие обеспечить весьма хорошую точность хода компьютерных часов по отношению к внешнему “физическому времени”. При этом могут использоваться внешние точные источники времени или внутренние высокостабиль-

ные аппаратные счётчики (к ним относится и регистр TSC микропроцессора) с точно известной в каждом конкретном случае частотой.

ОС QNX/Neutrino имеет свою нишу применения, в которой не предъявляется очень высоких требований к точности хода системных часов. Вместе с тем, программные таймеры должны работать стабильно даже в случае, если системное время подвергается корректировке. Системный вызов `ClockAdjust()` позволяет плавно "подводить" системные часы таким образом, чтобы скачкообразно не нарушалась работа запущенных таймеров. Хотя библиотека предоставляет программисту довольно ограниченный набор функций для корректировки и поддержания точности часов (фактически только `ClockAdjust()`), эффективная микроядерная архитектура и весь богатый набор средств API даёт возможность программисту при необходимости как использовать в службе времени дополнительные аппаратные таймеры помимо станлартного 8254, так и осуществлять корректировку системного времени, привязываясь к внешним серверам точного времени.

Изложенные в главе сведения о статистической обработке выборок случайных величин и соответствующие примеры могут быть приняты в качестве начальной основы при грубой оценке точности и стабильности таймеров. Следует иметь в виду, что в математической статистике и теории случайных процессов к настоящему времени разработаны более точные и информативные методы.

3. ТЕКСТЫ ПРОГРАММ

Эти программы-примеры иллюстрируют применение различных средств службы времени QNX/Neutrino. Тексты программ могут служить справочным материалом при выполнении учебных заданий и написании реальных проектов.

3.1. ПРОГРАММА, ИЛЛЮСТРИРУЮЩАЯ ПРИМЕНЕНИЕ СРЕДСТВ ЗАДАНИЯ, ИЗМЕНЕНИЯ И ПОДГОНКИ СИСТЕМНОГО ВРЕМЕНИ

```
//Только суперюзер (root) имеет право устанавливать и корректировать системное время.
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/neutrino.h>
#include <inttypes.h>
#include <errno.h>
#include <hw/inout.h>
#define BUFF_SIZE (256 + 1)
//-------------------------------------------------------------
Get_ExactTime(volatile long long * old_t, long long * new_t)
{    /* Функция симулирует получение “точного” времени new_t.  В действительности точное время
        может  быть получено  с сервера или  введено  с  клавиатуры */
    long long delta;
    delta =  -300000000000LL;    // ±300 seconds –насколько отстают/спешат системные часы

    printf("FUNCTION delta= %lld nanoseconds = %f minutes\n", delta,
          (float) delta/1000000000LL/60LL);
```

```
    *new_t = *old_t + delta;
    return * new_t;
}
typedef struct
/* Структура содержит параметры звукового сигнала будильника, генерируемые по завершении ядром
корректировки хода системных часов */
{
    unsigned int Freq;      // Частота звука, Гц
    unsigned int Dur;       // Длительность звука, мс
    unsigned int Inter;     //  Интервал между звуками, мс
    unsigned int Numb;      //  Количество повторений звука
} Sound_Param;
//---------------------------------------------------------------
void PlaySound(Sound_Param sp);
int main( void )
{
    struct tm new_time_of_day,  // Задаваемые для установки системных часов дата и время
    struct timespec time_spec;  /* Задаваемые дата и время в формате, понятном функции
                                   clock_settime() */
    time_t time_of_day;         // Системное время в нс
    char   buffer[ 80 ];              // Буфер для хранения “разорванного” времени
    char   buff[BUFF_SIZE];     // Буфер для хранения значения врЕменной зоны
    char   *tz;
    uint64_t  old_system_time;  // “Старое”-перед корректировкой - системное время
    long long old_t, new_t, delta;  /*”Старое” и заданное для корректировки системное
```

время и разница между для функции
Get_ExactTime()*/
```
    struct _clockadjust clka; /*Структура, содержащая параметры корректировки времени
                                с помощью функции ClockAdjust()*/
    struct _clockperiod clkp; // Структура, содержащая период хода системных часов
    int errvalue;             //Переменная для хранения кода ошибки функции ClockAdjust()
    long long wait_adjusting; /* Оценка интервала времени, в течение которого ядро
                                 корректирует системные часы */

    /*Выводим текущее системное время и значение переменной среды TZ и
      конфигурационной переменной CS_TIMEZONE */
    time_of_day = time( NULL );   // Текущее системное время
    strftime( buffer, 80, "Now UTC system time is %A %B %d %Y %T",
              gmtime(&time_of_day));
    printf( "%s\n", buffer );
    if( confstr( _CS_TIMEZONE, buff, BUFF_SIZE ) > 0 ) {
        printf( "default TIMEZONE name, returned by confstr( _CS_TIMEZONE)
                 is: %s\n", buff );
    }
    printf("getconf utility returns default TIMEZONE (see next line):\n");
    system("getconf CS_TIMEZONE");

    printf( "TZ by getenv(): %s\n",
    (tz = getenv( "TZ" )) ? tz : strcat(buff, "(default-TZ is not set)"));
    printf("Global TimeZone variables have the following values: ");
    printf( "daylight: %d ", daylight );
```

```
printf( "timezone: %ld ", timezone );
printf( "time zone names: %s %s\n", tzname[0], tzname[1] );

// Задаём значения и устанавливаем “скачком” новые системную дату и время
printf("   CHANGING SYSTEM TIME...\n");
new_time_of_day.tm_year = 2004 - 1900;
new_time_of_day.tm_mon = 10-1;
new_time_of_day.tm_mday = 27;
new_time_of_day.tm_hour = 14;
new_time_of_day.tm_min = 21;
new_time_of_day.tm_sec = 0;
new_time_of_day.tm_isdst = 1;

time_spec.tv_sec = mktime( &new_time_of_day );
time_spec.tv_nsec = 0;

if( clock_settime( CLOCK_REALTIME, &time_spec) == -1 )
{
    perror( "setclock" );
    return EXIT_FAILURE;
}
system("rtc -s hw"); /*Устанавливаем аппаратные часы реального времени в соответствии с
                                текущим системным временем */
/*Выводим локальные и универсальные (UTC) дату-время , соответствующие текущему
  системному   времени, в виде удобовоспринимаемой строки */
time_of_day = time( NULL );
```

```
strftime( buffer, 80, "Today is %A %B %d %Y %T",
          localtime( &time_of_day ) );
printf( "%s\n", buffer );
strftime( buffer, 80, "Today is UTC %A %B %d %Y %T",
           gmtime(&time_of_day));
printf( "%s\n", buffer );

printf("   date command returns local time:\n");
system("date");
printf("   Today is UTC time:\n");
system("date -u");
```


//Устанавливаем переменную окружения TZ и выводим местное время для разных часовых поясов

```
printf("   SETTING TIME ZONES and EVALUATING LOCAL TIMES...\n");
setenv( "TZ", "ANAST-13,M3.5.0/2,M10.5.0/2", 1 );
tzset();
strftime( buffer, 80, "   Kamchatka - local time is %A %B %d %Y %T",
          localtime( &time_of_day ) );
printf( "%s\n", buffer );

setenv( "TZ",  "NOVST-07,M3.5.0/2,M10.5.0/2", 1 );
tzset();
strftime( buffer, 80, "   Novosibirsk - local time is %A %B %d %Y %T",
           localtime( &time_of_day ) );
printf( "%s\n", buffer );
```

```
setenv( "TZ",  NULL, 1 );
tzset();
strftime( buffer, 80, "      Moscow - local time is %A %B %d %Y %T",
          localtime( &time_of_day ) );
printf( "%s\n", buffer );

setenv( "TZ",  "CEST-02,M3.5.0/2,M10.5.0/2", 1 );
tzset();
strftime( buffer, 80, "      Madrid - local time is %A %B %d %Y %T",
          localtime( &time_of_day ) );
printf( "%s\n", buffer );

setenv( "TZ",  "eng-01,M3.5.0/2,M10.5.0/2", 1 );
tzset();
strftime( buffer, 80, "      London - local time is %A %B %d %Y %T",
          localtime( &time_of_day ) );
printf( "%s\n", buffer );

setenv( "TZ",  "EDT04,M4.1.0/2,M10.5.0/2", 1 );
tzset();
strftime( buffer, 80, "      New York - local time is %A %B %d %Y %T",
localtime( &time_of_day ) );
printf( "%s\n", buffer );

setenv( "TZ",  "AKDT08,M4.1.0/2,M10.5.0/2", 1 );
tzset();
```

```
strftime( buffer, 80, "       Alaska - local time is %A %B %d %Y %T",
localtime( &time_of_day ) );
printf( "%s\n", buffer );

setenv( "TZ",  NULL, 1 );
tzset();
strftime( buffer, 80, "       Bryansk - local time is %A %B %d %Y %T",
          localtime( &time_of_day ) );
printf( "%s\n", buffer );

/*Подготавливаем подстройку системных часов под "точное" время,
  возвращаемое ф-цией Get_ExactTime()*/
printf("   ADJUSTING SYSTEM TIME...\n");
ClockTime( CLOCK_REALTIME, NULL,  &old_system_time ); /* текущее системное
                                                         время в нс */
old_t = (long long) old_system_time;
Get_ExactTime(&old_t, &new_t);        // Получаем "точное" время
printf("Current system time is %lld nanoseconds = ",  old_t);

time_of_day = time( NULL );
strftime( buffer, 80, " UTC %A %B %d %Y %T",  gmtime(&time_of_day));
printf( "%s\n", buffer );

delta = new_t - old_t; // Разница между "точным" и текущим системным временем
printf("Adjusting system clock for %f sec to new system time %lld
        nanoseconds\n",(double) (new_t - old_t)/ 1000000000LL, new_t);
```

```
ClockPeriod( CLOCK_REALTIME, NULL, &clkp, 0 );
clka.tick_nsec_inc = (long) (clkp.nsec / 4UL); /* Задаём корректировочную
                                добавку для каждого тика системного времени равной 0.25 тика*/
if (delta < 0) clka.tick_nsec_inc = -clka.tick_nsec_inc;
//Определяем количество системных тиков, потребных для заданной корректировки
clka.tick_count = (unsigned long) ((double) delta/clka.tick_nsec_inc);
printf(" for the space of %ld system ticks", clka.tick_count);
printf("  with tick increment %ld nsec\n", clka.tick_nsec_inc);
//Запускаем корректировку системных часов
errno = EOK;
ClockAdjust( CLOCK_REALTIME, &clka, NULL );
errvalue = errno;
printf( "The error generated by ClockAdjust() was %d, that means:
        %s\n", errvalue, strerror( errvalue ));
/* Оцениваем интервал wait_adjusting реального времени, требуемый для подгонки системных часов */
wait_adjusting = ((long long) clka.tick_count *
            (long long) (clkp.nsec+clka.tick_nsec_inc)) / 1000000000LL;
printf("wait_adjusting = %lld sec\n", wait_adjusting);
printf("Sleeping %f seconds while the kernel adjusts system
        clock...\n",(float) wait_adjusting);
/* Запускаем программу phlocale с отображением системных часов. Двойной повтор одного и того же
пути-не ошибка. Он предусмотрен для тех файлов, запуск которых может осуществляться под
разными именами (например, gzip и gunzip) */
spawnl( P_NOWAIT, "/usr/photon/bin/phlocale",
       "/usr/photon/bin/phlocale", "-t", NULL);
sleep(wait_adjusting);
```

/* “засыпаем” на wait_adjusting секунд, ожидая окончания корректировки системного времени. Наблюдаем за ходом корректировки в окне phlocale (см. рис.1.1). Ядро выполняет подгонку системных часов независимо от состояния потока, запустившего функцию ClockAdjust(). Обратите внимание на индикатор загрузки процессора во время и по окончании корректировки для случаев delta>0 и delta<0 */

```
    Sound_Param alarm_sound={1500, 120, 30, 8};// Проснувшись, подаём звуковой
    PlaySound(alarm_sound);                    // сигнал
    system("rtc -s hw");   /*Синхронизируем аппаратные часы реального времени с
                            системными часами  */
    //Выводим скорректированное универсальное (UTC) системное время и местное время
    time_of_day = time( NULL );
    strftime( buffer, 80, "Today is %A %B %d %Y %T",
                localtime( &time_of_day ) );
    printf( "%s\n", buffer );
    strftime( buffer, 80, "Today is UTC %A %B %d %Y %T",
               gmtime(&time_of_day));
    printf( "%s\n", buffer );

    printf("   date command returns local time:\n");
    system("date");
    printf("   Today is UTC time:\n");
    system("date -u");

    return EXIT_SUCCESS;
}
void PlaySound(Sound_Param sp)
//Воспроизведение звука через встроенный динамик
```

```
{
    short int port61val, lsb, msb, i;
    unsigned int CounterValue;
    ThreadCtl(_NTO_TCTL_IO, 0);
    CounterValue = 1193181.0 / sp.Freq;   /* Делитель канала 2 аппаратного таймера,
                                             определяемый заданной частотой sp.Freq звука */
    lsb=CounterValue & 0x00FF;
    msb=CounterValue >> 8;
    out8(0x43, 0xB6); /* Записываем управляющее слово в управляющий регистр
                         аппаратного таймера */
    out8(0x42, lsb;// Записываем младший байт делителя частоты в 16-битный счётчик таймера
    out8(0x42, msb);// Записываем старший байт делителя частоты в 16-битный счётчик таймера
    for (i=0; i < sp.Numb; i++)
    {
        port61val=in8(0x61);
        out8(0x61, port61val | 0x03); /* Подключаем встроенный динамик ко второму
                                         каналу аппаратного таймера */
        delay(sp.Dur);                // Выдерживаем заданное время звучания
        port61val=in8(0x61);
        out8(0x61, port61val & 0xFC); // Отключаем динамик
        delay(sp.Inter);              // Выдерживаем заданную паузу между звуками
    }
}
```

3.2. ПРОГРАММА, РЕАЛИЗУЮЩАЯ АБСОЛЮТНЫЙ ТАЙМЕР ЗАДЕРЖКИ НА ОСНОВЕ ФУНКЦИИ clock_nanosleep( )

```
#include <time.h>
#include <unistd.h>
#define BUFF_SIZE (256 + 1)

main()
{
    struct     timespec rqtp;
    time_t     start_time, time_of_day;
    struct tm  inputtm;
    time_t     dt;
    char       buff[BUFF_SIZE];

    // Установливаем дату и время срабатывания однократного таймера на 25 мая 2004 г. 14:52
    inputtm.tm_sec  = 0;          // секунд в последней неполной минуте
    inputtm.tm_min  = 52;         // минут в последнем неполном часе
    inputtm.tm_hour = 14;         // часов после полуночи
    inputtm.tm_mday = 25;         // день месяца
    inputtm.tm_mon  = 5-1;        // месяц (январь имеет порядковый номер 0)
    inputtm.tm_year = 2004-1900;  // Год от точки отсчёта 1900

    dt = mktime(&inputtm); // Преобразуем местное время срабатывания таймера в календарное
//Выводим заданную дату срабатывания таймера, имя  временной зоны и её смещение в сек от UTC
```

```
    printf("dt = %ld seconds UTC\n", (long)dt);
    printf("The given date (UTC) is %d-%d-%d ",  inputtm.tm_mday,
            inputtm.tm_mon+1, inputtm.tm_year+1900);
    printf("%d:%d:%d\n", inputtm.tm_hour,  inputtm.tm_min,inputtm.tm_sec);
    printf("Offset from UTC is %ld seconds, time zone name is %s\n",
            inputtm.tm_gmtoff, inputtm.tm_zone);
    if( confstr( _CS_TIMEZONE, buff, BUFF_SIZE  ) != 0 )
         printf( "TimeZone set to: %s\n", buff );    // значение _CS_TIMEZONE
    printf("%d seconds between 1-1-1970, 00:00:00 and given absolute UTC
            date & time\n", dt);   /* Заданное календарное время срабатывания
                                      таймера в секундах  */
    printf("%d seconds between 1-1-1970, 00:00:00 and given local date &
            time\n", dt+inputtm.tm_gmtoff);   /* Заданное местное время срабатывания
                                      таймера в секундах  от точки отсчёта Unix Epoch*/
    start_time = time( NULL );
    rqtp.tv_sec = dt;
    rqtp.tv_nsec = 0L;
    // Запустим таймер ожидания как абсолютный
    clock_nanosleep( CLOCK_REALTIME, TIMER_ABSTIME, &rqtp, NULL);
    time_of_day = time(NULL);
    printf("Timer expired. Now is %s\n", ctime( &time_of_day));
    printf("Elapsed %f seconds since timer started\n",
         difftime( time_of_day, start_time ));     /* Сколько секунд процесс был
                                                      блокирован таймером */
}
```

3.3. ПРИМЕРЫ ПЕРИОДИЧЕСКИХ ТАЙМЕРОВ, РЕАЛИЗОВАННЫХ СИСТЕМНЫМИ ВЫЗОВАМИ `TimerAlarm()` И `TimerSettime()` С УВЕДОМЛЕНИЕМ СИГНАЛАМИ

Программа иллюстрирует также способы получения информации о состоянии таймеров и использование сигналов.

```
#include <sys/neutrino.h>
#include <pthread.h>
#include <inttypes.h>
#define BILLION  1000000000L;

    timer_t  timer_id, timer_id2;
    struct _timer_info info, info2;
//-------------------------------------------------------
void*  grandchild_function()  /* Эта функция выполняется потоком GRANDCHILD,
                                 созданным потоком CHILD */
{
    struct sigevent    event, event2;
    int                value = 123,value2 = 234; /* Значения, переносимые
                                                    сигналами от таймеров*/
    struct _itimer     itime, itime2;
    sigset_t           set_grandchild;
    extern void        handler();
    struct sigaction   act;
    int                count = 0;
    int                i, j, k;
    time_t             timer_start;
```

```
uint32_t          timerinfo_flags;
struct timespec   start, stop;
double            accum;
struct timespec   rqtp;
printf( "The GRANDCHILD %d starts with priority %d\n", pthread_self(),
getprio(0) );
 sigemptyset(&set_grandchild); /* Дополнительное множество сигналов, которые будут
замаскированы (блокированы) при выполнении обработчика сигналов. Здесь-пустое множество*/
act.sa_flags = 0;       // Задаём действия по получении сигналов SIGRTMIN и SIGINT
act.sa_mask = set_grandchild; // Иначе эти сигналы будут по умолчанию убивать поток
act.sa_handler = &handler;
SignalAction(0, __signalstub, SIGRTMIN, &act, NULL);
SignalAction(0, __signalstub, SIGINT, &act, NULL);

// Задаём схему уведомления для таймера 1 – сигнал SIGRTMIN, посланный вызывающему потоку
event.sigev_notify = SIGEV_SIGNAL_THREAD;
event.sigev_signo = SIGRTMIN;      //SIGRTMIN = 41
event.sigev_value.sival_int = value;/* =123 переносимое сигналом 32 разрядное
                                    значение, интерпретируемое обработчиком сигналов */
event.sigev_code = SI_TIMER;  /* Определённый стандартом POSIX 1003.1 код сигнала
                                    от таймера SI_TIMER=-3 */
// То же для таймера 2
event2.sigev_notify = SIGEV_SIGNAL_THREAD;
event2.sigev_signo = SIGRTMIN;
event2.sigev_value.sival_int = value2;    //=234;
```

```
event2.sigev_code = SI_TIMER;

timer_id = TimerCreate(CLOCK_REALTIME, &event); // Создаём два таймера
timer_id2 = TimerCreate(CLOCK_REALTIME, &event2);
TimerInfo(0, timer_id, 0, &info); // Получим и выведем информацию о таймере 1
timerinfo_flags = info.flags;
if (timerinfo_flags == _NTO_TI_ACTIVE)
    printf("Timer %d is active, ", timer_id);
else printf("Timer %d is not active, ", timer_id);
if (timerinfo_flags ==_NTO_TI_ABSOLUTE)printf("absolute\n", timer_id);
else printf("relative\n", timer_id);

// Инициализируем значением периода 2.3 сек и запустим таймер 1
itime.nsec = 2300000000UL;
itime.interval_nsec = 2300000000UL;
TimerSettime(timer_id, 0, &itime, NULL);
TimerInfo(0, timer_id, 0, &info);
// Время запуска таймера 1
timer_start = (time_t) (info.itime.nsec / 1000000 / 1000);
printf("GRANDCHILD: Starts timer #%d for thread %d at (local time)%s",
       timer_id, info.tid, ctime(&timer_start));
timerinfo_flags = info.flags; /    // Получим и сообщим статус активности таймера
if (timerinfo_flags == _NTO_TI_ACTIVE)
    printf("Now timer %d is active\n", timer_id);

// Инициализируем значением периода 7.5 сек и запустим таймер 2
```

```
itime2.nsec = 7500000000UL;
itime2.interval_nsec =  7500000000UL;
TimerSettime(timer_id2, 0, &itime2, NULL);
TimerInfo(0, timer_id2, 0, &info);
// Время запуска таймера 2
timer_start = (time_t) (info.itime.nsec / 1000000 / 1000);
printf("GRANDCHILD: Starts timer #%d for thread %d at(local time) %s",
        timer_id2, info.tid , ctime(&timer_start));

sigaddset(&set_grandchild, SIGRTMIN);//Добавим к маске сигналов потока SIGRTMIN

clock_gettime( CLOCK_REALTIME, &start);// Начальная засечка времени
while(1)
{
    if (count == 10) // На 10-той итерации заблокируем сигналы SIGRTMIN  от таймеров
    {
        SignalProcmask(0, 0, SIG_SETMASK, &set_grandchild, NULL);
        printf("Now-------SIGRTMIN-------is----------BLOCKED\n");
    }
    /* Нагрузим поток имитацией полезной работы с паузами случайной длительности, которые
       будут  прерываться обработчиком сигналов handler() при поступлении  незамаскиро-
       ванного  сигнала */
    ;// “Полезная” работа.
    for(i=0; i<50000; i++) for(j=0; j<5000; j++)  k=i*j;
    nsec2timespec(&rqtp, (_uint64)(lrand48()+100000000LL));
    nanosleep(&rqtp, NULL);     // Случайная пауза в работе
```

```
        if(count >=10)  /* Начиная с 10-той итерации выводим относительное время  и
                           характеристики работы таймеров */
        {
            TimerInfo(getpid(), timer_id, 0, &info);
            TimerInfo(getpid(), timer_id2, 0, &info2);
            clock_gettime( CLOCK_REALTIME, &stop);
            accum = ( stop.tv_sec - start.tv_sec ) +
                  (double)( stop.tv_nsec - start.tv_nsec )/(double)BILLION;
            printf(" GRANDCHILD: %.2lf sec count=%d", accum, count);
            printf("   Timer %d  overruns %d remains %7.1f ms\n", timer_id,
                       info.overruns, (float) info.otime.nsec/1000000);

            printf("\t\t\tTimer %d overruns %d remains %7.1f ms\n",
              timer_id2, info2.overruns, (float) info2.otime.nsec/1000000);

        }
        count++;
    }
}
//---------------------------------------------------------
void*  child_function()    /* Эта функция выполняется потоком CHILD, созданным в потоке
                              main() (PARENT)
{
    sigset_t  set_child;
    siginfo_t info_child;
    pthread_attr_t attr;
```

```
struct _itimer timer_time;
int sigwaitinfo_ret;
struct timespec start, stop;
double accum;
printf("CHILD %d with priority %d starts\n",pthread_self(),getprio(0));

// Запустим поток GRANDCHILD как отсоединённый. Приоритет наследуется от родителя.
pthread_attr_init( &attr );
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED );
pthread_create( NULL, &attr, &grandchild_function, NULL );

sigemptyset(&set_child);   // Создадим пустую маску “отложенных (pending)” сигналов
sigaddset(&set_child, SIGALRM);    //Добавим в маску сигнал SIGALRM=14

timer_time.nsec = 5000000000UL;    /*Запустим таймер с интервалом 5 сек и схемой
                                     уведомления SIGALRM */
timer_time.interval_nsec =    5000000000UL;

TimerAlarm( CLOCK_REALTIME, &timer_time, NULL );
clock_gettime( CLOCK_REALTIME, &start);  // Начальная засечка времени
while(1)
{    /* Блокируемся в ожидании сигналов SIGALRM от таймеров.  При получении сигнала
        выводим его параметры.*/
     sigwaitinfo_ret = SignalWaitinfo(&set_child, &info_child);
     if (sigwaitinfo_ret==-1) printf("SignalWaitInfo error ");
     clock_gettime( CLOCK_REALTIME, &stop);
```

```
        accum = ( stop.tv_sec - start.tv_sec ) +
                (double)( stop.tv_nsec - start.tv_nsec ) / (double)BILLION;
        printf("\t\t\tCHILD: signal %d with code %d and value %d.
                Elapsed %.2lf sec\n" ,info_child.si_signo,
                info_child.si_code, info_child.si_value, accum);
    }
}
// ------------------------------------------------------------------------------------------------------------------
main()      // Родительский поток (PARENT)
{
    pthread_t          thread;
    pthread_attr_t     attr;
    sigset_t           set;
    int                prio;
    struct sched_param param;
    sigset_t           set2;
    siginfo_t          info;

    prio = getprio(0);
    printf( "PARENT pid=%d tid=%d with priority %d starts\n", getpid(),
              pthread_self(), getprio(0));
    sigemptyset(&set); /* Создаём пустую маску блокированных сигналов и включаем в неё
                          сигнал SIGALRM */
    sigaddset(&set, SIGALRM);

    pthread_sigmask( SIG_SETMASK, &set, NULL); // Используем созданную маску для
```

блокирования сигнала SIGALRM в данном потоке, чтобы сигнал не убивал поток (в этом заключается действие сигнала по умолчанию). Сигнал SIGALRM действует в пределах всего процесса и принимается потоком CHILD, в котором для этого предусмотрены соответствующие средства)

```
pthread_attr_init( &attr );    /* Создаём дочерний поток CHILD с приоритетом на 1
                                  больше чем у родительского */
pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED );
param.sched_priority = prio+1;
pthread_attr_setschedparam(&attr, &param );
pthread_create( &thread, &attr, &child_function, NULL );
printf("  MAIN  WAITS  SIGNALS ...\n");

sigemptyset(&set2);// Инициализируем маску “отложенных” (pending) сигналов как пустую
sigaddset(&set2, SIGRTMAX + 8);// Добавляем в неё сигнал с номером SIGRTMAX+8
while(1)
{    /* Обрабатываем “отложенные” сигналы, посланные обработчиком сигналов handler().
        Значение (value) сигнала SIGRTMAX+8  несёт код сигнала SIGRTMIN от таймера,
        позволяющее идентифицировать таймер. Код (code) сигнала SIGRTMAX+8 несёт
        количество необработанных срабатываний (overruns) таймера. */
     sigwaitinfo(&set2, &info);
     printf("PARENT on behalf of SIGNAL HANDLER:Timer %d overruns %d\n",
             info.si_value.sival_int -123 ?1:0, -info.si_code);
}
}
```

```
//-------------------------------------------------------
void handler( int signo, siginfo_t *info_handler, void *extra)
/* Обработчик сигналов SIGRTMIN от таймеров, запущенных потоком GRANDCHILD, и сигнала
   SIGINT  с  терминала */
   {
     if (signo == SIGINT)       // SIGINT=2
     {
         printf("SIGNALHANDLER: Received signal SIGINT (Ctrl-C).
                  Program interrupted by user.\n");
         _exit();
     }
     else
     {/* Использование функции printf() в обработчике сигнала не рекомендуется, т.к.обработчик
         блокирует все программные потоки в системе, ожидая завершения её работы, которое может
         быть сравнительно длительным. Поэтому вместо выполнения следующей закомментирован-
         ной строки отправим требуемые для вывода на экран значения с помощью одного из специ-
         альных сигналов Neutrino в функцию main(), которая и выведет их в рамках установленной
         дисциплины диспетчеризации. Использование специального сигнала позволяет ставить сигна-
         лы в очередь и обработать последовательно все из них, не потеряв ни одного. */
       //printf("Sighandler: Received signal %d,with code= %d,value= %d.",
             signo,info_handler->si_code, info_handler->si_value.sival_int);
         if ( info_handler->si_value.sival_int == 123)
         {
             TimerInfo(getpid(), timer_id, 0, &info);
             //printf("Timer %d overruns %d\n", timer_id, info.overruns);
             SignalKill( 0,  getpid(), 1, SIGRTMAX+8, -info.overruns,
```

```
                    info_handler->si_value.sival_int);  // Timer 0
        }
        else
        {
            TimerInfo(getpid(), timer_id2, 0, &info2);
            //printf("Timer %d overruns %d\n", timer_id2, info2.overruns);
            SignalKill( 0,  getpid(), 1, SIGRTMAX+8, - info2.overruns,
                    info_handler->si_value.sival_int);  // Timer 1
        }
    }
}
```

3.4. ПРОГРАММА, РЕАЛИЗУЮЩАЯ АБСОЛЮТНЫЙ ПЕРИОДИЧЕСКИЙ ТАЙМЕР

Время срабатывания задаётся как абсолютное, отстоящее на 2 минуты вперёд от момента запуска таймера

```
#include <sys/neutrino.h>
#include <inttypes.h>
int main( void )
{
    time_t            time_of_day, time_of_expiration;
    struct tm         now, future;
    struct _itimer    itime;
    char              buffer[ 80 ];
    struct sigevent   event;
    timer_t           id;
    sigset_t          set;
    int               sig;
    struct sigaction  act;
    extern void       Alarm_Handler();

    time_of_day = time( NULL );          // Определим текущее календарное время
    now = *localtime(&time_of_day);      //по системным часам и выведем его
    strftime( buffer, 80, "Now is %R %S %A %B %d, %Y", &now);
    printf( "%s\n", buffer );
    future = *localtime(&time_of_day);   // Определим текущее местное время
    future.tm_min = now.tm_min + 2;      /* Зададим время первого срабатывания на 2
```

```
                                                    минуты позже текущего и выведем его*/
    strftime( buffer, 80, "Expiration time is %R %S %A %B %d,%Y",&future);
    printf( "%s\n", buffer );
    time_of_expiration = mktime(&future);
    itime.nsec = (uint64_t) time_of_expiration*1000000UL*1000;
    itime.interval_nsec = (uint64_t) 1000*1000*1000;   /* Интервал периодических
                                                              срабатываний таймера 1 сек */
    SIGEV_SIGNAL_INIT( &event,SIGALRM );// Зададим схему уведомления-сигнал SIGALRM
    sigfillset(&set);                          // Заполним множество сигналов всеми сигналами
    sigdelset( &set, SIGINT );/* Удалим из множества сигнал SIGINT чтобы иметь
                                 возможность прекратить работу программы комбинацией клавиш "Ctrl-C" */
    act.sa_handler = &Alarm_Handler;  // Зададим адрес функции-обработчика сигналов
    act.sa_mask = set;      // Включим множество сигналов в сигнальную маску
    act.sa_flags = 0;
    SignalAction( 0,__signalstub, SIGALRM, &act, NULL );/* Зададим действия по
                                                              получению сигналов */
    id = TimerCreate( CLOCK_REALTIME, &event );   // Создадим и запустим таймер
    TimerSettime( id, TIMER_ABSTIME, &itime, NULL );
    while(1){
        sigwait(&set, &sig);   // Ожидаем сигнал. По его получении разблокируемся
        printf("Timer\n");     // выводим на экран строку
    }
}
void Alarm_Handler(int signo){} /* Обработчик сигнала. Здесь от него не требуется никаких
                                   действий */
```

3.5. ПРОГРАММА ПОЛУЧЕНИЯ ИНФОРМАЦИИ О ТАЙМЕРАХ ПРОЦЕССОВ

/* На основе [43]. 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_my_timer2.out procfs_my_timer2.c

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

static  void    dump_procfs_timer (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 соответствующего процесса
            pid = atoi (dirent -> d_name);
            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 {        // 1) set up structure
        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), // Получает имя
                                                               // процесса
    dump_procfs_timer (fd, pid, name.info.path);
    close (fd);
}
//-------------------------------------------------------------------------------------------
#define MAX_TIMERS        512

static void
dump_procfs_timer (int fd, int pid, char *name)
{// Получает информацию отаймерах заданного процесса и выводит её
    procfs_timer   timers [MAX_TIMERS];
    int            ntimers;
    int            i;

    // fetch information about the timers for this pid
    if (devctl (fd, DCMD_PROC_TIMERS, timers, sizeof (timers), &ntimers)
        != EOK)
    {
```

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

```
    fprintf (stderr, "couldn't get TIMERS information for process %d,
             errno %d (%s)\n", pid, errno, strerror (errno));
    exit (EXIT_FAILURE);
}

if (ntimers > MAX_TIMERS) {
    fprintf (stderr, "process %d has more than %d timers (%d in fact)
             !!! ***\n", pid, MAX_TIMERS, ntimers);
    exit (EXIT_FAILURE);
}

if (ntimers > 0)
{
    printf ("\nPROCESS ID %d  NAME %s\n", pid, name);
    printf ("Info from DCMD_PROC_TIMERS\n");
    for (i = 0; i < ntimers; i++) {/*вывод членов структуры irqs тех процессов, в
                                     которых созданы таймеры */
        printf ("\tBuffer %3d timer ID %d\n", i, timers [i].id);
        printf ("\t\titime %10lld.%09lld s, %9lld.%09lld interval s\n",
                timers [i].info.itime.nsec / 1000000000LL,
                timers [i].info.itime.nsec % 1000000000LL,
                timers [i].info.itime.interval_nsec / 1000000000LL,
                timers [i].info.itime.interval_nsec % 1000000000LL);
```

```
            printf ("\t\totime %10lld.%09lld s, %9lld.%09lld interval s\n",
                    timers [i].info.otime.nsec / 1000000000LL,
                    timers [i].info.otime.nsec % 1000000000LL,
                    timers [i].info.otime.interval_nsec / 1000000000LL,
                    timers [i].info.otime.interval_nsec % 1000000000LL);
            printf ("\t\tflags 0x%08X\n", timers [i].info.flags);
            printf ("\t\ttid   %d\n", timers [i].info.tid);
            printf ("\t\tnotify %d\n", timers [i].info.notify);
            printf ("\t\tclockid %d\n", timers [i].info.clockid);
            printf ("\t\toverruns %d\n", timers [i].info.overruns);
            dump_sigevent (&timers [i].info.event);
        }
        printf ("\n");
    }
}
//-------------------------------------------------------------------------------------------------
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;
    }
}
```

Тестовая программа

```
#include <stdio.h>
#include <signal.h>
#include <sys/time.h>
//-----------------------------------------------------------------
void handler( signo )
  {
  }
//-----------------------------------------------------------------
int main( void )
{
    sigset_t set, set2;
    struct itimerval value = {{1, 500000}, {1, 500000}};

    sigemptyset( &set );
    signal(SIGALRM, handler);
    setitimer ( ITIMER_REAL, &value, NULL );
    ualarm(5000000, 3200000); //prosfs doesn't contain any data about
    alarm(), ularm(). Only setitimer() and timer_create()
    while(1)
    {
```

```
        sigsuspend( &set );
        printf("*\n");
    }
}
```

Фрагменты вывода программы (кроме интервала таймера важной информацией является вид события о стабатывании таймера и его приоритет (в случае импульса)):

PROCESS ID 1 NAME /home/builder/daily/x86/boot/sys/procnto **(менеджер процессов)**
Info from DCMD_PROC_TIMERS
Buffer 0 timer ID 0
itime 1140265872.000000000 s, 0.000000000 interval s
otime 292.712768478 s, 0.000000000 interval s
flags 0x0000000B
tid 0
notify 65540
clockid 0
overruns 0
event (sigev_notify type 4)
SIGEV_PULSE (sigev_coid 0x40000000, sigev_value.sival_int 0, sigev_priority 63, sigev_code 0)

PROCESS ID 6 NAME proc/boot/devb-eide **(драйвер жёстких дисков)**
Info from DCMD_PROC_TIMERS
Buffer 0 timer ID 0
itime 8096.291230910 s, 1.000000000 interval s
otime 0.000000000 s, 0.000000000 interval s
flags 0x00000000

tid 0
notify 65540
clockid 0
overruns 0
event (sigev_notify type 4)
SIGEV_PULSE (sigev_coid 0x40000002, sigev_value.sival_int 0, sigev_priority 21, sigev_code 3)
Buffer 1 timer ID 1
itime 144.801994928 s, 1.000000000 interval s
otime 0.000000000 s, 0.000000000 interval s
flags 0x00000000
tid 0
notify 131076
clockid 0
overruns 0
event (sigev_notify type 4)
SIGEV_PULSE (sigev_coid 0x40000005, sigev_value.sival_int 0, sigev_priority 21, sigev_code 3)
Buffer 2 timer ID 2
itime 8095.505236418 s, 0.250000000 interval s
otime 0.210006120 s, 0.250000000 interval s
flags 0x00000001
tid 0
notify 196612
clockid 0
overruns 0
event (sigev_notify type 4)
SIGEV_PULSE (sigev_coid 0x40000008, sigev_value.sival_int 0, sigev_priority -1, sigev_code 1)

Buffer 3 timer ID 3
itime 6.380406207 s, 0.500000000 interval s
otime 0.000000000 s, 0.000000000 interval s
flags 0x00000004
tid 0
notify 262148
clockid 0
overruns 0
event (sigev_notify type 4)
SIGEV_PULSE (sigev_coid 0x40000008, sigev_value.sival_int 0, sigev_priority -1, sigev_code 2)

PROCESS ID 77839 NAME sbin/devc-ser8250 **(драйвер последовательного порта)**
Info from DCMD_PROC_TIMERS
Buffer 0 timer ID 0
itime 30.372421169 s, 0.050000000 interval s
otime 0.000000000 s, 0.000000000 interval s
flags 0x00000004
tid 0
notify 65540
clockid 0
overruns 0
event (sigev_notify type 4)
SIGEV_PULSE (sigev_coid 0x40000002, sigev_value.sival_int 0, sigev_priority 24, sigev_code 1)

PROCESS ID 77842 NAME sbin/io-net **(драйвер сетевой карты)**

```
Info from DCMD_PROC_TIMERS
        Buffer   0 timer ID 0
                itime        132.626067916 s,       120.000000000 interval s
                otime          0.000000000 s,         0.000000000 interval s
                flags 0x00000000
                tid   0
                notify 65538
                clockid 0
                overruns 0
                event (sigev_notify type 2)
                SIGEV_SIGNAL_CODE (sigev_signo 16, sigev_value.sival_int 23257, sigev_code -96)
        Buffer   1 timer ID 1
                itime       8095.335334289 s,         0.050000000 interval s
                otime          0.032105215 s,         0.050000000 interval s
                flags 0x00000005
                tid   0
                notify 131076
                clockid 0
                overruns 0
                event (sigev_notify type 4)
               SIGEV_PULSE (sigev_coid 0x40000009, sigev_value.sival_int 14, sigev_priority 20, sigev_code 1)
        Buffer   2 timer ID 2
                itime       8095.907330923 s,         3.000000000 interval s
                otime          0.604101849 s,         3.000000000 interval s
                flags 0x00000005
                tid   0
```

notify 196612
clockid 0
overruns 0
event (sigev_notify type 4)
SIGEV_PULSE (sigev_coid 0x40000014, sigev_value.sival_int 0, sigev_priority 21, sigev_code 2)

PROCESS ID 2932777 NAME ./setitimer.out **(пользовательский процесс)**
Info from DCMD_PROC_TIMERS
Buffer 0 timer ID 0
itime 8095.593102857 s, 1.500000000 interval s
otime 0.284874548 s, 1.500000000 interval s
flags 0x00000005
tid 0
notify 65538
clockid 0
overruns 0
event (sigev_notify type 1)
SIGEV_SIGNAL (sigev_signo 14, sigev_value.sival_int 0) (сигнал **SIGALRM**)

3.6. ПРОГРАММА, ИЛЛЮСТРИРУЮЩАЯ ИСПОЛЬЗОВАНИЕ ТАЙМАУТОВ ЯДРА

В зависимости от условий использования ядро отрабатывает таймаут либо на ожидание завершения дочернего потока, либо на ожидание аппаратного прерывания от устройства /dev/ser2. Запускать на пару с программой, посылающей байт на /dev/ser2 с соседнего PC на скорости 57600.

```
#include <stdio.h>
#include <sys/neutrino.h>
#include <stdlib.h>
#include <hw/inout.h>
#include <errno.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                // IIR (read),  FIFO control register (write)
#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

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;       // Принимаемый символ
int                i = 1;  // Вспомогательные переменные для расчёта среднего временнОго
_uint64            Sum = 0LL;  // интервала между получаемыми символами
//-------------------------------------------------------------------------------------------------
const struct sigevent *handler( void *arg, int id )
// Обработчик прерываний
{
    int  iir;           // Значение регистра идентификации прерываний
    struct sigevent *event = (struct sigevent *)arg;
    serial_rx = in8 (base_reg + REG_RX); // Прочесть  полученный байт
    return (event);     // Разбудить поток-обработчик, возвратив событие
}
//-------------------------------------------------------------------------------------------------
// Функция потока, реагирующего на прерывания
void *int_thread (void *arg)
{
    struct sigevent    int_wait_unblock_event;
    struct timespec    timeout, otime, start, finish;
    int                errvalue;
```

```
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
out8 (base_reg + LCR, 0x80);
out8(base_reg+REG_RX, 0x02);  // скорости передачи  57600 бит/с
out8(base_reg+IER, 0x00);
out8 (base_reg + LCR, 0x03);/*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);
}

int_wait_unblock_event.sigev_notify = SIGEV_UNBLOCK;/* генерируемое
                                                       при истечении таймаута событие */
while(1)
```

```
{
    errno = EOK;
    timeout.tv_sec  = 5;   // Параметры таймаута (5 или 10 секунд)
    timeout.tv_nsec =  0;
    clock_gettime( CLOCK_REALTIME, &start);


    // Задаём таймаут на ожидание аппаратного прерывания
    timer_timeout( CLOCK_REALTIME, _NTO_TIMEOUT_INTR,
                   &int_wait_unblock_event, &timeout, &otime);
    InterruptWait( 0, NULL ); /* Ждать прерывания от /dev/ser2
    errvalue = errno;
    printf( "INTERRUPTWAIT - The error generated was %d  ",
             errvalue );
    printf( "That means: %s\n", strerror( errvalue ) );
    if (errvalue == ETIMEDOUT)
        {    // Истёк таймаут на ожидание аппаратного прерывания
                printf("Timeout on interrupt waiting expires\n");
                InterruptDetach(interruptID);// Disconnect the ISR handler
                pthread_exit(&i);
        }
    else
        {    // Принят байт устройством /dev/ser2
                S = serial_rx;
                if (serial_rx == 'q') pthread_exit(&i);
                printf("Received symbol is %c ", S);
```

```
                clock_gettime( CLOCK_REALTIME, &finish);
                Sum = timespec2nsec(&finish) -
                                        timespec2nsec(&start) + Sum;
            printf(" i=%d  sum=%lld\n", i, Sum);
                i++;
            }
    }
}
//-------------------------------------------------------------------------------------------------
int main()
{
    pthread_t          threadID;
    struct sigevent    join_unblock_event;
    struct timespec    timeout2, otime2;
    int                rval;
    int                errvalue2;

    system("slay devc-ser8250");
    // генерируемое при истечении таймаута событие */
    join_unblock_event.sigev_notify = SIGEV_UNBLOCK;
  //SIGEV_UNBLOCK_INIT( &join_unblock_event );  // Можно и так
    // Создаём присоединённый (по умолчанию) дочерний поток
    pthread_create (&threadID, NULL, int_thread, NULL);
    printf("The child %d created\n", threadID);
    timeout2.tv_sec  = 7; // Параметры таймаута 7 секунд
    timeout2.tv_nsec =  0;
```

```
    printf("Joining child %d\n", threadID);
    errno = EOK;
    // Задаём таймаут на ожидание завершения дочернего потока
    timer_timeout( CLOCK_REALTIME, _NTO_TIMEOUT_JOIN ,
                   &join_unblock_event, &timeout2, &otime2);
    rval = pthread_join(threadID, NULL);// Ждём завершения дочернего потока
    errvalue2 = errno;
    printf( "JOIN - The error generated was %d ", errvalue2 );
    printf( "That means: %s\n", strerror( errvalue2 ) );
    if (rval == ETIMEDOUT)
           printf("Timeout on thread joining expired.");
    else printf("Child thread terminated before join timeout
                 expired.");
    printf(" It was %i interrupts with med interval %f ms\n", i-1,
             (double) Sum/1000000/(i-1));// Итоги приёма символов
    pthread_cancel(threadID);
}
```

3.7. ПРОГРАММА TIME_INTERVAL_MEASURE_3.C, ИЛЛЮСТРИРУЮЩАЯ ПРИМЕНЕНИЕ СРЕДСТВ ИЗМЕРЕНИЯ ВРЕМЕННЫХ ИНТЕРВАЛОВ

```
#include <pthread.h>
#include <sys/neutrino.h>
#include <errno.h>
#include <sys/times.h>
#include <sys/syspage.h>
#include <inttypes.h>
#define BILLION  1000000000L;          //Для функции clock_gettime()
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void* function2( void* arg )           //Эту функцию выполняет дочерний поток 2
{
    struct timespec   timeout;
    int               rval;
    clock_t           start_time, end_time;            //Для функции clock()
    time_t            strt_tm, end_tm;                 //Для функции time()
    uint64_t          cps, cycle1, cycle2, ncycles;    //Для функции ClockCycles()
    _uint64           first, second, diff, diff_nsec; //Для функции ClockTime()
    struct timespec   diff_ClockTime;
    float             sec;
    struct timespec   start, stop;                /     /Для функции clock_gettime()
    double            accum;
    struct tms        childtim;                        //Для функции times()
    rval = ETIMEDOUT;
```

```
start_time = clock();
strt_tm = time( NULL );
cycle1 = ClockCycles( );
clock_gettime( CLOCK_REALTIME, &start);
ClockTime( CLOCK_REALTIME, NULL, &first );
while (rval == ETIMEDOUT)  //Опрос состояния мутекса и его захват при первой возможности
 {
    timer_timeout( CLOCK_REALTIME, _NTO_TIMEOUT_MUTEX , NULL, NULL,
                   NULL);
    rval = pthread_mutex_lock( &mutex );
 }
end_time = clock();
end_tm = time( NULL );
cycle2 = ClockCycles( );
clock_gettime( CLOCK_REALTIME, &stop);
ClockTime( CLOCK_REALTIME, NULL, &second );
printf( "Mutex locked. Elapsed time measured with clock-function was %lu
        seconds\n",(long) ((end_time - start_time) / CLOCKS_PER_SEC));
printf( "Mutex locked. Elapsed time measured with difftime-function:
        %f seconds\n", difftime( end_tm, strt_tm ));
ncycles = cycle2 - cycle1;
cps = SYSPAGE_ENTRY(qtime)->cycles_per_sec;
sec=(float)ncycles/cps;
printf("Mutex locked. Elapsed time measured with ClockCycles-function
       is %f \n",sec);
accum = ( stop.tv_sec - start.tv_sec ) +
```

```
            (double)( stop.tv_nsec - start.tv_nsec ) / (double)BILLION;
    printf("Mutex locked. Elapsed time measured with clock_gettime-
           function:%lf\n", accum );
    diff = second - first;
    nsec2timespec( & diff_ClockTime, diff );
    accum = diff_ClockTime.tv_sec  +
           (double) diff_ClockTime.tv_nsec / (double)BILLION;
    printf( "Mutex locked. Elapsed time measured with ClockTime-function:
             %lf\n", accum );

    times( &childtim );
    printf( "Mutex locked. times-function results: System time = %f s ",
           (float) childtim.tms_stime/CLK_TCK );
    printf( "user time   = %f s\n", (float) childtim.tms_utime/CLK_TCK );
    pthread_mutex_unlock( &mutex );//Разблокировать мутекс
    sleep( 2 ); // После разблокирования мутекса жить во сне ещё ≈2 секунды
    pthread_cancel(pthread_self());
}
//-------------------------------------------------------------------------------------------------------
void* function1( void* arg )          //Эту функцию выполняет дочерний поток 1
{
    struct timespec nanosleep_time;
    nanosleep_time.tv_sec = 2;
    nanosleep_time.tv_nsec = 500000000;
    pthread_mutex_lock( &mutex ); // Захватить мутекс на ≈2.5 секунды
    nanosleep(&nanosleep_time, NULL);
```

```
    pthread_mutex_unlock( &mutex ); //Разблокировать мутекс
    sleep( 2 );                     /* После разблокирования мутекса перейти в режим сна
                                       приблизительно на 2 секунды*/
    pthread_cancel(pthread_self());
}
//-------------------------------------------------------------------------------------------------------
int main( void )
{
    pthread_create( NULL, NULL, &function1, NULL );//Создаём 2 дочерних потока
    pthread_create( NULL, NULL, &function2, NULL );
    sleep( 10 );    //Заснуть на 10 секунд пока работают дочерние потоки
    return 0;
}
```

3.8. ИСПОЛЬЗОВАНИЕ ПРЕРЫВАНИЙ ОТ ПОСЛЕДОВАТЕЛЬНОГО ПОРТА /dev/ser1 КАК ИСТОЧНИКА ПЕРИОДИЧЕСКИХ ИМПУЛЬСОВ

/* Программа com1_timer.c. Использует периодические прерывания по отправке символов
 * последовательного порта /dev/ser1 в качестве интервального таймера. Период срабатывания таймера
 * может регулироваться заданием: а) скорости передачи символа; б) порога генерирования
 * прерывания по количеству отправленных/принятых символов (interrupt trigger level);
 * в) включением/отключением буфера FIFO. Выходная линия асинхронного адаптера (АА) соединена
 * входной аппаратной заглушкой. Необходимые сведения об асинхронном адаптере содержатся в
 * Приложении 1 части 2 настоящего учебного пособия. Уведомлением о прерывании является сигнал в
 * стиле POSIX 1003.1.*/


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

#define REG_RX              0          /* Адрес регистра-буфера приёма/передачи
                                          относительно базового адреса АА  /dev/ser1 */
#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 IIR_THE             0x02       /* Значение регистра IIR, идентифицирующее
                                           прерывание по отправке символа */
#define HW_SERIAL_IRQ       4          // IRQ  асинхронного адаптера /dev/ser1
```

```
struct sigevent event;      /* Структура, описывающая генерируемое  при возникновении
                               прерывания событие */


static int base_reg = 0x3f8;            // Базовый адрес  /dev/ser1
int interrupt_trigger_level = 14;       // Возможные значения : 1;  4;  8;  14
unsigned short int FIFO;     /* Содержимое регистра FCR, задающее: а) значение interrupt
trigger level; б) включение/выключение буферов FIFO и очищающее эти буфера*/

    struct sigaction sa;  // Структура, задающая обработчики и параметры обработки сигнала
    sigset_t signal_set;   //  Множество блокированных (замаскированных) сигналов
//_____________________________________________________________________________
// Обработчик сигнала. В данной программе не требуется от него к.-л. действий
void signal_handler(int sig, siginfo_t* info, void* other)
{
}
//_____________________________________________________________________________
const struct sigevent *handler( void *area, int id ) {
// Обработчик прерываний
    int  i, iir;   // iir - Значение регистра идентификации прерываний

    iir = in8 (base_reg + IIR) & IIR_MASK;/* Считать биты, характеризующие
                                             источник  прерываний и очистить регистр IIR */
```

```
    switch (iir) {          // Определить причину прерывания

    case   IIR_THE : //  прерывание вызвано отправкой байта
    out8 (base_reg + IIR, FIFO);  /* установить значение interrupt trigger level, задать
                                     режим использования буферов FIFO и очистить эти буфера*/
    // Заполнить буфер передачи:
    for(i=0; i <  interrupt_trigger_level; i++) out8(base_reg, 0xFF);
    return (&event);   // Возвратить событие
    break;
    }
}
//_____________________________________________________________________________
int main()
{
    int i, type;
    int id;

    system("slay devc-ser8250");  // Убить конкурента в обладании /dev/ser1
    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 АА  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, 0x00);  /* Установить младший байт  делителя частоты для
                                     скорости передачи  50 бит/с */
    out8(base_reg+IER, 0x09);          /* Установить старший байт делителя частоты для
                                     скорости передачи  50 бит/с */

    out8 (base_reg + LCR, 0x03);        /* Обнулить DLAB и установить параметры
                коммуникации: нет проверки  на чётность, 1 стоповый бит, длина слова 8 бит */
    /* В зависимости от значения interrupt trigger level и необходимости использования буферов FIFO
      выбрать значение переменной FIFO  и занести его в регистр FCR (тоже самое, что IIR) :  */
     switch (interrupt_trigger_level) {   /* Проверьте, как влияет выключение буфера
                                                на период прерываний *.
        case 1: FIFO = /*0x07*/ 0x06; break;     /* FIFO enabled */
        case 4: FIFO = /*0x47*/ 0x46; break;     /* FIFO enabled */
        case 8: FIFO = 0x87 /*0x86*/; break;     /* FIFO disabled */
        case 14: FIFO = 0xC7 /* 0xC6*/;          /* FIFO disabled */
    }
     out8 (base_reg + IIR, FIFO);


    // Определить тип UART :
    type = in8 (base_reg + IIR) & 0xC0;
    switch (type) {
      case 0x00: printf("UART type is 16450 or earlier\n"); break;
```

```
  case 0x80: printf("UART type is 16550\n"); break;
  case 0xC0: printf("UART type is 16550a or later (16650 or
                     16750)\n"); break;
}

 out8(base_reg+IER, 0x02);          // Разрешить прерывания по посылке байта

sigemptyset(&sa.sa_mask);                 // Очистить сигнальную маску процесса
sa.sa_sigaction = signal_handler;         /* Записать в структуру sa адрес обработчика
                                          сигнала */
    // Задать действие по сигналу SIGUSR1 :
if (sigaction(SIGUSR1 , &sa, NULL) < 0) {
    perror("sigaction");
    exit(2);
}
sigemptyset(&signal_set);        /* Очистить множество заблокированных сигналов
                                          процесса */

// Подключиться к источнику прерываний :
id=InterruptAttach( HW_SERIAL_IRQ, &handler, NULL, 0, 0 );

SIGEV_SIGNAL_INIT(&event, SIGUSR1);  // Инициализировать структуру события
/* Альтернативный вариант:
       event.sigev_notify = SIGEV_SIGNAL;
       event.sigev_signo = SIGUSR1;*/
```

```
    for(; ; ) {
        // Ждать сигнал:
         sigwait(&signal_set, NULL);   // POSIX 1003.1 (Realtime Extensions)
        /* Вариант : sigsuspend(&set);    // POSIX 1003.1 */
        printf( "interrupt\n" );
    }
}
```

3.9. ПРОГРАММА “КАЛИБРОВКИ” ДЛИТЕЛЬНОСТИ МАКРОСА `ClockCycles()` И ОПРЕДЕЛЕНИЯ ВРЕМЕНИ, ПОТРАЧЕННОГО ПРОЦЕССОРОМ НА ВЫПОЛНЕНИЕ ПРОЦЕССА И ПОТОКА

/* Полужирным начертанием выделены фрагменты программы, определяющие количество тактов процессора на выполнение ClockCycles() (“калибровка” ClockCycles())*/

```
#include <sys/neutrino.h>
#include <inttypes.h>
#include <stdlib.h>
#include <sys/procfs.h>
#include <fcntl.h>
#include <math.h>
#define BILLION 1000000000L   // количество наносекунд в секунде

int main( void )
{
    uint64_t cps, cycle1, cycle2, ncycles; /* Значения регистра TSC процессора до и
                                              после замеряемого действия и разность между ними*/
    double cpnsec;  // длительность 1 такта процессора в нс
    int i;          // Параметр цикла

    int fd;         // Файловый дескриптор
    char buf[512];  /* Буфер для хранения пути к адресному пространству текущего процесса в
                       виртуальной файловой системе /proc менеджера процессов */
    procfs_info *info = NULL;  /*Структура, содержащая информацию о процессе. См.
```

определение типа debug_process_t в заголовочном файле <sys/debug.h>. prosfs_info – псевдоним структуры debug_process_t, см. <sys/debug.h> */
procfs_status *status = NULL; /*Структура, содержащая информацию о потоке. См. определение типа debug_thread_t в заголовочном файле <sys/debug.h>. procfs_status- псевдоним структуры debug_thread_t, см. <sys/debug.h> */
pid_t pid; //Идентификатор текущего процесса
_Uint64t thread_time1; // Время использования процессора потоком в нс
uint64_t system_clock_time, system_clock_time_prev, delta; /* Время по системным часам на текущем и предыдущем шагах цикла и разность между ними*/
struct _clockperiod res; /* Разрешающая способность системных часов = период хода системных часов = системный тик */
pid = getpid();
printf("\nprocess with pid %d starts ",pid);
snprintf(buf, 511, "/proc/%d/as", pid); /* Путь к адресному пространству “as” (adres space) текущего процесса */
fd = open (buf, O_RDONLY); /* Открываем для чтения виртуальный файл – адресное пространство процесса */
info = (procfs_info *) &buf;
devctl (fd, DCMD_PROC_INFO, info, sizeof (* info), 0); /* Считываем информацию о процессе и размещаем её в структуре info. Команда DCMD_PROC_INFO описана в <sys/procfs.h> */
printf("process start time = %lld ns\n", info->start_time);
cps = SYSPAGE_ENTRY(qtime)->cycles_per_sec; // Циклов процессора в 1 сек
cpnsec = cps/(double) BILLION;
printf("This system has %lld cycles/sec = %f cycles/nsec.\n",cps,

```
          cpnsec);
ClockPeriod(CLOCK_REALTIME, NULL, &res, 0); // Разрешение системных часов
printf("System Clock resolution(System Clock period) is %ld nsec\n\n",
        res);
for (i=0; i<6; i++)                    // Повторяем 6 раз
{
    cycle1=ClockCycles( );             // засечка1 TSC
    cycle2=ClockCycles( );             // засечка2 TSC
    ncycles=cycle2-cycle1;             //Разность засечек
    printf("%lld cycles = %.3f nsec  elapsed\n", ncycles,
           rint((double)ncycles / cpnsec));/*

    devctl (fd, DCMD_PROC_INFO, info, sizeof (* info), 0); /* Считываем
                                                текущую информацию о процессе */
    printf("process utime: %lld ns = %.3f tick  stime: %lld ns =
           %.3f tick\n",
           info->utime,// время использования процессора пользовательском режиме в нс
           rint((double) (info->utime)/(res.nsec)),// и в тиках
           info->stime, // время использования процессора в режиме ядра в нс
           rint((double)(info->stime)/(res.nsec))); // и в тиках

    status = (procfs_status *) buf;
status->tid = 1;    // Нас интересует информация о потоке №1 процесса
    devctl (fd, DCMD_PROC_TIDSTATUS, status, sizeof (*status), 0); );
    /* Считываем  информацию  о потоке и размещаем её в структуре  status. Команда
```

```
      DCMD_PROC_ TIDSTATUS описана в <sys/procfs.h> */
    thread_time1 = status->sutime;
    ClockTime(CLOCK_REALTIME, NULL, &system_clock_time);/* Время по
                                                        системным часам нс */
    printf("Sytem clock time is %lld ns  ", system_clock_time);
    delta = system_clock_time - system_clock_time_prev;
    if(i) printf("delta :  %lld ns = %.3f tick\n", delta,
                  (float) delta/res.nsec); else printf("\n");
    system_clock_time_prev = system_clock_time;
    printf ("process %d  thread %d state %X start_time=%lld ns  sutime:
            %lld ns = %.3f tick\n\n",
             status->pid,     // pid процесса
             status->tid,     // tid потока
             (int)status->state,  // состояние потока (см. <sys/states.h>)
             status->start_time,  // время запуска потока
             status->sutime, /* суммарное время использования процессора потоком в
                                режиме пользователя и режиме ядра в нс */
             rint((double) (status->sutime)/(res.nsec)) );/* и в тиках
   Т.к. в этом процессе 1 поток, то info->utime+info->stime равно status->sutime*/
  }
  close(fd);
  return EXIT_SUCCESS;
}
```

3.10. ПРОГРАММА ДЛЯ ЗАМЕРА И ЗАПИСИ В ФАЙЛ ДЛИТЕЛЬНОСТИ ИНТЕРВАЛОВ ПЕРИОДИЧЕСКОГО ТАЙМЕРА

```
/*  Данные из файла подвергаются последующему статистическому анализу */
#include <stdio.h>
#include <inttypes.h>
#include <sys/neutrino.h>
#include <stdlib.h>
#include <sys/netmgr.h>
int main(int argc, char **argv){
    int i;
    volatile uint32_t Sample[4101]={};  // Массив отсчётов TSC процессора
    FILE *fp_out;
    char buffer[21*4101]="";   // буфер, из которого замеренные значения интервалов
                               // записываются в файл
    char buf_aux[21];

   struct sigevent         event;
   struct itimerspec       itime;
   timer_t                 timer_id;
   int                     chid, rcvid;
   struct _pulse msg;
   setprio(0, 63);
   chid = ChannelCreate(0);
   event.sigev_notify = SIGEV_PULSE; // Уведомление импульсом
   event.sigev_coid = ConnectAttach(ND_LOCAL_NODE, 0, chid,
```

```
                                        _NTO_SIDE_CHANNEL, 0);
event.sigev_priority = getprio(0);
timer_create(CLOCK_REALTIME, &event, &timer_id);

itime.it_value.tv_sec = 0;
 itime.it_value.tv_nsec = 999847*5; // точно 5 системных тиков
 itime.it_interval.tv_sec = 0;
itime.it_interval.tv_nsec = 999847*5;

 timer_settime(timer_id, 0, &itime, NULL);
  for (i=0; i < 4101; i++)
  {
     MsgReceivePulse(chid, &msg, sizeof(msg), NULL);
     Sample[i] = ClockCycles();
  }
 for (i=0; i < 4101-1; i++) Sample[i] = Sample[i+1] - Sample[i];
 for (i=0; i<4101-1; i++)
 {
     ultoa( Sample[i], buf_aux, 10);
     strcat(buf_aux, "\n");
     strcat(buffer, buf_aux);
 }
 fp_out = fopen( "outfil64.txt", "w" );// Запись интервалов в файл outfil64.txt
 fputs( buffer, fp_out );
 fclose( fp_out );
exit;}
```

3.11. ПРОГРАММА ЗАМЕРА ДВУМЯ СПОСОБАМИ СЛУЧАЙНОГО ИНТЕРВАЛА СРАБАТЫВАНИЯ ОДНОКРАТНОГО ТАЙМЕРАС ЗАПИСЬЮ РЕЗУЛЬТАТОВ В ФАЙЛ

```
/*  Данные из файла подвергаются последующему статистическому анализу */
#include <stdio.h>
#include <stdlib.h>
#include <sys/syspage.h>
#include <inttypes.h>
#include <sys/neutrino.h>
#define BILLION  1000000000L;
int main()
{
    struct timespec delay;
    uint64_t cps, cycle1, cycle2, ncycles,  nsec, start, stop;
    struct timespec res;
    float delay_nsec;
    float accum, aux2;

    delay.tv_sec = 0;
    srand( (int) time( NULL ) );
    // Константа RAND_MAX определена в  < stdlib.h >:#define RAND_MAX 32767u
    delay.tv_nsec = 250000000 + (int) (rand() - (int) RAND_MAX) * 1000;
    ClockTime(CLOCK_REALTIME, NULL, &start);
    cycle1=ClockCycles( );
    nanosleep( &delay, NULL);
    cycle2=ClockCycles( );
    ClockTime(CLOCK_REALTIME, NULL, &stop);
```

```
    ncycles=cycle2-cycle1;
    cps = SYSPAGE_ENTRY(qtime)->cycles_per_sec;
    clock_getres( CLOCK_REALTIME, &res);
    delay_nsec = (float) ncycles/cps;
    accum = (float) (stop - start) / BILLION;
    aux2 = (float) delay.tv_nsec / BILLION;
    printf("   delay.tv_nsec = %.4f s delay = %.6f s  accum = %.6f s\n",
    aux2, delay_nsec, accum);
    exit( EXIT_SUCCESS);
}
```

3.12. ЗАГОЛОВОЧНЫЙ ФАЙЛ ДЛЯ ИСПОЛЬЗОВАНИЯ БИБЛИОТЕКИ

paired_stat.lib


```
#include <stdio.h>
#include <stdlib.h>
#include <inttypes.h>
#include <math.h>
#include <errno.h>
#include <ncurses.h>
#include <unistd.h>
```

// Сравнивает op1 и op2 типа double и возвращает: 1,если op1>op2, 0,если op1=op2 и -1,если op1<op2.

```
int compare( const void* op1, const void* op2 );
```

/* Сравнивает |op1| и |op2| типа double и возвращает: 1,если |op1|>|op2|, 0,если |op1|=|op2| и -1,если |op1|<|op2| */

```
int compare2( const void* op1, const void* op2 );
```

/*Ранжирование элементов 1…n массива w длиной n+1 после удаления нулевых элементов с подсчётом суммы sp и количества np положительных рангов, суммы sn и и количества nn отрицательных рангов, общей суммы рангов s=sp+|sn| и количества ненулевых элементов n_total. Для каждой группы из k совпадающих элементов рассчитывается средний ранг и корректирующая поправка $corr_k = (k^3-k)/48$, corr есть сумма всех $corr_k$. См. [75, разд 14.6, файл c14-6.pdf]

```
void crank(unsigned long n, double w[], double *s, double *sp, double *sn,
int *n_total, int *np, int *nn, double *corr);
```

/* Сравнение однородности средних и законов распределения двух парных выборок sample1 и sample2 длиной m. Для выборок большого объёма (m*(m+1)/2>20) использует нормальную аппроксимацию критических значений и выдаёт заключение об однородности выборок (двусторонний тест) на уровнях значимости 0.1%, 0.5%, 1%, 5%, 10% (Орлов, прикладная статистика, wsrt.pdf). Параметр shift зарезервирован, должен задаваться равным 0*/

```
int paired_samples_test( double sample1[], double sample2[], int m, double
shift);
```

//Поиск наибольшего при f=1 или наименьшего при f≠1 элемента массива длиной n элементов

```
double minmax(double arr[], int n, int f);
```

/* Формирование двумерного массива viborka из двух одномерных sample1 длиной m1 и sample2 длиной m2, вычисление для обеих выборок начального количества интервалов interval[2] гистограммы, мат. ожиданий и среднеквадратических отклонений mean[2] и RMS[2], наименьших и наибольших элементов min[2] и max[2], наибольшего max12, наименьшего min12 значений и размаха scope объединённой выборки */

```
void prepare(double sample1[], double sample2[], int m1, int m2,
             int sample_m[],double viborka[][max(m1,m2)],int intervalov[],
             double mean[], double RMS[], double min[], double max[],
             double *min12, double *max12, double *scope);
```

/* Для выборки - строки gn (gn=0 или 1) массива viborka длиной sample_m[gn] элементов по заданному количеству интервалов гистограммы intervalov[gn] рассчитывает длину интервала interval[gn] гистограммы, массивы абсолютных gisto_frequencies, относительных promille в %% частот и высот screen_frequencies столбцов гистограммы на экране, а также ширину столбца width на экране. Задаются массивы значений наименьших min и наибольших max элементов выборок, наибольшая высота столбца на экране `height` и общее количество позиций экрана COL, отводимое для построения гистограммы*/

```
void freq(int gn, int intervalov[], double interval[],
          double gisto_frequencies[][20], int screen_frequencies[][20],
          int width[], double min[], double max[], int sample_m[],
          double viborka[][max(sample_m[0],sample_m[1])],
          int promille[][20], int height, int COL[]);
```

/* Выводит на экран гистограммы выборок sample1 длиной m1 и sample2 длиной m2 с возможностью интерактивного изменения количества интервалов для каждой из выборок в пределах 1…20*/

```
void histogram(double sample1[], double sample2[], int m1, int m2);
```

3.13. ПРОГРАММА ПОСТРОЕНИЯ ГИСТОГРАММ ДВУХ СЛУЧАЙНЫХ ВЕЛИЧИН С ИСПОЛЬЗОВАНИЕМ ncurses

```
#include "paired_stat.h"
//-------------------------------------------------------------------------------------------------------------
void histogram(double sample1[], double sample2[], int m1, int m2)
                  //выборка 1              выборка 2              число элементов
{
    // FOR ncurses :
    WINDOW *my_win[2][20],// массив окон для размещения столбцов гистограммы
           *floor_win;    // окно для вывода текста под гистограммой
    int height;           // Наибольшая высота столбца гистограммы
    int startx=3, starty, width[2];     //Начальная позиция и ширина столбца для 1ой и
                                        // 2ой гистограммы
    int ch;               // Код нажатой клавиши
    char str[4];
    int promille[2][20]= {};// Массив для хранения относительных частот интервалов
                           //  в промилле
    char *menu_str =      // Строка-подсказка меню
         "<-- previous histogram, --> next histogram,
            arrow_up-more intervals, arrow_down-less intervals, F10 exit";
    int i, j, k;

    double viborka[2][max(m1,m2)];// Внутренний массив, хранящий обе выборки
    int sample_m[2] = {m1, m2};   // Внутренний массив, хранящий размеры выборок
```

```
double min[2], max[2],// наименьшие и наибольшие значения каждой из выборок
       min12, max12,  // наименьшее и наибольшее значения среди двух выборок
       scope;         // размах = max12-min12
double gisto_frequencies[2][20] = {};// Массив абсолютных частот интервалов

int screen_frequencies[2][20] = {};   // Массив частостей (относительных частот)
                                       // интервалов в  экранных единицах
double interval[2] = {0, 0};          // Ширина столбцов в экранных единицах
unsigned int gisto_number, gn;
unsigned int intervalov[2] = {0, 0}; // Количество интервалов каждой из
                                       // гистограмм
double h_scale;                       // Масштаб гистограммы по горизонтали
int COL[2],    // Количество позиций по горизонтали, занимаемых гистограммой на экране
    COLMIN[2]; // начальная позиция гистограммы на экране по горизонтали

// for histogram name
FILE *fp = stdin;
char gisto_name[2][79] = {}; // Вводимые названия гистограмм
double mean[2], RMS[2];      /* Математические ожидания и среднеквадратические
                                  отклонения выборок */
double s1;                   // Сумма абсолютных частот
int  s2;                     // Сумма относительных частот в промилле

                             // Предварительная обработка выборок
prepare(sample1, sample2, m1, m2,  sample_m, viborka, intervalov,
```

```
        mean,  RMS, min, max, &min12,  &max12, &scope);

for(i=0; i<2; i++)      // Вывод характеристик выборок и ввод названий гистограмм
{
    printf(" VIBORKA%d: m=%d mean=%f  std.dev=%f  min%d=%f max%d=%f
        intervalov=%d\n", i+1, sample_m[i], mean[i], RMS[i], i+1,
        min[i], i+1, max[i], intervalov[i]);
    printf("Vvedi nazvanie gistogrammi dlya VIBORKA%d:  ", i+1);
    input_line( fp, gisto_name[i], sizeof(gisto_name[i]));
}
initscr();              // Инициализация ncurses
h_scale = ((double)COLS - 6.0) / scope;
COL[0] = (int) rint((max[0] - min[0]) * h_scale);
COLMIN[0] = (int) rint((min[0] - min12) * h_scale)+startx;
COL[1] = (int) rint((max[1] - min[1]) * h_scale);
COLMIN[1] = (int) rint((min[1]-min12) * h_scale)+startx;
//endwin();             // Временный выход из режима ncurses
//refresh();            // Возврат в режим ncurses

savetty();              // Сохранить настройки терминала для последующего восстановления
start_color();          // Разрешить управления цветом
cbreak();               // Отключить буфера клавиатуры
keypad(stdscr, TRUE);   // Разрешить обработку функциональных клавиш
noecho();               // Запретить эхо-вывод вводимых символов
clear();                // Очистить экран
```

```
height = LINES - 12;
starty = LINES - 6;
init_pair(1, COLOR_WHITE, COLOR_BLACK);  // используемые цветовые пары
init_pair(2, COLOR_YELLOW, COLOR_BLACK);
init_pair(3, COLOR_BLACK, COLOR_BLACK);
init_pair(6, COLOR_RED, COLOR_BLACK);
init_pair(7, COLOR_BLACK, COLOR_CYAN);

init_pair(8, COLOR_MAGENTA, COLOR_BLACK);      // Сделать цветовую пару 8 активной
gisto_number = 0;

do                          // Цикл прорисовки гистограмм
{
    clear();
    for(gn=0; gn <2; gn++)
    {
        if(! ((gisto_number == gn) && sample_m[gn])) continue;
        /* Если выборка с номером gn не задана, перейти к другой*/
        freq(gn, intervalov, interval, gisto_frequencies,
             screen_frequencies, width, min, max, sample_m, viborka,
             promille, height, COL);  // Рассчитать высоту столбцов
        // clear();
        move(0, COLS/2-strlen(gisto_name[gn])/2);
        printw("%s\n", gisto_name[gn]);// Вывести название гистограммы
```

```
move(2, 0);
printw(" start point\t   histogram\trelative\n");
/* Вывести абсолютные и относительные частоты гистограммы*/
printw(" of interval\t   frequencies\tfrequencies %%%%\n");
for(j=0; j<45; j++) printw("="); printw("\n");
for (j=0; j < intervalov[gn]; j++)
    wprintw(stdscr, "  %f \t\t%.0f\t%d\n",
            min[gn]+interval[gn]*j, gisto_frequencies[gn][j],
            promille[gn][j]);
s1 = s2 = 0;
for(j=0; j< intervalov[gn]; j++)
{s1 += gisto_frequencies[gn][j]; s2 += promille[gn][j]; }
for(j=0; j<45; j++) printw("-"); printw("\n");
printw("  SUM\t\t\t%.0f\t%d\n", s1, s2); // Вывести суммы частот
wprintw(stdscr, "Press Delete\n");
while (getch() != KEY_DC);                  // Ждать нажатия клавиши Del
clear();

curs_set(0);        // Сделать курсор невидимым
// Задать окно и вывести меню-подсказку вверху экрана
floor_win = newwin( 1, COLS-2, starty +3, startx-2);
refresh();
COLOR_PAIR(1);
move(0, COLS/2-strlen(menu_str)/2);
printw("%s", menu_str);
COLOR_PAIR(8);
```

```
    move(3, COLS/2-strlen(gisto_name[gn])/2);
    printw("%s", gisto_name[gn]); // Вывести название гистограммы
    // Расчёт и вывод на экран столбцов гистограммы
    for (i=0; i <intervalov[gn]; i++)
    {
        my_win[gn][i] = newwin((int) screen_frequencies[gn][i]+1,
            width[gn]+1, starty - (int)screen_frequencies[gn][i]+1,
            COLMIN[gn] + i*width[gn]);// Задание окна для столбца
        box(my_win[gn][i], 0 , 0);     // рисование в окне прямоугольника
            //(0,0 символы по умолчанию для вертикальных и горизонтальных линий)
        itoa(promille[gn][i], str,10);
        strncat( str, "%%",2);
        waddnstr(my_win[gn][i], str, -1); // вывод относительной частоты для
                                                    //столбца
        // Закрашивание окна цветом фона
        wbkgd(my_win[gn][i], gn ? COLOR_PAIR(6) : COLOR_PAIR(7));
        wrefresh(my_win[gn][i]);   // Показать столбец на экране
    }
    //Вывести внизу экрана характеристики выборки и гистограммы
    wprintw(floor_win, "%f s", min[gn]);
    mvwprintw(floor_win, 0, COLS-15, "%f s", max[gn]);
    mvwprintw(floor_win, 0, COLS/2-31,
    "mean=%.4f stand.deviation=%.4f  interval=%.6f intervalov=%d ",
mean[gn], RMS[gn], interval[gn], intervalov[gn]);
    wrefresh(floor_win);   // Показать характеристики на экране
```

```
    }
    ch = getch();
    switch(ch)  // Выбор действия в зависимости от нажатой клавиши        {
        case KEY_LEFT: if (gisto_number) gisto_number--;
                       else gisto_number++; break;
        case KEY_RIGHT: if (gisto_number) gisto_number--;
                        else gisto_number++; break;
        case KEY_UP: if (gisto_number)
                       {if (intervalov[1]<20) intervalov[1]++;}
                        else {if(intervalov[0]<20) intervalov[0]++;}
                       break;
        case KEY_DOWN: if (gisto_number)
                           {if(intervalov[1]>1) intervalov[1]--;}
                           else {if(intervalov[0]>1) intervalov[0]--;}
                       break;
        case KEY_F(10): clear(); refresh(); resetty(); endwin();
                  return;  // По F10 очистка экрана, завершение режима ncurses и возврат
    }

} while(1);
```

4. ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОЙ ПОДГОТОВКИ

1. Запустите одним из возможных способов плавную корректировку системного времени до нужного значения. Пока ядро изменяет системное время, запустите программу, содержащую таймеры, реализующие интервалы времени срабатывания (во временной базе CLOCK_REALTIME), которые легко можно проверить наручными часами. По результатам запуска программы и замеров по наручным часам проанализируйте, как влияет плавная подгонка системного времени на фактически отрабатываемые программными таймерами интервалы. Проделайте аналогичные сравнения при скачкообразном изменении системного времени.
2. Создайте в программе группу периодических таймеров с различными периодами срабатывания, используя в качестве схемы уведомления один и тот же сигнал. Различать срабатывания таймеров требуется: а) по кодам сигнала, уникальным для каждого таймера; б) по значениям сигнала, уникальным для каждого таймера.
3. Проделайте задание, аналогичное п.2, приняв в качестве схемы уведомления импульс (*pulse*).
4. Задайте в программе в качестве схемы уведомления о срабатывании таймера создание нового потока. Созданный поток должен выводить на экран сообщение и сразу самоликвидироваться.
5. Есть основания предполагать, что микроядро при реализации периодических таймеров использует единый механизм независимо от вида функций API, запускающих таймер. Если это так, статистические характеристики (математическое ожидание и среднеквадратическое отклонение) интервалов срабатывания таймеров, созданных на основе различных функций библиотеки, должны отличаться незначимо. Проверьте это как статистическую гипотезу, запустив по очереди несколько таймеров с одинаковыми временнЫми характеристиками в одинаковых условиях (приоритет, загрузка компьютера другими программами) с помощью разных функций-таймеров API QNX/Neutrino.
6. Используя средство высокоточного замера интервалов времени ClockCycles(), оцените статистические характеристики - математическое ожидание и среднеквадратическое отклонение – интервалов

срабатывания периодического таймера для случая интервалов, кратных и некратных системному тику. Постройте гистограммы распределения относительных частот интервалов срабатывания (для построения гистограмм используйте библиотеку `ncurses`). Проанализируйте, как влияет на статистические характеристики приоритет потока, в котором работает таймер.

7. Проделайте аналогичное задание для таймера, основанного на периодических прерываниях от последовательного порта.

8. Проделайте аналогичное задание для таймера, использующего в качестве источника периодических прерываний микросхему часов реального времени.

9. Выберите несколько значений интервалов таймера, основанного на прерываниях последовательного порта, которые мало отличаются от интервалов, кратных периоду хода системных часов. Запустите в одной программе такой таймер и найдите для каждого заданного интервала среднеквадратическое отклонение разброса отрабатываемых интервалов. Запустите в другой программе интервальный таймер из библиотеки функций QNX/Neutrino с заданными интервалами, кратными периоду системных часов, и определите среднеквадратическое отклонение разброса отрабатываемых интервалов в этом случае. Сделайте вывод о том, какой таймер работает точнее.

10. Проделайте аналогичное сравнение таймера библиотеки QNX и таймера, использующего прерывания микросхемы часов реального времени.

11. Следуя обсуждению на форуме [3] оцените время переключения контекста потоков, используя следующий алгоритм:

1) Два потока одного процесса с одинаковым приоритетом большим 10 (чтобы не прерывали другие потоки с приоритетом по умолчанию) и дисциплиной планирования FIFO заданное число `n` раз в цикле последовательно выполняют операцию захвата и освобождения двух семафоров S1 и S2 по схеме:

- S1=0, S2=1.
- взять засечку времени `T1` счётчика TSC.
- повторять n раз:
  - захватить S1;
  - освободить S2.
- взять засечку времени `T2` счётчика TSC.

- разница `ΔT=T2-T1` есть суммарное время, потраченное на парную операцию захватить+освободить семафор `tm1` и переключение контекста чередующихся потоков `tm2`.

2) Для определения `tm1` проделайте захват-освобождение семафоров n раз в цикле одним потоком.

3) Усреднённое по n замерам время переключения контекста потоков приблизительно равно (ΔT – tm1)/n.

Используя аналогичный алгоритм, оцените среднее время переключения контекста процессов. Неименованные семафоры для совместного доступа разместите в разделяемой памяти.

12. Используя средства высокоточного замера интервалов времени ClockCycles(), оцените по ряду замеров среднее, наибольшее и наименьшее значения, а также среднеквадратические отклонения интервалов времени, затрачиваемых системой на: а) захват и освобождение семафора; б) захват и освобождение именованного семафора; г) передачу сигнала; д) передачу импульса; е) передачу сообщения объёмом 1 байт средствами обмена сообщениями QNX/Neutrino; ж) передачу сообщения объёмом 1 байт средствами очередей сообщений POSIX; и) запись и чтение байта в/из разделяемой памяти; к) создание потока; л) создание пула потоков; м) создание процесса.

13. Замеры случайных интервалов в п. 2.6.3 двумя способами делались одновременно. Очевидно, это в определённой степени искажало результаты. Проведите замеры одного массива случайных интервалов на одном компьютере по отдельности двумя способами и статистически сравните полученные парные выборки. Проведите сравнение результатов одного способа замера интервалов на компьютерах с различающимися микропроцессорами.

14. Используя “симметричную схему” [42] обмена активностями двух процессов и потоков, при которой заданное число раз программные субъекты добровольно уступают друг другу процессор, не выполняя никаких больше действий, определите среднее время переключения контекста процессов/потоков в тактах процессора и микросекундах на вашем компьютере.

15. Современные наборы системной логики персональных компьютеров поддерживают стандарт ACPI (Advanced Configuration and Power Interface, расширенного интерфейса конфигурирования компьютера и управления питанием). Южный мост таких чипсетов содержит таймер управления питанием

содержит таймер управления питанием (Power Management Timer - PMTMR) [76]. Отсчёты этого таймера используются, в частности, как источник точных временных меток отдельными версиями ядра ОС Linux, являясь альтернативой TSC микропроцессора. Напишите программу, использующую PMTMR в качестве средства точного измерения временных интервалов. Сравните с результатами замеров “штатными” средствами ОС QNX.

16. Работа интервального таймера может быть описана алгоритмом, блок-схема которого приведена на рис.4.1. Для формального описания алгоритма введём следующие обозначения переменных: start_interval - заданный промежуток времени от запуска до первого срабатывания таймера; timer_period – заданный интервал периодических срабатываний; timer_time –“таймерное время” – промежуток времени от первого после запуска таймера системного тика до очередного срабатывания; ticks – системное время в тиках с момента запуска таймера до текущего момента; local_fires – счётчик срабатываний таймера в пределах большого периода; local_ticks_count – счётчик системных тиков в пределах большого периода; fire_condition – условие, при котором таймер срабатывает. Величина системного такта обозначена как tick. Составьте программу расчёта моментов срабатывания таймера по вышеуказанному алгоритму. Сравните результаты вычислений с последовательностью интервалов срабатывания реального таймера, запущенного в QNX-программе при различных значениях start_interval и соотношениях timer_period / tick. Отдельно просчитайте проиллюстрированные на рис.2.2 (3*tick/3.1*tick, 2.3*tick) и рис.2.3 (1000 и 999.847) случаи. Сформулируйте вывод о границах применимости данного алгоритма работы таймера. Проверьте, для какого из таймеров – реального или имитируемого – среднее значение в пределах большого периода абсолютных величин разностей заданных и фактических моментов срабатывания меньше.

17. В статье [77] Р.Кёртен подробно анализирует временно́е поведение одной реальной системы управления. Акцент сделан на влиянии диспетчеризации в многопоточной программе на кажущуюся продолжительность выдержки временнЫх интервалов таймеров. Автор приводит формулу для моментов времени, в которые микроядро генерирует импульсы таймера, сопровождая её пояснением “…the formula above will accurately reflect the kernel itself de-

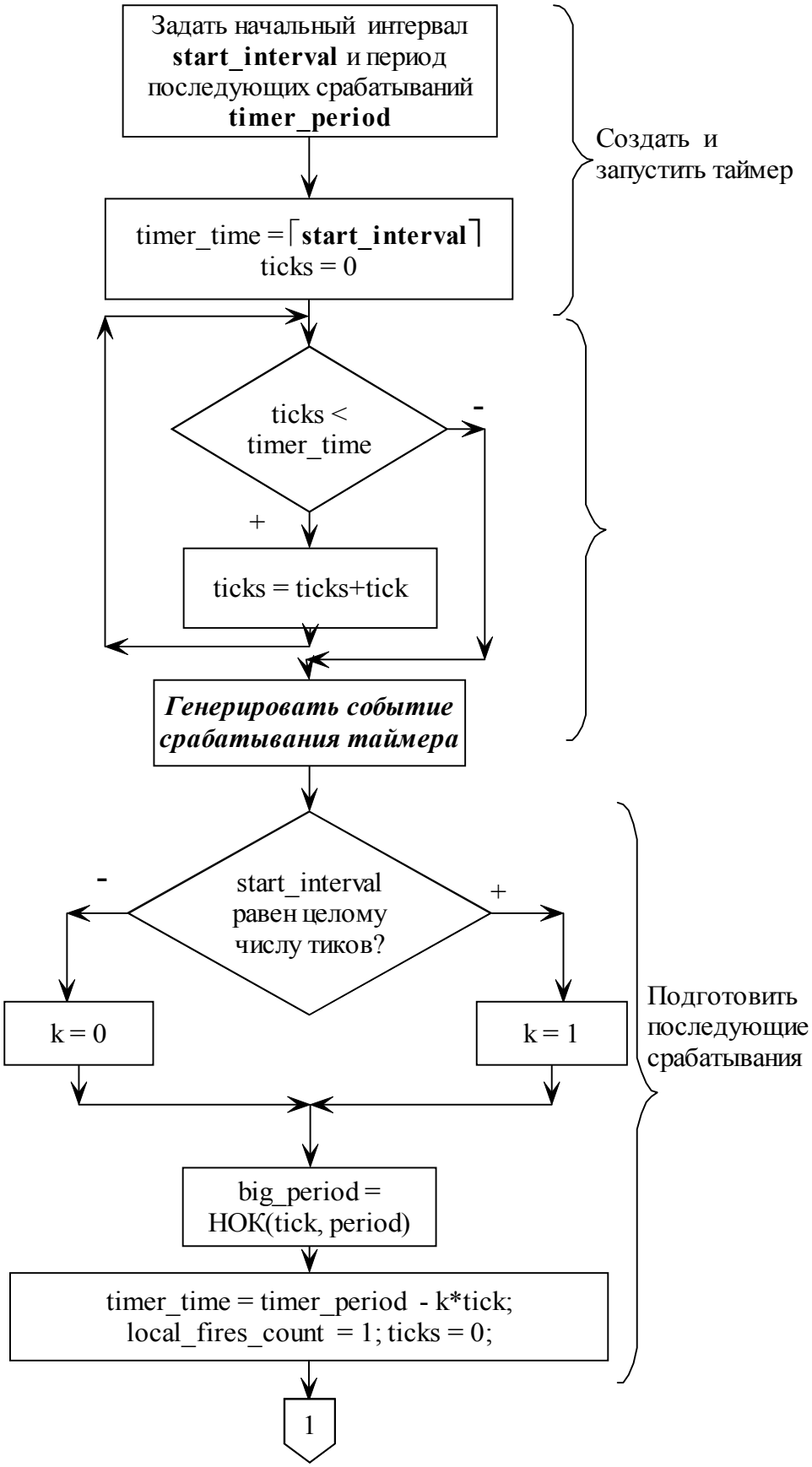


Рис. 4.1. Блок-схема работы интервального таймера (начало)

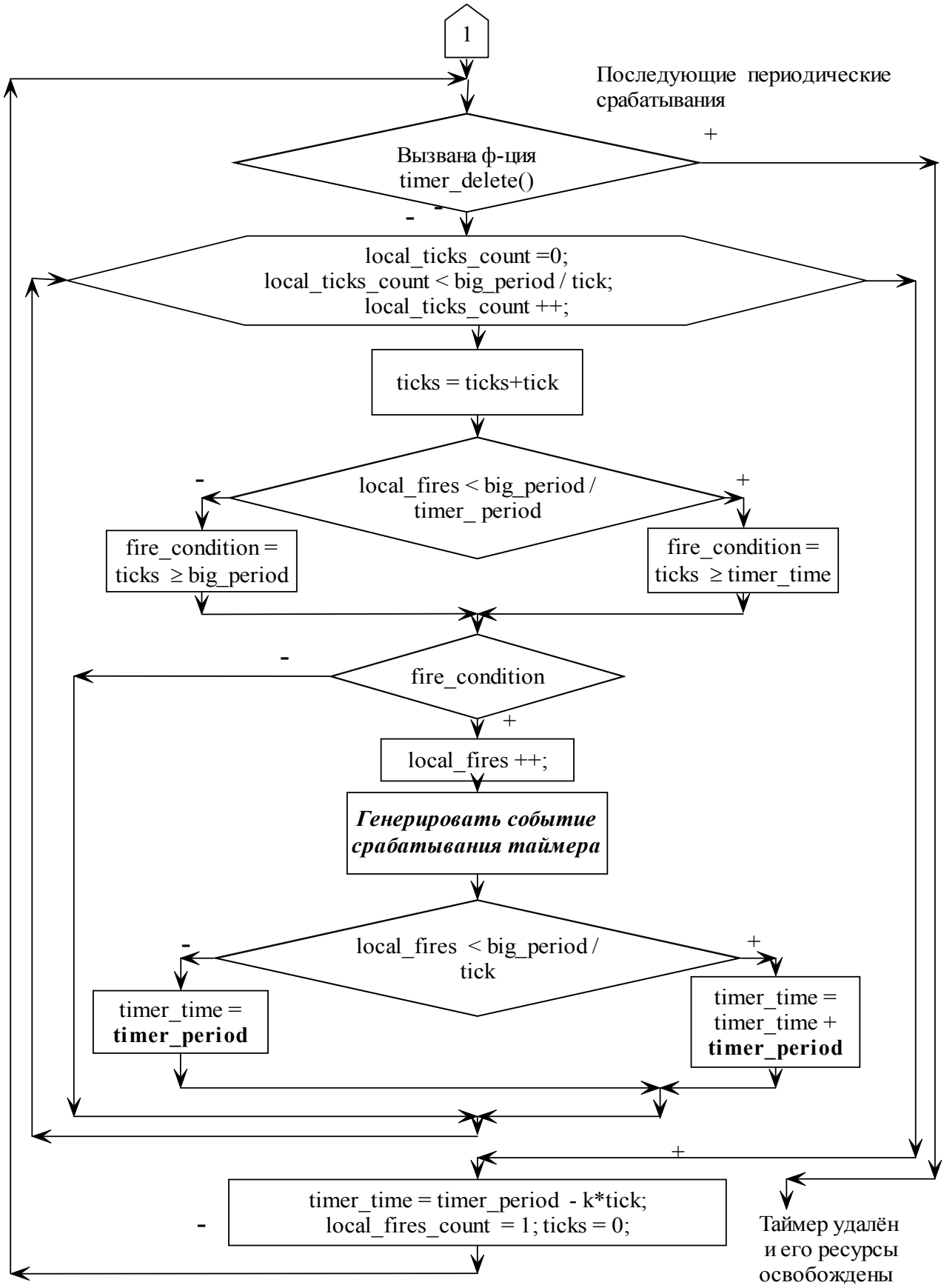


Рис. 4.1. Блок-схема работы интервального таймера (окончание)

cided when to issue the pulse" ("вышеприведенная формула аккуратно отражает как само ядро решает, когда генерировать импульс"):

NSI*PN*((TP+NSI) / NSI) + ES,

где

- **NSI**- величина системного тика в нс (значение, возвращаемое макросом SYSPAGE_ENTRY(qtime)→nsec_inc,
- **PN** - номер импульса,
- **TP** - заданный период таймера в нс,
- **ES** - системное время старта таймера в нс.

Сравните результаты вычислений по формуле Кёртена с последовательностью интервалов срабатывания реального таймера, запущенного в QNX-программе при различных значениях соотношения **TP/NSI**. Проверьте, насколько точно воспроизводит последовательные интервалы между срабатываниями периодического таймера следующая программа:


```
#include <stdio.h>
#define relative_interval 2.3 // Заданный интервал таймера в системных тиках    TP / NSI

main()
{
    long long NSI = 999847LL; // NSI – системный тик в нс
    long long PN;  // PN - номер импульса таймера
    long long TP = (long long) (double) NSI *relative_interval;
                                /* TP- заданный период таймера в нс */
    long long ES = 0LL,         // ES – системное время запуска таймера в нс
                  expire,       // Заданные моменты срабатывания таймера в нс
```

```
                prev;
long long next_tick = 0LL, prev_tick; /* Узлы шкалы системного времени,
                    ближайшие к заданному моменту срабатывания таймера сверху и снизу */

printf("TP = %lld\n", TP);
for(PN=0LL; PN<25LL; PN++)
{
    expire = PN*TP;
    printf("specified fire time=%lld ", expire);
    if (PN == 0) printf("\n");
    while (next_tick < expire) next_tick = NSI + next_tick;
    if(PN)
    {
        prev_tick = next_tick - TP;
        if ((next_tick-expire) < (expire - prev_tick))
        {
            printf(" real fire time = %lld ns  intrvl=%lld ns
                  intrvl=%.2f tick\n", next_tick, next_tick - prev,
                      (double) (next_tick - prev)/NSI);
            prev = next_tick;
        }
        else
        {
            printf(" real fire time = %lld ns  intrvl=%lld ns
                      intrvl=%.2f tick\n", prev_tick, prev_tick-prev,
                    (double) (prev_tick - prev)/NSI);
```

```
                prev = prev_tick;
            }
        }
    }
}
```

ЗАКЛЮЧЕНИЕ

Осмысление материала настоящей части пособия должно сформировать у читателя правильные представления о возможностях и способах использования предоставляемых ОС РВ QNX средств службы времени, об имеющихся проблемах и “подводных камнях”.

Прежде всего, следует избавиться от иллюзии, которая часто возникает по прочтении термина “операционная система реального времени”. У неискушённого человека может появиться мнение, что такая операционная система позволяет очень точно отрабатывать в программах одиночные или повторяющиеся интервалы времени произвольной длительности. Как мы видели, это далеко не всегда так.

Во-первых, никакая операционная система не может “перепрыгнуть через голову железа”, т.е. дать что-то, выходящее за пределы возможностей аппаратных устройств, лежащих в основе службы времени. Ответственность аппаратного таймера за неточность хода системных часов и основанных на этих часах различного вида программных таймеров обусловлена ограниченной частотой импульсов, долговременными и кратковременными случайными флуктуациями частоты и не кратностью периода целому числу микросекунд. Дополнительные погрешности может вносить временна́я непредсказуемость работы других компонентов аппаратной части компьютера, в частности системы прерываний и кэша процессора.

Во-вторых, на длительность отрабатываемых программными таймерами интервалов существенным образом влияет программное окружение, в котором эти таймеры работают. Хотя сами таймеры как объекты микроядра работают по принципу “солдат спит, служба идёт”, т.е. срабатывают независимо от того, в каком состоянии находится поток, ожидающий срабатывания, само поведение такого потока после истечения времени таймера может быть разным в зависимости от того, какие ещё неблокированные потоки с какими приоритетами и политиками планирования присутствуют в системе.

В-третьих, на величины интервалов программных таймеров, особенно однократных, влияет требование стандарта POSIX “не раньше чем”. Как мы видели, это обстоятельство совместно с некратностью периода аппаратного таймера целому числу микросекунд может приводить к существенному (трёхкратному) увеличению циклически отсчитываемого таймером задержки времени “сна” потока по сравнению с заданным. Не менее опасным в программах реального

времени может быть обусловленный теми же причинами периодический пропуск отдельных срабатываний в цикле при близости заданного периода таймера задержки к величине системного тика.

Весь комплекс факторов, влияющих на работу программных таймеров, проявляется в том, что интервалы срабатываний таймеров выглядят как случайные величины. Поэтому делать какие-то заключения о точности и/или стабильности работы таймеров можно только статистическими методами. Рассмотренные в настоящей части пособия простейшие статистические подходы и критерии могут быть основой для этого. Более аккуратная оценка точности и/или стабильности возможна на основе внешних, не влияющих на поведение таймеров средств измерения интервалов и использования более современных и информативных методов анализа теории случайных процессов и математической статистики.

Однако после прочтения материала первой части у читателя не должно возникнуть и иллюзии противоположного направления – “служба времени не точна, все таймеры врут, никаких нужных при программировании реальных систем временнЫх интервалов достичь нельзя”. Такой взгляд - глубокое заблуждение. И вот по каким причинам.

Во-первых, указанные проблемы актуальны для любой ОС РВ. Для их решения QNX/Neutrino, как никакая другая ОС, предоставляет программисту богатейший API по работе со временем, позволяющий использовать все возможности аппаратной части компьютера. Сюда прежде всего входят как функции стандартов POSIX, так и расширяющие их возможности системные вызовы самого микроядра. Полезными могут быть также группы функций, принадлежащие стандартам ANSI, UNIX и др. Отметим в качестве ценных следующие свойства службы времени QNX/Neutrino:

- возможность в широких пределах менять системный тик;
- возможность плавно корректировать время системных часов без скачкообразного “обрушения” интервалов запущенных в системе таймеров;
- развитые средства задания различного вида таймаутов, в том числе возможность прямо задавать таймауты на большиство блокированных состояний потоков;
- возможность использовать дополнительные аппаратные таймеры, расположенные на внешних платах расширения или микро-

схемах чипсета материнской платы. Модульная микроядерная архитектура QNX/Neutrino в сочетании с техникой менеджера ресурсов делает эту задачу практически решаемой без необходимости изменять ядро операционной системы;

– наличие развитых средств измерения временнЫх интервалов, включая доступ ко внутренним системным структурам, содержащим времена использования процессора каждым процессом/потоком.

Во-вторых, несомненным достоинством QNX/Neutrino являются малые накладные расходы по времени на выполнение различных системных действий, что является следствием хорошо спроектированной архитектуры и в целом повышает временну́ю предсказуемость системы.

В-третьих, широкий практический опыт применения ОС QNX/Neutrino в сфере автоматизации и управления различными устройствами (космос, военная техника, промышленное, медицинское, транспортное оборудование), в том числе такими быстродействующими, как сетевые маршрутизаторы Cisco [78], даёт все основания утверждать, что служба времени умеет адекватно выполнять свои функции при правильном её использовании.

В-четвёртых, разработчик программ должен руководствоваться принципом “предупреждён – значит вооружён”. На практике это означает, что квалифицированный программист применяет средства службы времени с пониманием того, что за ними стоит, какие средства предпочтительно использовать в конкретных случаях, какие могут быть нежелательные эффекты от их применения и как этих нежелательных эффектов избежать, чтобы “не наступить на грабли”, положенные на тропинку всеми отмеченными выше взаимодействующими свойствами аппаратуры и программного окружения.

Если после прочтения этой части пособия становится ясно, на что необходимо обращать особое внимание, чего можно и чего нельзя ожидать от службы времени QNX/Neutrino при программировании систем реального времени, читатель может считать, что потратил своё реальное время не зря.

Автор будет признателен за конструктивные замечания по пособию. Связаться с автором можно по адресу nikolsky@ctam.tu-bryansk.ru.

СПИСОК ИСПОЛЬЗОВАННОЙ И РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

Рекомендуемые источники выделены полужирным шрифтом

1. **http://www.qnx.com**
2. **http://qnx.org.ru/forum**
3. **http://qnxclub.net/modules.php?name=Forums**
4. **http://www.swd.ru/qnx**
5. http://ed1k.qnx.org.ru/timesync.html
6. **Зыль, С.Н. – Операционная система реального времени QNX: от теории к практике/С.Н. Зыль – 2-е изд., перераб. и доп. – СПб.: БХВ-Петербург, 2004. – 192 с.**
7. **Зыль, С.Н. –QNX Momentics: основы применения/ С.Н. Зыль – СПб.:БХВ-Петербург,2005.-256 с.** Имеется электронный вариант книги по адресу http://swd.ru/index.php3?pid=499
8. IEEE Std 1003.1, 2004 Edition, RATIONALE http//www.opengroup.org/onlinepubs/009695399/xrat/xsh_chap02.html#tag_03_02_08_18
9. David W. Allan, Neil Ashby, Cliff Hodge; Science of Timekeeping; Hewlett-Packard Application Note 1289; 1997. http://www.allanstime.com/Publications/DWA/Science_Timekeeping/index.html
10. Burns,A., Wellings,A.Real-Time Systems and Programming Languages (3rd Edition), ch.12. http://www.cs.york.ac.uk/rts/RTSBookThirdEdition.html
11. Time and Frequency Users Manual. NBS Special Publication 559 May 1997. Chapter 2, Section 2.1 “TIME SCALES, UTC, and LEAP SECONDS” http://www.ieee-uffc.org/freqcontrol/fc_utc.html
12. Время (методы измерения) http://www.astronet.ru/db/msg/1190813
13. **Кёртен, Р. Введение в QNX Neutrino 2: руководство по программированию приложений реального времени в QNX Real-time Platform/ Р. Кёртен – СПб.: Петрополис, 2001 - 480 с. - СПб.:БХВ-Петербург, 2005 доп. тираж -400 с.**
14. Отчёт Dedicated System Experts о тестировании QNX NEUTRINO RTOS V 6.2 http://www.cniil.org/QNX_NEUTRINO_RTOS_V6_2_0.html
15. Отчёт Dedicated System Experts WINDOWS CE 5.0 ON AN X86 PLATFORM. EVA-2.9-TST-CE-x86-01.pdf http://www.microsoft.com/downloads/details.aspx?FamilyID=ba634ab4-ebea-44f8-8500-12b186ebe10b&DisplayLang=en

16. Laptop And ACPI. http://lists.freebsd.org/pipermail/freebsd-acpi/2005-March/001326.html
17. Снижение производительности многопроцессорных компьютеров под управлением Windows XP с пакетом обновления 2 (SP2), которые поддерживают функции управления питанием процессоров http://support.microsoft.com/?id=896256
18. How Microsoft Windows NT 4_0 Handles Internal Time.htm. http://home1.gte.net/dougho/TimeServ.html
19. Timing Pitfalls and Solutions for Windows\PC Timers.htm http://lists.ntp.isc.org/pipermail/questions/2005-March/004597.html
20. Srinivasan, B., Pather, S., Hill, R., Ansari, F., and Niehaus, D. 1998. A Firm Real-Time System Implementation Using Commercial Off-The Shelf Hardware and Free Software, in: Proceedings of RTAS-98, June, Denver, CO, 112-119. http://uk.builder.com/whitepapers/0,39026692,60009321p-39000844q,00.htm
21. http://www.ittc.ku.edu/kurt/
22. IA-PC HPET (High Precision Event Timers)Specification Revision: 1.0a Date: October 2004 www.intel.com/hardwaredesign/hpetspec_1.pdf http://www.intel.com/labs/platcomp/hpet/hpetspec.htm
23. Алексей Шашков. Технологии ACPI и OnNow. http://www.ixbt.com/mainboard/acpi.html
24. Отчёт Dedicated Systems Experts DSE-RTOS-EVA-002 EVALUATION REPORT DEFINITION http://dedicated-systems.com/encyc/BuyersGuide/RTOS/Evaluations/
25. N.Carter. How to Monitor Interrupts on the Parallel Port. http://psy.swansea.ac.uk/staff/Carter/QNX/
26. K. Gordon, J. Gwinn, J. Oblinger, F. Prindle. A Programmer's Overview of the POSIX.1d Draft Standard: Additional Realtime Extensions. http://www.usenix.org/publications/login/standards/22.posix.html
27. IEEE Std 1003.1, 2004 Edition, раздел ADVANCED REALTIME http://www.opengroup.org/onlinepubs/009695399/functions/clock_getcpuclockid.html
28. **Галатенко, В.А. Программирование в стандарте POSIX. Часть 2. Курс лекций: Учеб пособие: Для студентов вузов, обучающихся по специальности 351400 “Прикладная информатика”/ В.А. Галатенко. – М.: Интернет-университет информ. технологий. 2005.-384 с.** Имеется электронный вариант книги по адресу http://www.intuit.ru/department/se/posix2/
29. http://www.opengroup.org/onlinepubs/009695399/functions/timer_gettime.html

30. QNX/Neutrino CVS Repository. http://cvs.qnx.com/cgi-bin/cvsweb.cgi/lib
31. http://www.opengroup.org/onlinepubs/009695399/functions/getitimer.html
32. Статья Real-time Operating System Timing Jitter and its Impact on Motor Control, Frederick M. Proctor and William P. Shackleford National Institute of Standards and Technology www.isd.mel.nist.gov/projects/rtlinux/motor-jitter.pdf
33. Kevin M. Obenland. POSIX in Real-Time. Embedded Systems Programming 03/15/01 www.embedded.com/story/OEG20010312S0073
34. Статья Tick-tock - Understanding the Neutrino micro kernel's concept of time, Part II by Mario Charest, with Brian Stecher's assistance. http://www.qnx.com/developers/articles/article_826_2.html
35. Учебный курс фирмы QNX Software Systems Ltd.“Realtime Programming for the QNX® Neutrino RTOS” 2004/04/28 R22
36. Статья Tick-tock - Understanding the Neutrino microkernel's concept of time by Brian Stecher http://www.qnx.com/developers/articles/article_834_1.html. Русский перевод http://qnx.org.ru/article29.html
37. Ada 95 Reference Manual, Annex D. http://www.adahome.com/rm95/rm9x-D.html
38. Ada 95 Reference Manual. http://www.adahome.com/LRM/95/rm95html/rm9x-09-06.html#2
39. http://www.ada-ru.org/
40. http://www.gnat.com/
41. **Алексеев, Д. Практика работы с QNX/Д.Алексеев, Е.Видревич, А.Волков, Е.Горошко, М.Горчак, Р.Жавнис, А.Крисак, Д.Сошин, О.Цилюрик, А.Чиликин – М.: Издательский Дом “КомБук”, 2004. – 432 с.**
42. **Цилюрик О., Горошко Е. QNX/UNIX: анатомия параллелизма/ О.Цилюрик, Е.Горошко – СПб.: Символ-Плюс, 2006. -288 с.**
43. Krten, R. The QNX CookBook: Reciepes for Programmers/ R.Krten – PARSE Software Devices, 2003, ISBN 0-9682501-2-2
44. Agner Fog. Руководство “How to optimize for the Pentium family of microprocessors”. www.agner.org/assem/pentopt.pdf
45. Статья Эдуард Кромской Использование аппаратных часов реального времени в QNXRTP (x86) http://qnx.org.ru/article23.html
46. A.Pasztor, D.Veitch. PC Based Precision Timing Without GPS http://parapet.ee.princeton.edu/~sigm2002/papers/p1-pasztor.pdf
47. Mills, D.L. The network computer as precision timekeeper. Proc. Precision Time and Time Interval (PTTI) Applications and Planning Meeting (Reston VA, December 1996), 96-108. http://www.eecis.udel.edu/~mills/database/papers/ptti.pdf

48. M.A. Weiss, D.W. Allan, D.D. Davis, and J. Levine; Smart Clock: A New Time; IEEE Trans. Instrum. Meas., 41, 915-918, 1992. http://www.allanstime.com/Publications/DWA/SmartClock.pdf
49. АО “МОРИОН” http://www.morion.com.ru/
50. NTP temperature compensation. http://www.ijs.si/time/temp-compensation/
51. http://karavaikin.h1.ru/_private/rab152.htm
52. Касперски, К. Техника оптимизации программ: Эффективное использование памяти/ К. Касперски – СПб.: БХВ-Петербург, 2003. – 560 с.
53. Зубков, С.В. Assembler. Для DOS, Windows и UNIX/ С.В.Зубков – М.:ДМК, 1999. – 640 с., ил.
54. Юров В. Ассемблер: Практикум /В.Юров.- СПб.:Издательство Питер,2002.-400с. http://computerbooks.ru/books/Programming/Book-Tasks-and-Examples-Assembler/Glava%208/Index1.htm
55. Using the RDTSC Instruction for Performance Monitoring. http://cedar.intel.com/software/idap/media/pdf/rdtscpm1.pdf
56. Отчёт www.eecis.udel.edu/~mills/database/reports/allan/secureb.pdf
57. Т. Шайе, С. Юхансен, Т. Лекстад, Э. Холмайде Точная синхронизация времени для задач автоматизации http://www.abb.ru/GLOBAL/RUABB/RUABB056.NSF/99ad595c32e0c2d9c12566e1000a4540/f60af869d98e3545c1256d96003c22f9/$FILE/P9_14.pdf
58. Michael A. Lombardi. "Time Measurement." Chapter 18 82.171.205.59/downloads/PDFBOOKS/SensorsHandbook/CH18.PDF
59. Poul-Henning Kamp. Using the Soekris computers for timestamping. http://phk.freebsd.dk/soekris/pps/
60. Jeff Roberson. Accurate, High Resolution Absolute Timing on the PC Platform. Systems Magazine - 2001 Q3 http://www.eyring.com/eyrx/NEWS/ARTICLE901.HTM http://www.omimo.be/magazine/01q3/2001q3_p077.pdf
61. Сайт экспертов в области систем реального времени и встроенных систем http://www.dedicated-systems.com
62. W.J. Riley. The calculation of time domain frequency stability. Addendum to a Test Suite for the Calculation of Time Domain Frequency Stability, Proc. 1996 IEEE Freq. Contrl. Symp., pp. 880-882, June 1996 www.ieee-uffc.org/freqcontrol/paper1ht.html.
63. http://perso.club-internet.fr/yazdiet/
64. D.A. Howe, D.W. Allan, and J.A. Barnes. PROPERTIES OF OSCILLATOR SIGNALS AND MEASUREMENT METHODS http://tf.nist.gov/timefreq/phase/Properties/main.htm

65. **Орлов, А.И. ПРИКЛАДНАЯ СТАТИСТИКА. Учебник для вузов. Издательство «ЭКЗАМЕН» МОСКВА 2004. Прикладная статистика. Учебник. / А.И.Орлов.- М.: Издательство «Экзамен», 2004. - 656 с. http://orlovs.pp.ru/stat.php**
66. Браунли, К.А. Статистическая теория и методология в науке и технике. Перевод с английского/К.А.Браунли – М:. Главная редакция физико-математической литературы изд-ва “Наука”,1977,-408с.
67. Engineering Statistic Handbook (NIST/SEMATECH e-Handbook of Statistical Methods), http://www.itl.nist.gov/div898/handbook
68. http://www.statisticalengineering.com/goodness.htm
69. http://www.itl.nist.gov/div898/software/dataplot/homepage.htm
70. http://www.statplus.net.ua/ru/
71. http://www.ge.infn.it/geant4/analysis/HEPstatistics/gof/deployment/userdoc/statistics/index.html
72. MIL-HDBK-5J “METALLIC MATERIALS AND ELEMENTS FOR AEROSPACE VEHICLE STRUCTURES” http://www.weibull.com/knowledge/milhdbk.htm
73. Большев, Л.Н., Смирнов, Н.В. Таблицы математической статистики/ Л.Н. Большев, Н.В.Смирнов. – М.:Наука. Главная редакция физико-математической литературы, 1983,-416 с.
74. Айвазян, С.А., Енюков И.С., Мешалкин Л.Д. Прикладная статистика: основы моделирования и первичная обработка данных. Справ. Издание.-М.: Финансы и статистика, 1983,471с.
75. NUMERICAL RECEIPES in C. http://www.library.cornell.edu/nr/cbookcpdf.html
76. ACPI Specification 3.0a http://cdgenp01.csd.toshiba.com/content/pr/download/AdvancedConfiguration_PowerSpecificationv3.pdf
77. R.Krten. Tips, Tricks and Techniques to Tame Timer Trouble. http://www.parse.com/articles/timers.html
78. 25 Ways QNX Touches Your Life. http://www.qnx.com/news/pr_1329_3.html