**Кафедра «Программное обеспечение ЭВМ и информационные технологии»**

Курсовая работа

**на Тему: «Реализация системы управления реального времени в ОС Windows»**

**Содержание**

1. Введение

2. Конструкторская часть

2.1. Общие принципы

2.2. Програмное обеспечение

2.2.1. Драйвер режима ядра

2.2.2. Управляющее приложение

2.2.3. Приложение для создания нагрузки

2.2.4. Обратная связь

3. Технологическая часть

3.1. Выбор средства разработки

3.2. Организация задержек

3.3. Взаимодействие с драйвером

4. Исследовательская часть

4.1. Цели и задачи

4.2. Конфигурация тестового стенда

4.3. Работа на небольших частотах

4.4. Точность изменения задержек

4.5. Точность работы таймера

4.6. Увеличение частоты срабатывания

4.7. Работа параллельно с другими приложениями

4.7.1. Нагрузка на подсистему GDI

4.7.2. Работа со страничными отказами

5. Заключение

Приложение 1. Исходный код управляющего потока

Приложение 2. Исходный код рабочего потока

**1. Введение**

В настоящее время компьютеры прочно вошли в нашу жизнь. Сложно найти такой аспект повседневной жизнедеятельности, в которой еще не используется современная вычислительная техника. Не являются исключением и различные научно-исследовательские работы. Так, к нам в руки попало специальное устройство, предназначенное для аналого-цифрового и цифро-аналогого преобразования сигнала на аппаратном уровне. С устройством поставляется специальный драйвер для работы в операционной системе (далее ОС) Windows.

После первых же экспериментов с устройством выяснилось, что работа с ним возможна только на небольших частотах обрабатываемого сигнала. При увеличении частоты наблюдается искажение сигнала связанное с тем, что система не успевает обрабатывать приходящие данные и выдавать данные в ответ. Этот результат можно считать закономерным, учитывая что ОС Windows вообще говоря не является операционной системой реального времени. Однако, сама операционная система содержит в себе набор средств, которые предположительно могут позволить создать систему управления реального времени в ОС Windows.

**2. Конструкторская часть**

**2.1 Общие принципы**

Одним из возможных способов решения поставленной задачи может быть использование специальных потоков реального времени ОС Windows. Такие потоки имеют приоритеты от 16 до 31 и выполняются в режиме ядра. Кроме того, важным отличием таких потоков от обычных является то, что они являются потоками с добровольной передиспетчеризацией. Это означает, что если такой поток получает процессор (как системный ресурс), то он будет занимать его до тех пор, пока сам добровольно не вернет его системе, т.е. не перейдет в состояние блокировки (например в ожидание на функции WaitForSingleObject).

Именно этой особенность системных потоков реального времени мы и попробуем воспользоваться при реализации системы управления реального времени в ОС Windows.

**2.2 Программное обеспечение**

Для проведения исследований нам понадобится следующий набор программных средств: драйвер режима ядра, управляющее приложение, приложение для создания нагрузки, обратная связь.

Каждый из этих пунктов подробнее рассматривается далее в этом разделе.

**2.2.1 Драйвер режима ядра**

Для создания системных потоков нам необходимо использовать драйвер, которым в нашем случае является драйвер виртуального устройства. Связано это с тем, что в ходе наших исследований использование реального устройства было невозможно, кроме того, в этом не было острой необходимости. Работу реального устройства мы будем эмулировать при помощи еще одного системного потока внутри нашего драйвера.

Итак, при начальной инициализации (в функции DriverEntry) драйвер запускает два системных потока. Первый из них (эмулирующий реальное устройство, управляющий поток, см. Приложение 1) исполняется с приоритетом 31 и ждет на системном объекте «ожидающий таймер» (waitable timer). Это позволяет потоку пробуждаться через заранее определенные промежутки времени и будить второй поток, имитируя тем самым приход некоторых данных от внешнего устройства.

Второй поток (рабочий поток, см. Приложение 2) выполняется с настраиваемым приоритетом (от 16 до 30) и предназначен для обработки данных приходящих от внешнего устройства. Для этого он ждет на событии до тех пор пока оно не будет взведено управляющим потоком. Затем поток выполняет некоторое число холостых циклов на процессоре для имитации обработки данных. Количество таких циклов зависит от того, какую длительность задержки мы хотим использовать.

**2.2.2 Управляющее приложение**

При помощи специального приложения происходит управление работой драйвера. Внешний вид этого приложения показан на рис. 2.1.

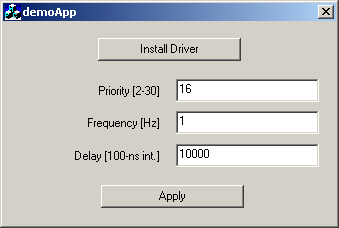


Рис 2.1. Внешний вид управляющего приложения.

Верхняя кнопка позволяют установить драйвер. Следующие три поля ввода позволяют задать соответственно приоритет рабочего потока (от 2 до 30), частоту запросов от эмулируемого внешнего устройства (в герцах) и задержку (в 100-наносекундных интервалах), которая будет использована для имитации обработки полученных от устройства данных.

Нижняя кнопка позволяет применить сделанные изменения.

**2.2.3 Приложение для создания нагрузки**

Поскольку в реальных условиях параллельно с нашим драйвером будут выполняться и другие приложения, важно учесть это в ходе нашей исследовательской работы. Для этих целей было разработано специальное приложение, внешний вид которого показан на рис. 2.2.

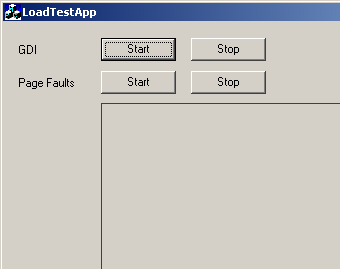


Рис 2.2. Внешний вид приложения для создания нагрузки (фрагмент)

Первая пара кнопок позволяет включать или выключать нагрузку на подсистемы GDI, вторая пара кнопок соответственно влияет на генерацию множественных страничных отказов.

**2.2.4 Обратная связь**

Для определения текущего состояния драйвера необходимо поддерживать с ним обратную связь. Из всех возможных способов такой связи был выбран самый простой — посылка текстовых строк отладчику ядра функцией DbgPrint. Для чтения таких строк можно использовать специальные программные средства, в нашем случае будет использовано приложение DebugView 4.31 от Марка Руссиновича (Mark Russinovich), внешний вид которого показан на рис. 2.3.

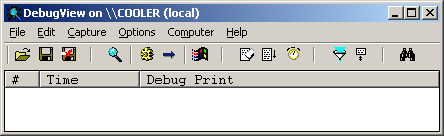


Рис 2.3. Приложение DebugView

**3. Технологическая часть**

**3.1 Выбор средства разработки**

Для разработки драйверов наиболее широкое применение получили языки C/C++ и Ассемблер. Для разработки настоящего драйвера был выбран язык C++, поскольку он сочетает в себе простоту разработки программ с возможностью использования ассемблерных вставок для критических участков кода. Кроме того, заголовочные файлы для этого языка идут в стандартной поставке DDK. Для сборки конечного драйвера использовались компилятор и линковщик из DDK.

**3.2 Организация задержек**

Для имитации обработки данных, полученных от устройства необходимо создавать задержки. Для этого можно использовать вызов специальной функции ядра KeStallExecutionProcessor, однако в этом случае мы не можем контролировать что на самом деле произойдет с потоком и мы получим меньше информации о том сколько тактов центрального процессора было востребовано.

Кроме того, при реальной работе вместо фиктивных задержек будут использованы реальные математические вычисления на центральном процессоре. Наиболее удачной имитацией этого на наш взгляд будет создание холостых циклов при помощи ассемблерных вставок например такого вида:

mov ecx, count

label: xchg eax,eax

loop label

Здесь параметр count определяет, сколько именно холостых циклов нужно выполнить, причем мы даже можем приблизительно предположить, сколько тактов центрального процессора на это уйдет.

Для приблизительного пересчета числа холостых циклов во временные интервалы при начальной инициализации драйвера выполняется порядка 108 холостых циклов с вычислением времени которое было затрачено на эту операцию. Далее необходимое число холостых циклов для организации задержки рассчитывается по следующей формуле:

,



где *count0 и time0* соответственно количество холостых циклов и время на их выполнение при начальной инициализации драйвера, *time* — требуемая длина задержки.

**3.3 Взаимодействие с драйвером**

Управление параметрами работы драйвером производится при помощи следующей структуры:

struct ControlStruct

{ int Priority;

int Frequency;

int Delay;};

Поле Priority задает текущий приоритет рабочего потока драйвера (приоритет управляющего потока всегда 31). Поле Frequency задает частоту прихода данных от эмулируемого внешнего устройства. Поле Delay определяет длительность задержек на обработку данных от устройства.

Передача обновленной структуры из управляющего приложения в драйвер осуществляется следующим образом. Приложение открывает виртуальное устройство, ассоциированное с драйвером, а замет вызовом WriteFile передает в драйвер нужные данные.

Драйвер получает данные в своей функции DispatchWrite и сохраняет их в глобальной переменной, внося необходимые изменения в свою работу.

**4. Исследовательская часть**

**4.1 Цели и задачи**

Для того чтобы начать исследование, необходимо определиться, как будет использоваться полученное программное обеспечение в ходе его проведения, и каких результатов мы хотим добиться.

При помощи управляющего потока мы будем имитировать работу реального устройства на некоторой частоте. Начав с небольших частот мы попытаемся довести частоту до 1 кГц и добиться устойчивой работы на ней.

Следует также определиться с тем, что мы будем понимать под устойчивой работой. Поскольку речь идет о системе управление реального времени, то необходимо, чтобы к моменту прихода следующего запроса от устройства предыдущий был уже полностью обработан. Если по приходу запроса мы обнаружим что предыдущий запрос еще обрабатывается, мы будем считать что на данной частоте с данным временем задержки на используемой системе реализация системы управления реального времени невозможна.

**4.2 Конфигурация тестового стенда**

Исследования проводились на компьютере со следующей конфигурацией: AMD Duron 1000 MHz, EPoX 8KHA+ (VIA KT266A), 384 Mb DDR SDRAM. Файлы подкачки системы располагались на винчестерах Seagate Barracuda и Western Digital со скоростью вращения шпинделя 7200 об/мин, подключенными по интерфейсу IDE.

Операционная система, установленная на тестовой машине, Microsoft Windows XP Pro SP2. Система недавно переустанавливалась, работает стабильно.

**4.3 Работа на небольших частотах**

Начнем наше исследование с запуска драйвера на небольших частотах, а именно ограничимся частотой в 1 Гц и пронаблюдаем какие максимальные задержки мы сможем выставить сохраним стабильную работу драйвера.

Для простоты пока будем предполагать что кроме нашего драйвера система больше не загружена никакими ресурсоемкими приложениями. То есть в диспетчере задач более 95% процессорного времени приходится на псевдо-процесс Idle («Бездействие системы»). Приоритет рабочего потока выставим в значение 16.

Частота 1 Гц дает нам приход запросов от имитируемого устройства каждые 1000 мс. Изначально была выставлена задержка 100 мс. Это дает системе 900 мс на «накладные расходы», что позволило драйверу стабильно выполнить необходимые действия. На прикладном уровне работа драйвера без привлечения специальных средств не ощущалась.

Затем мы стали увеличивать длительность задержки. Значение было доведено до 990 мс (что дает системе 10 мс на обработку «накладных расходов»). При такой задержке стабильная работа драйвера сохранилась, однако на прикладном уровне серьезно ощущались «рывки» — система тратила 99% времени на обработку в драйвере.

Возможно, длительность задержки можно было еще увеличить, однако это было сделать довольно трудно, т.к. система плохо реагировала на действия пользователя. При значении задержки 1000 мс, как и следовало ожидать, стабильность работы драйвера нарушалась.

Еще нужно сказать о том, что если для частоты 1 Гц затраты на накладные расходы системы длиной в 10 мс можно считать нормальными, то на более высоких частотах (выше 100 Гц) это будет недопустимо большой паузой.

**4.4 Точность изменения задержек**

Далее по ходу исследования нам будет необходимо увеличивать частоту запросов и уменьшать длительность задержек. Не лишним будет по логу работы драйвера убедится, что выбранный нами способ организации задержек работает корректно. На рис 4.1 показан один из участков лога.

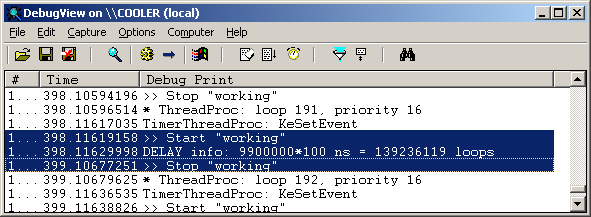


Рис 4.1. Точность измерения задержек

Начало задержки 990 мс — время 398.11619158, конец — 399.10677251. Расхождение с расчетным временем составило 0.00058093 с. Конечно, при уменьшении задержек и при включении внешней нагрузки погрешность увеличится, однако такой результат все равно можно назвать достаточно хорошим.

**4.5 Точность работы таймера**

Вместо внешнего устройства мы использовали ожидающий таймер, поэтому интересно будет проверить, насколько точно он позволит нам выдерживать необходимую частоту срабатывания. На рис. 4.5 представлены два момента срабатывания таймера для частоты 1 Гц.

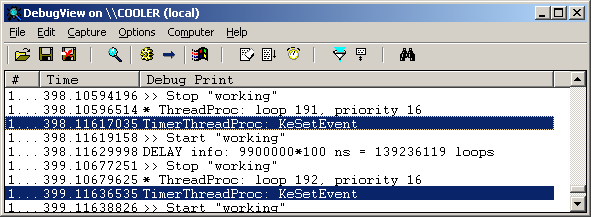


Рис. 4.5. Точность работы таймера

Как видно, разница составила 0.000195 с, что тоже можно считать хорошим результатом.

Посмотрим какие задержки мы сможем использовать в случае более высоких частот (до 1 кГц) и что будет если запустить драйвер не на «холостых оборотах» системы, а с наличием внешней нагрузки.

**4.6 Увеличение частоты срабатывания**

Драйвер был запущен для частоты 1 кГц. Увеличивая постепенно время задержки было получении предельное значение около 200 мкс. Как видно, увеличение частоты в 1000 раз привело к уменьшению длительности задержки более чем в 1000 раз. В принципе даже такого интервала задержки могло бы хватить для реализации системы управления реального времени, однако следует учесть еще один фактор. В реальной системе помимо нашего драйвера выполняются еще и другие приложения, которые тоже требуют процессорного времени и других системных ресурсов. Проверим работу драйвера в присутствии таких приложений.

**4.7 Работа параллельно с другими приложениями**

Для проведения исследования этого рода было написано специальное приложение, которое может создавать нагрузку определенного рода.

**4.7.1 Нагрузка на подсистему GDI**

Известно, что подсистема GDI Windows имеет свой системный поток с приоритетом реального времени. Можно ожидать, что при большом количестве запросов к этой подсистеме может вызвать сбой в работе нашего драйвера. Проверим это.

Драйвер был запущен на частоте 1 кГц параллельно с приложением, выполняющим активное рисование средствами GDI. Стабильная работа драйвера сохранилась, однако графический вывод производился с заметными «рывками».

**4.7.2 Работа со страничными отказами**

Серьезной проблемой могут стать страничные отказы, происходящие в системе. Запуск на частоте 1 кГц параллельно с приложением, генерирующем множественные страничные отказы дал очень интересные результаты. Драйвер не смог выдержать временные задержки в 200 мкс. Более того, даже более короткие задержки (до 25 мкс) не давали устойчивой работы драйвера.

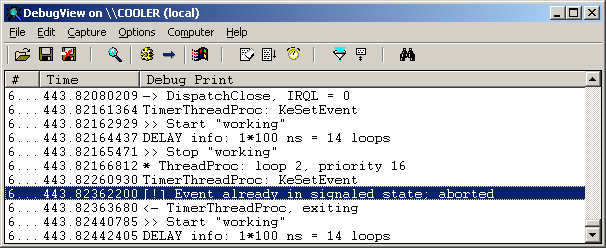


Рис 4.6. Остановка при обнаружении сбоя

Ситуация меняется к лучшему при увеличении приоритета потока драйвера. Можно предположить, что обработка страничных отказов выполняется также при помощи системного потока. Если мы устанавливаем свой приоритет выше его, то стабильность работы повышается. На приоритете 30 и частоте 1 кГц удалось добиться стабильной работы при страничных отказах с задержками до 100 мкс.

**5. Заключение**

Мы пронаблюдали работу драйвера, предназначенного для реализации системы реального времени, содержащего в себе системный поток, выполняющийся с приоритетом реального времени.

После проведения некоторого количества экспериментов были получены допустимые времена задержек для частоты 1 кГц в «холостом» режиме работы системы и в режиме множественных страничных отказов. Для тестовой системы в первом случае время составило 200 мкс, а во втором 100 мкс.

Теоретический предел для частоты 1 кГц очевидно 1000 мкс, т.е. для полезной обработки данных внешнего устройства могут быть использованы 10—20% процессорного времени.

При соблюдении этих условий на операционной системе Windows может быть реализована система управления реального времени предложенного типа.

**Приложение 1**

**Исходный код управляющего потока**

VOID TimerThreadProc(PVOID startContext)

{TRACE("-> TimerThreadProc, IRQL = %d",

KeGetCurrentIrql());

KeSetPriorityThread((PKTHREAD)pTimerThread,31);

KPRIORITY priority =

KeQueryPriorityThread((PKTHREAD)pTimerThread);

TRACE("TimerThread Priority: %d",priority);

SetTimer();

while(!bExitThread)

{KeWaitForSingleObject(&kTimer,

Executive,KernelMode,FALSE,0);

if (bExitThread) break;

LONG state = KeReadStateEvent(&kEvent);

if (state == 0)

{TRACE("TimerThreadProc: KeSetEvent");

KeSetEvent(&kEvent,FALSE,FALSE);}

else

{TRACE("[!] Event already in signaled "

"state; aborted");

bTimerWorks = false;

break;}}

KillTimer();

TRACE("<- TimerThreadProc, exiting");

PsTerminateSystemThread(STATUS\_SUCCESS);}

**Приложение 2**

**Исходный код рабочего потока**

VOID ThreadProc(PVOID startContext)

{TRACE("-> ThreadProc, IRQL = %d",KeGetCurrentIrql());

KeSetPriorityThread((PKTHREAD)pThread,curSett.Priority);

for(int i = 0;!bExitThread;i++)

{TRACE("\* ThreadProc: loop %d, "

"priority %d",i,

KeQueryPriorityThread((PKTHREAD)pThread));

KeWaitForSingleObject(&kEvent,Executive,KernelMode,

FALSE,0);

if (bExitThread) break;

// имитируем работу

TRACE(">> Start \"working\"");

DELAY(curSett.Delay);

TRACE(">> Stop \"working\"");

// сбрасываем событие чтобы снова начать его ждать

KeResetEvent(&kEvent);}

TRACE("<- ThreadProc, exiting");

PsTerminateSystemThread(STATUS\_SUCCESS);}