Основы написания драйвера уровня ядра для ОС Windows 2000, XP и XP Embedded

Валерий Яковлев

Статья посвящена практическим аспектам написания драйверов уровня ядра в операционных системах семейства NT — Windows 2000/XP (XP Embedded). Для понимания работы подсистемы ввода-вывода, составляющим звеном которой являются драйверы, дано краткое описание структурной организации ОС в целом. Приводится рабочий пример простейшего драйвера уровня ядра.

Введение

Современный уровень аппаратных средств, используемых в автоматизации, давно «размыл» чёткую границу между компьютерами верхнего уровня и контроллерами полевого уровня, с точки зрения используемого системного программного обеспечения. Аппаратные ресурсы современных контроллеров позволяют использовать на них не только оптимизированные к минимальным аппаратным требованиям операционные системы, например Windows CE .NET, но и стандартные ОС класса Windows или Linux. Кроме того, из встраиваемых, но более ресурсоёмких операционных систем, всё более активно применяющихся на рынке автоматизации отечественной промышленности, можно назвать Embedded Windows XP, общая структурная организация которой во многом схожа с организацией стандартного (офисного) варианта. Возможность «конструирования» разработчиком образа этой ОС из большого числа компонентов не влияет на основные принципы общей организации ОС в части затрагиваемых в этой статье вопросов, поэтому рассматриваемые здесь возможности написания драйверов уровня ядра применимы и для этой ОС. На текущий момент доминирующее положение среди стандартных ОС Windows в сфере автоматизации занимают ОС Windows 2000/XP, поэтому дальнейшее изложение материала касается именно этих ОС. Использование стандартных ОС позволяет максимально использовать опыт программирования, накопленный специалистами, привыкшими работать со знакомым набором API (Application Programming Interface — интерфейс прикладного программирования). В числе решаемых программистами задач, кроме собственно написания прикладной программы, может возникнуть необходимость в написании драйвера, например, если по каким-либо причинам драйверы под ОС Windows не поставляются (например, платы ввода-вывода в формате MicroPC фирмы Octagon Systems или Fastwel) или речь идёт о плате собственной разработки. Несмотря на большое количество технической литературы, посвящённой программированию в ОС Windows, проблемы написания драйверов практически не освещаются. Безусловно, этот вопрос достаточно специфичен для программистов общего плана, но для специалистов, работающих в сфере автоматизации производства, эта тема представляет большой интерес. Даже если программисту не придётся непосредственно заниматься написанием драйверов, более глубокое понимание функционирования этой подсистемы ОС Windows даст более чёткое представление о возможностях функционирования ОС в целом. Это позволяет трезво оценивать временны́е возможности стандартных драйверов и при необходимости получения лучших временны́х показателей проектируемой системы в целом написать свой, возможно менее функциональный, но более быстрый драйвер. Тема драйверов достаточно обширна и не укладывается в рамки одной статьи, поэтому мы ограничимся минимально необходимым общим материалом по структуре Windows 2000/XP и затем решим прикладную задачу – напишем простейший драйвер, осуществляющий доступ к регистрам платы, установленной в слот ISA (пусть это будет модуль UNIOXX-5 фирмы Fastwel). Выбор устройства на шине ISA обусловлен более простой схемой программного взаимодействия с таким устройством, на самом деле это может быть и плата, установленная в слот PCI, или любой другой системный ресурс, обращение к которому на уровне регистров запрещено из программ режима пользователя. При этом усложняется алгоритм взаимодействия с устройством, но принцип доступа через порты ввода-вывода остаётся. Драйвер напишем на языке ассемблера, который наиболее эффективно используется именно при написании драйверов, где требования к минимизации кода и, как следствие, малое время исполнения объективно необходимы.

Необходимые инструменты программирования

Для решения поставленной задачи используем пакет MASM32 v.8.2, а также комплект разработки драйверов (Windows 2000 Driver Development Kit, DDK), который можно бесплатно скачать с сайта Microsoft (www.microsoft.com/ddk/). К сожалению, фирма Microsoft пакет DDK для ОС Windows XP не выкладывает для свободного доступа, предлагая его получить по почте на CD-носителе. И хотя при этом оплачивается только доставка, получение пакета затруднительно. В комплект DDK входят документация (папка help), являющаяся наиболее полным источником информации о внутренних структурах данных и внутрисистемных функциях используемых драйверами устройств, включаемые файлы (*.inc) из папки inc, примеры реализаций драйверов устройств (папка src), утилиты (папка tools). Главное, в DDK входит набор библиотечных файлов (*.lib), необходимых при компоновке законченных драйверов и нашего простейшего драйвера в том числе. В DDK есть два комплекта этих файлов — checked build и free build, нам необходим free build (папка libfre) для окончательной версии Windows, называемой свободным выпуском. Функции из библиотечных файлов (папка libchk) отладочной версии (checked build) отличаются более строгой проверкой ошибок. Но, увы, не всё так хорошо для прямого использования включаемых файлов, содержащих определения прототипов функций, а также необходимых структур, символьных констант и макросов. Есть одна проблема, заключающаяся в том, что все описания в DDK даны из расчёта использования языка C, в то время как мы хотим воспользоваться «великим и могучим» ассемблером. Если ещё есть утилиты, позволяющие автоматизировать получение inc-файлов для имеющихся lib-файлов, то формирование inc-файлов, содержащих структуры и константы,— это в основном ручной труд. Эту работу в необходимом объёме придётся проделать самостоятельно. Она не из категории творческих, но в Интернете совершенно бесплатно можно найти достаточно полные версии конвертированных для использования с ассемблером inc-файлов, и таким образом можно начать не с нуля [1].

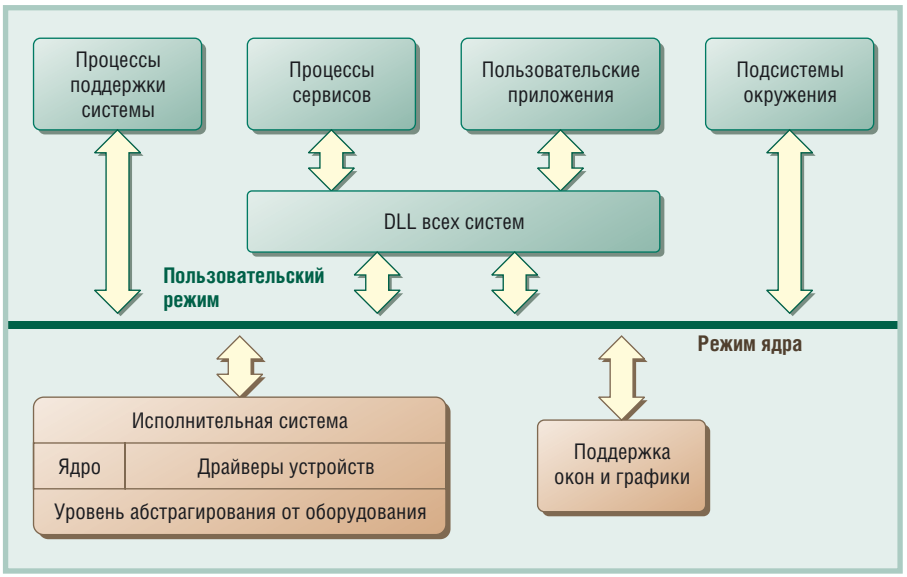

Рис. 1. Упрощённая схема архитектуры Windows

Краткий обзор архитектуры Windows

В Windows 2000/XP существует чёткое разграничение двух областей в оперативной памяти и режимов процессора для исполняемого кода: 1) область исполняемого кода в непривилегированном режиме работы процессора (пользовательском режиме) для приложений пользователя и части компонентов ОС, и область исполняемого кода операционной системы в привилегированном режиме процессора (режиме ядра). Под областью исполняемого кода надо понимать области загрузки (диапазон адресов) в оперативной памяти вычислительной системы. Windows 2000/XP — 32-разрядная ОС (существующую 64-битовую версию этой ОС в данной статье не рассматриваем), и поэтому всем приложениям доступно до 4 Гбайт линейного адресного пространства. Чаще всего в системе установлен существенно меньший объём физической памяти, но тем не менее для работающих программ это незаметно. Специальные системные механизмы обеспечивают возможность виртуального присутствия 4 Гбайт памяти в системе. Деление 4 Гбайт виртуального (или не виртуального, если Вы можете себе это позволить) адресного пространства между пользовательскими приложениями и системными программами осуществляется поровну: первые 2 Гбайт пользовательские, остальное — системное адресное пространство. К ограничению свободы трудно привыкнуть, но, как и в наличии законов в стране, в этом есть элемент вековой мудрости, накопленной человечеством. Исполняемый код в пользовательском режиме имеет ограничения на доступ к системным ресурсам, в частности, на прямой доступ к оборудованию. Это связано с желанием обеспечить более устойчивое функционирование системы при наличии ошибок в программах пользователей. Надо учитывать, что Windows проектировалась как многозадачная и многопользовательская система, поэтому крах одного приложения не должен приводить к краху ОС и, следовательно, к краху других пользовательских приложений, запущенных на исполнение в этой системе. Приложения ОС и другие программы, исполняющиеся в режиме ядра, имеют полный доступ ко всем ресурсам системы. Упрощённая схема архитектуры Windows [2] приведена на рис. 1.

Как уже отмечалось, в режиме пользователя функционируют не только прикладные программы пользователя, но и часть процессов самой ОС.

К компонентам операционной системы, работающим в режиме пользователя, относятся:

- некоторые процессы поддержки системы, например процесс обработки входа в систему (Winlogon);
- процессы Windows-сервисов (о сервисах поговорим чуть позже). В виде сервисов оформлены как некоторые системные сервисы (например Task Scheduler), так и отдельные компоненты прикладных программ, например Microsoft SQL Server, а также некоторые драйверы;
- пользовательские приложения. На текущий момент они бывают шести типов: Win32, Win64 (в 64-битовой

версии ОС), Windows 3.1, MS-DOS, POSIX и OS/2;

- подсистемы окружения. Это часть операционной системы (программные оболочки), предоставляющая приложениям пользователя определённый для конкретной подсистемы набор функций. Windows обеспечивает работу с тремя подсистемами окружения: Win32, POSIX и OS/2. Windows 2000 поставляется с двумя подсистемами, а в Windows XP, кроме Win32, не поставляются другие подсистемы окружения.

К компонентам ОС, работающим в режиме ядра, относятся:

- исполнительная система, обеспечивающая базовыми сервисами в части управления памятью, процессами и потоками, вводом-выводом и т.д.;
- ядро, которое содержит обобщённый набор функций ОС, скрывающий различия между аппаратными платформами (на разных этапах развития ОС NT поддерживались не только процессоры Intel, но и MIPS, Alpha AXP, Motorola PowerPC). Ядро предоставляет процедуры/функции и базовые объекты, используемые исполнительной системой и драйверами для реализации структур и функций более высокого уровня. К таким функциям относятся планирование потоков, диспетчеризация прерываний, синхронизация процессов и т.д.;
- драйверы устройств;
- уровень абстрагирования от оборудования (Hardware Abstraction Layer, HAL) — набор низкоуровневых функций (около 92), обеспечивающий стандартный интерфейс взаимодействия с аппаратно-зависимыми элементами для функций, вызываемых компонентами ядра, драйверов и исполнительной системы, позволяющий абстрагироваться от того, на какой конкретно элементной базе (чипе контроллера прерывания, контроллера ПДП) реализовано выполнение доступа к шине, таймеру и т.д.;
- подсистема поддержки окон и графики.

Драйверы устройств в Windows, в отличие от DOS, для поддержки переносимости не обращаются к оборудованию напрямую, а используют функции, предоставляемые HAL. Драйверы устройств режима ядра делятся на следующие основные категории:

- драйверы файловой системы (например сетевые редиректоры и серверы). Не стоит понимать буквально, что речь идёт только о файловой системе ОС. На самом деле многие физические устройства (например COM-порты) представляются в системе как файлы, и обращение к ним осуществляется посредством вызова функций, как к обычным файлам, но со специфическими параметрами. Далее уже драйверы файловой системы, получившие запрос на ввод-вывод, определяют, о каком устройстве идёт речь, и вызывают соответствующие физическому устройству драйверы следующего уровня;
- драйверы с поддержкой Plug-and-Play (PnP) и ACPI (advanced configuration power-management interface — усовершенствованный интерфейс управления конфигурацией и энергопотреблением);
- драйверы, не поддерживающие спецификации PnP и ACPI (например драйверы протоколов TCP/IP, IPX/SPX и т.д.), которые расширяют функциональность системы, предоставляя доступ из режима пользователя к системным сервисам и драйверам режима ядра.

В свою очередь, в каждой из категорий есть группы драйверов, которые различаются в зависимости от модели устройства и места драйверов в цепочке обработки запроса на обслуживание операций ввода-вывода.

Начиная с Windows 2000, была введена поддержка PnP и энергосберегающих технологий (ACPI), что привело к расширению модели драйверов, называемой Windows Driver Model (WDM); напомню, что речь идёт о линейке NT, модель драйверов WDM ранее реализована в Windows 98 и Windows Millennium Edition. ОС Windows 2000 и более поздние версии ОС линейки NT поддерживают и так называемые унаследованные драйверы (NT4), естественно, с некоторой потерей функциональности.

Модель WDM предусматривает существование трёх типов драйверов:

- драйвер шины. Интересным моментом является то, что, в отличие от ОС NT4, Windows 2000 и выше, позволяют реализовать поддержку новых типов шин, не поддерживаемых самой ОС, не путём создания своего HAL (DLL), а всего лишь добавлением своего драйвера шины. Это крайне существенно для поставщиков OEM-оборудования;
- функциональный драйвер;
- драйвер фильтра.

В рамках обобщения понятия устройства в Windows существует понятие класса устройств. Введение этого уровня абстрагирования сопровождается неизбежным появлением типа драйверов, отвечающих за обслуживание устройств одного класса (например CD-ROM), и драйверов, отвечающих за решение того или иного уровня взаимодействия с конкретным оборудованием. В рамках этого деления существуют драйверы:

- классов устройств;
- порт-драйверы;
- минипорт-драйверы.

Драйверы устройств в ОС Windows могут работать как в режиме ядра, так и в пользовательском режиме. К последним относятся:

- драйверы виртуальных устройств (VDD);
- драйверы принтеров.

Важнейшим компонентом исполнительной системы, отвечающим за связь с устройствами, является подсистема ввода-вывода. Построение подсистемы ввода-вывода, как и других компонентов ОС Windows призвано обеспечить максимальную устойчивость ОС в целом. Поэтому в соответствии с общей доктриной разделения ответственности, связанной с режимами работы в ОС Windows, приложения пользователя не могут обращаться к устройствам (читай — драйверам) напрямую, а лишь через посредников в лице диспетчеров. Некоторые компоненты подсистемы и диспетчеры, как, например, диспетчер Plug-and-Play, работают как в пользовательском режиме, так и в режиме ядра, но в целом вся подсистема и, в частности, её главный компонент — диспетчер ввода-вывода работает в режиме ядра. В некотором промежуточном положении (с точки зрения доступности из разных режимов) оказываются inf- и cat-файлы (хранят цифровые подписи, удостоверяющие аттестацию лаборатории Microsoft WHQL — Microsoft Windows Hardware Quality Lab) и реестр. Диспетчер ввода-вывода не только обеспечивает взаимосвязь между приложениями пользователя и драйверами устройств, но также предоставляет общий для драйверов код, используемый при обработке запросов, что существенно влияет на минимизацию кода самих драйверов. Он также обеспечивает управление буферами запросов ввода-вывода и при необходимости вызовы

одним драйвером других драйверов для организации обработки запроса по цепочке. Упрощённая схема организации подсистемы ввода-вывода [2] изображена на рис. 2. Подсистема ввода-вывода Windows проектировалась с целью обеспечения максимальной гибкости, как с точки зрения возможности её расширения драйверами специфических устройств, так и с учётом поддержки максимального абстрагирования устройств для прикладных приложений. Важными моментами обеспечения подобной функциональности являются возможности динамической загрузки (явной или на основе перечисления) и выгрузки драйверов, обобщённый вид формируемых структур запросов на ввод-вывод и диспетчеризация. Одним из инструментов регистрации, запуска, останова и выгрузки драйверов служит механизм управления сервисами.

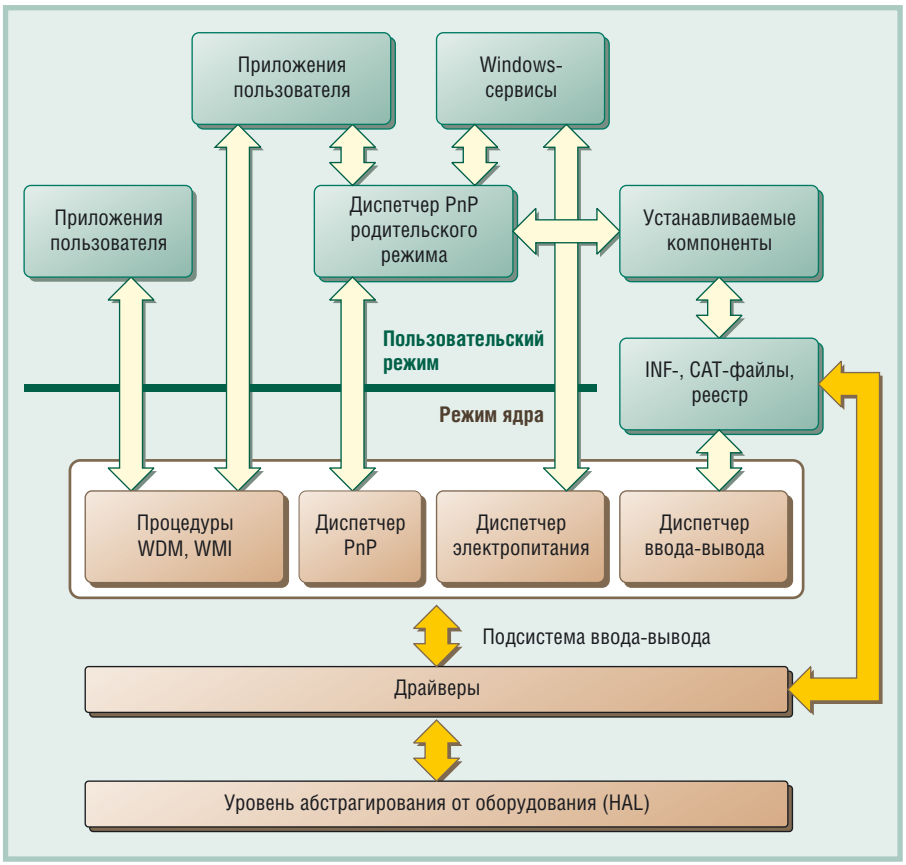

Рис. 2. Компоненты подсистемы ввода-вывода

Сервисы

Сервисы (Services), или службы, являются процессами, предоставляющими дополнительную функциональность в системе, не зависящую от интерактивных действий пользователя. То есть это приложения, запускаемые без учёта того, зарегистрировался ли в системе какой-либо пользователь или нет, и довольно часто запуск самого сервиса происходит до момента появления окна регистрации пользователя. В части взаимодействия с системой сервисы используют «язык» Windows API и функционально состоят из трёх компонентов:

- Service application — сервисное приложение (или драйвер);
- Service control program (SCP) — программа управления сервисом;
- Service Control Manager (SCM) — диспетчер управления сервисом.

Весь необходимый API для реализации механизма диспетчеризации (SCM-функции) сервисов сосредоточен в системной DLL-библиотеке Advapi32.dll (Advanced API).

Сервисное приложение — это драйвер или обычное Windows-приложение (чаще всего реализуемое как консольное приложение), имеющее дополнительный блок кода, обеспечивающий обработку команд от SCP и возврат ему определённой статусной информации.

SCP — стандартное Windows-приложение, использующее для управления сервисом функции из набора SCM-функций. В основном SCP-приложение предназначено для запуска, останова и конфигурирования сервиса, иногда это приложение расширяет политику управления сервисом по отношению к реализуемой SCM-функции. В рамках ОС Windows существуют встроенные SCP, но для увеличения возможностей конфигурирования сервиса разработчик может написать свою собственную программу SCP, возможно, даже не реализуя её как отдельное приложение, а встраивая её функции в основное приложение, использующее сервис. Сервисные приложения только косвенно взаимодействуют с SCP, изменяя свою статусную информацию в процессе выполнения команд от SCM. Перед использованием SCM-функций SCP необходимо установить канал связи с SCM (используется функция OpenSCManager). Важным моментом является то, что при установке любого приложения, в рамках которого предполагается использование сервиса, необходимо предварительно зарегистрировать сервис, вызвав функцию CreateService (реализована в Advapi32.dll). На основании параметров вызова этой функции SCM создаёт для каждого сервиса индивидуальный раздел в ветке реестра HKEY_LOKAL_MACHINE\SYSTEM\CurrentControlSet\Service (основная информационная база SCM, или ServicesActive database), куда помещается как параметр вновь созданного раздела реестра информация о месте расположения на диске исполняемого файла соответствующего сервиса, его параметры при старте и конфигурационные настройки (эти настройки помещаются в подраздел Parameters созданного для сервиса раздела).

SCM отвечает за загрузку как сервисов, так и драйверов, поэтому не удивительно, что значения некоторых параметров явным образом указывают на контекст текущего приложения. Общий термин (Services), используемый Microsoft для обозначения как служб, так и драйверов устройств (они действительно во многом схожи по стилю работы и характеру выполняемых функций), часто вызывает путаницу в понимании программной документации. Наиболее важные для нашей прикладной задачи параметры созданного раздела реестра и их возможные значения для драйверов и сервисов показаны в таблице 1.

На конечном этапе загрузки ОС системный процесс Winlogon (перед появлением диалогового окна с при-

Таблица 1

Основные параметры при регистрации сервиса/драйвера и их возможные значения

Параметр	Значение		Описание
Start	SERVICE_BOOT_START	(0)	Драйвер загружается Ntldr или Osloader, то есть в период загрузки самой ОС
	SERVICE_SYSTEM_START	(1)	Драйвер загружается после загрузки и инициализации драйверов со значением
	SERVICE_AUTO_START	(2)	Драйвер или сервис запускается SCM автоматически после запуска SCM-процесса (Services.exe)
	SERVICE_DEMAND_START	(3)	Драйвер или сервис запускается SCM по требованию
	SERVICE_DISABLED	(4)	Драйвер или сервис не загружается и не инициализируется
ErrorControl	SERVICE_ERROR_IGNORE	(0)	Код ошибки, возвращаемый драйвером или сервисом, игнорируется
	SERVICE_ERROR_NORMAL	(1)	Если драйвер или сервис возвращает код ошибки, выводится предупреждение
	SERVICE_ERROR_SEVERE	(2)	Если драйвер или сервис возвращает ошибку и ещё не использовалась последняя удачная конфигурация, то используется последняя удачная конфигурация. Если используется именно она, то загрузка продолжается.
	SERVICE_ERROR_CRITICAL	(3)	Если драйвер или сервис возвращает ошибку и ещё не использовалась последняя удачная конфигурация , то используется последняя удачная конфигурация. Если используется именно она, то выводится BSOD «Синий экран смерти».
Type	SERVICE_KERNEL_DRIVER	(1)	Драйвер устройства режима ядра
	SERVICE_FILE_SISTEM_DRIVER	(2)	Драйвер файловой системы
ImagePath	Путь к исполняемому файлу		Путь к исполняемому файлу драйвера или сервиса. По умолчанию файл ищется в папке %SystemRoot%\System32\Drivers
DisplayName	Имя сервиса		Имя сервиса. По умолчанию имя его раздела в реестре.

глашением к регистрации) запускает SCM, который в созданной внутренней базе данных сервисов SCM ищет записи драйверов и сервисов с параметром Start, имеющим значение SERVICE_AUTO_START, и запускает их.

Далее рассмотрим пример реализации SCP на языке ассемблера (листинг 1). Минимально необходимый материал, позволяющий понять правила и нотацию для оформления программ на этом языке, можно найти в [3].

Действия программы заключаются в следующем. Устанавливается канал связи с SCM. Если при установлении канала связи происходит ошибка, выводится сообщение “Attempt of connection with SCM has failed!” и программа завершается. В случае установления канала регистрируем драйвер. Если происходит ошибка регистрации, выводим соответствующее сообщение “Attempt to register the driver has failed!” и, закрыв дескриптор SCM, завершаем программу. В случае успеха регистрации драйвера запускаем драйвер, удаляем его, закрываем дескриптор драйвера, закрываем дескриптор SCM, завершаем программу. В этой программе драйвер запускается один раз, и поэтому его действия носят характер, только подтверждающий факт его работы. Теперь более подробно.

Для вызова SCM-функций SCP должна установить канал связи с SCM, используя функцию OpenSCManager. Прототип функции выглядит следующим образом:

OpenSCManager proto
lpMachineName:LPSTR,
lpDatabaseName:LPSTR,
wDesiredAccess:DWORD

lpMachineName
Указатель на строку, завершающуюся нулём, содержащую имя компьютера. Устанавливая этот параметр в NULL, мы устанавливаем связь с локальным SCM (на этом компьютере).

lpDatabaseName
Указатель на строку, завершающуюся нулём, содержащую имя открываемой базы. Устанавливая этот параметр в NULL, мы устанавливаем связь с локальной, активной в текущий момент базой данных – SERVICES_ACTIVE_DATABASE.

dwDesiredAccess
Права доступа, запрашиваемые при открытии канала. Могут быть следующие значения:
SC_MANAGER_CONNECT (устанавливаются по умолчанию, параметр 0);
SC_MANAGER_CREATE_SERVICE (доступ для внесения в базу данных записи о новом драйвере);
SC_MANAGER_ALL_ACCESS (полный доступ).

Если эта функция возвращает не NULL (NULL говорит об ошибке), то мы получаем дескриптор активной базы данных. Следующий шаг — это регистрация нашего драйвера путём вызова функции CreateService, прототип функции выглядит так:

CreateService proto
hSCManager:HANDLE,
lpServiceName:LPSTR,
lpDisplayName:LPSTR,
dwDesiredAcces:DWORD,
dwServiceType:DWORD,
dwStartType:DWORD,
dwErrorControl:DWORD,
lpBinaryPathName:LPSTR,
lpLoadOrderGroup:LPSTR,
lpdwTagId:LPDWORD,
lpDependencies:LPSTR,
lpServiceStartName:LPSTR,
lpPassword:LPSTR

hSCManager
Дескриптор базы данных SCM.

lpServiceName
Указатель на строку, завершающуюся нулём, содержащую имя драйвера/сервиса. Длина до 256 символов. Соответствует имени подраздела в реестре.

lpDisplayName
Указатель на строку, завершающуюся нулём, содержащую экранное имя драйвера/сервиса. Длина до 256 символов. Соответствует значению параметра DisplayName в реестре.

dwDesiredAcces
Запрашиваемый тип доступа. Могут быть значения:
SERVICE_ALL_ACCESS (полный доступ); SERVICE_START (доступ на запуск драйвера/сервиса);

Листинг 1

```
; Пример простой программы управления сервисом (SCP)

.386
.model flat, stdcall
option casemap:none

include D:\masm32\INCLUDE\kernel32.inc
include D:\masm32\INCLUDE\user32.inc
include D:\masm32\INCLUDE\advapi32.inc
include D:\masm32\INCLUDE\windows.inc
includelib D:\masm32\LIB\kernel32.lib
includelib D:\masm32\LIB\user32.lib
includelib D:\masm32\LIB\advapi32.lib

.data
hSCManager         dd 0
hService           dd 0
ALIGN              4
CardUNIODriverPath db "D:\masm32\BIN\CardUNIO.sys",0
ALIGN              4
ErrMsg1            db "Attempt of connection with SCM has failed!",0
ALIGN              4
DrvName            db "CardUNIO.sys",0
ALIGN              4
ErrMsg2            db "Attempt to register the driver has failed!",0
ALIGN              4
DrvName1           db "CardUNIO",0
ALIGN              4
DispNameDrv        db "MyDriver",0

.code
 start proc

; Устанавливаем канал связи с SCM
     invoke OpenSCManager, NULL, NULL, SC_MANAGER_CREATE_SERVICE
     .if eax != NULL
         mov hSCManager, eax
; Регистрируем драйвер
     invoke CreateService, hSCManager, offset DrvName1, \
     offset DispNameDrv, \
     SERVICE_START + DELETE, SERVICE_KERNEL_DRIVER, \
     SERVICE_DEMAND_START, \
     SERVICE_ERROR_IGNORE, addr CardUNIODriverPath, \
     NULL, NULL, NULL, NULL, NULL
         .if eax != NULL
             mov hService, eax
; Запускаем драйвер
             invoke StartService, hService, 0, NULL
; Удаляем драйвер
             invoke DeleteService, hService
; Закрываем дескриптор драйвера
             invoke CloseServiceHandle, hService
         .else
; Если не удалось зарегистрировать сервис, выводим об этом сообщение
             invoke MessageBox, NULL, offset ErrMsg2, NULL,
  MB_ICONSTOP
         .endif
; Закрываем канал связи с SCM
         invoke CloseServiceHandle, hSCManager
     .else
; Сообщение в случае невозможности установить связь с SCM
     invoke MessageBox, NULL, offset ErrMsg1, NULL, MB_ICONSTOP
     .endif

     invoke ExitProcess, 0

 start endp

 end start
```

SERVICE_STOP (доступ на останов драйвера/сервиса); DELETE (доступ на удаление драйвера/сервиса из базы SCM).

dwServiceType

Тип сервиса, в нашем случае SERVICE_KERNEL_DRIVER. Соответствует значению параметра TYPE в реестре.

dwStartType

Тип запуска, в нашем случае SERVICE_DEMAND_START. Соответствует значению параметра START в реестре.

dwErrorControl

Характер контроля ошибок, в нашем случае SERVICE_ERROR_IGNORE. Соответствует значению параметра ErrorControl в реестре.

lpBinaryPathName

Указатель на строку, завершающуюся нулём, содержащую полный путь к файлу загружаемого драйвера. Соответствует значению параметра ImagePath в реестре.

lpLoadOrderGroup

Указатель на строку, завершающуюся нулём, содержащую имя группы, в случае если загружаемый драйвер является членом группы. В противном случае значение NULL или указатель на пустую строку.

lpdwTagId

Указатель на переменную, содержащую уникальное значение тега, идентифицирующее группу. В противном случае NULL.

lpDependencies

Указатель на массив имен драйверов или групп драйверов, загрузка которых должна быть осуществлена до запуска текущего драйвера. Массив заканчивается двумя нулями, имена драйверов или групп разделяются одним нулём. Если запуск драйвера не связан с предварительным запуском других драйверов, значение NULL.

lpServiceStartName

Указатель на строку, завершающуюся нулём, содержащую имя учётной записи (account), с правами которой запускается текущий драйвер. В случае типа сервиса SERVICE_KERNEL_DRIVER параметр содержит имя объекта драйвера. Если используется имя объекта драйвера, присвоенное подсистемой ввода-вывода, то NULL.

lpPassword

Указатель на строку, завершающуюся нулём, содержащую пароль учётной записи, с правами которой запускается текущий драйвер. В случае SERVICE_KERNEL_DRIVER значение этого параметра игнорируется.

Остальные функции — StartService, DeleteService и CloseServiceHandle — запускают, удаляют и закрывают дескриптор нашего драйвера. Далее приводятся прототипы функций и описания их параметров.

StartService proto
hService:HANDLE,
dwNumServiceArgs:DWORD,
lpServiceArgVectors:LPSTR

hService
Дескриптор драйвера/сервиса.

dwNumServiceArgs
Количество аргументов, передаваемых сервису. Драйверу не передаются аргументы, поэтому значения NULL.

lpServiceArgVectors
Указатель на массив указателей, ссылающихся на строки, завершающиеся нулём. В строках содержатся передаваемые службе аргументы. В нашем случае аргументы отсутствуют, поэтому значение NULL.

DeleteService proto *hService*:HANDLE

hService
Дескриптор удаляемой службы.

CloseServiceHandle proto
hSCObject:HANDLE

hSCObject
Дескриптор закрываемого SCM, сервиса, драйвера.

Прототип драйвера режима ядра

Рассмотрим шаблон простейшего драйвера (листинг 2).

Не правда ли, если Вы знакомы с динамически подключаемой библиотекой (DLL) [3], то приведённый пример во многом напомнит Вам структуру простейшей динамической библиотеки. Надо сказать, что между драйверами и DLL очень много общего. После загрузки драйвера операционная система передаёт управление на его точку входа. Предполагается, что выполняемая функция программы, которой передано управление сразу после загрузки драйвера, — это инициализация структур и переменных, необходимых для дальнейшей работы драйвера, и отчёт перед ОС о выполненной задаче ("DriverEntry is the first routine called after a driver is loaded, and is responsible for initializing the driver", — читаем мы в MSDN). Точкой входа является метка, указанная после директивы End, то есть в нашем случае это EntryDriverUNIO (у Вашего драйвера может быть другое имя). В нашем примере после передачи управления на точку входа происходит формирование перехода из 0 в 1 на одном из выходов FPGA1-платы UNIOXX-5

Листинг 2

```
; CardUNIO.asm
.386
.model flat, stdcall
option casemap:none

include D:\masm32\INCLUDE\DDK\wxp\ntddk.inc

UNIO_FPGA1_BaseAdrr     equ 0a110h
; базовый адрес FPGA1 UNIOxx-5 при поставке

MyDelay                  equ 100h
STATUS_IO_DEVICE_ERROR equ 0C00000185h

.code
ShortDelay proc ValDelay:DWORD
             push cx
             mov cx, ValDelay
@@delay:
             dec cx
             jnz @@delay
             pop cx
             ret

ShortDelay  endp

EntryDriverUNIO proc DriverObject:PDRIVER_OBJECT, \
  RegistryPath:PUNICODE_STRING

             push cx
             push dx

             xor al,al
             inc al
             mov dx, UNIO_FPGA1_BaseAdrr
             ; в контрольном регистре устанавливаем BANK = 1
             cli
             out dx,al
             sti
             invoke ShortDelay, MyDelay
             ; в регистрах маски каналы 0-23 на вывод
             inc dx
             mov al,0ffh
             cli
             out dx,al
             sti
             invoke ShortDelay, MyDelay
             xor al,al
             dec dx
             ; в контрольном регистре устанавливаем BANK = 0
             cli
             out dx,al
             sti
             invoke ShortDelay, MyDelay
             inc dx
             ; формируем импульс на 0-7 выходах
             Cli
             Out dx,al
             invoke ShortDelay, MyDelay
             mov al,0ffh
             out dx,al
             invoke ShortDelay, MyDelay
             xor al,al
             out dx,al
             ; возвращаем системе сообщение о неудачной
             ; инициализации драйвера
             mov eax, STATUS_IO_DEVICE_ERROR
             pop dx
             pop cx
             ret
EntryDriverUNIO endp

End EntryDriverUNIO
```

(используется стандартная прошивка g00). Реальность отработки драйвера можно наблюдать, подключив осциллограф или светодиод к соответствующему выходу платы. Естественно, для визуализации работы драйвера можно было обойтись и без платы, например, вывести какое-либо сообщение, или озвучить это событие на динамике ПК. Затем в регистр eax загружается возвращаемый параметр (STATUS_IO_DEVICE_ERROR), тем самым системе сообщают об ошибке. На этом работа (и «жизнь») драйвера завершается. Так как текущий пример является только шаблоном драйвера, каких-либо конструктивных действий в драйвере не производится. Код ошибки выбран произвольно, его использование имело целью сообщить системе о том, что работа драйвера не может быть продолжена и его необходимо выгрузить, что операционная система и сделает. Но главного мы добились! Совершенно корректно, используя все «джентльменские» правила поведения в ОС Windows, мы получили полный доступ в святая святых — к системным ресурсам ОС, имея нулевой уровень привилегий для исполняемого кода. Просто жуть берёт, если представить, что можно натворить, бездумно воспользовавшись открывшимися возможностями. Можно свободно читать и записывать любые данные из внешних портов операторами IN и OUT, что было так привычно в DOS, записывать данные в любые адреса памяти, но одним неподходящим оператором или некорректным значением, записанным в порт или операцию память, можно получить «синий экран смерти» (BSOD) на Вашем мониторе. Большие возможности прежде всего предполагают и большую ответственность. Вот уж где поговорка «Семь раз отмерь, один раз отрежь!» очень актуальна.

Рассмотрим прототип DriverEntry (у нас это процедура EntryDriverUNIO).

DriverEntry proto
DriverObject:PDRIVER_OBJECT,
RegistryPath:PUNICODE_STRING

DriverObject
Указатель на объект созданного драйвера. Если говорить проще, то речь идёт о структуре типа DRIVER_OBJECT, описанной в файле NTDDK.h DDK. Часть полей в этой структуре необходимо заполнить загруженному драйверу, именно по этой причине ему и передаётся указатель на эту структуру. В нашем примере мы не занимались этой работой, так как «не задерживаемся надолго». В полнофункциональном драйвере эту работу придётся проделать.

RegistryPath
Указатель на структуру типа UNICODE_STRING, содержащую указатель на UNICODE-строку, в которой содержится имя раздела в реестре с параметрами инициализации драйвера.

Надо особо подчеркнуть, что, в отличие от пользовательского режима, в режиме ядра ОС работает только со строками типа UNICODE_STRING. Теперь несколько слов о компиляции и компоновке драйвера. Строка компиляции достаточно традиционна и, если мы работаем с masm32, выглядит следующим образом:

```
ML /nologo /c /coff CardUNIO.asm
```

Опции компоновщика, естественно, отличаются от опций для стандартного исполняемого файла:

```
LINK /nologo /driver /base:0x1000
/out:CardUNIO.sys /subsistem:native
CardUNIO.obj
```

/driver
Выходной файл – драйвер.

/base:0x1000
Предопределённый адрес загрузки драйвера.

/out:CardUNIO.sys
Выходной файл драйвера должен иметь соответствующее расширение (по умолчанию — .exe).

/subsistem:native
Тип подсистемы, необходимый для работы выходного файла. Этого вопроса мы кратко касались в данной статье, в части, посвящённой обзору архитектуры Windows. Напомню, что речь шла о существовании трёх подсистем окружения: Win32, POSIX и OS/2. Параметр Native говорит о том, что нет необходимости ни в одной из этих подсистем. Драйвер работает в «родной» или естественной среде, то есть использует базовый API самой ОС Windows.

К сожалению, в статье из-за ограничений по объёму нет возможности привести и прокомментировать листинг ntddk.inc для создания простейшего драйвера. Файл будет выложен на сайте журнала «СТА».

Заключение, или Что впереди

В статье изложен минимально необходимый материал, для того чтобы обозначить основные направления развития темы написания драйверов в ОС Windows. Следующим шагом в создании полнофункционального драйвера должно быть создание объекта «устройство» функцией IoCreateDevice в фазе инициализации драйвера (в рамках процедуры EntryDriverUNIO) и достаточно полное изучение структуры DRIVER_OBJECT. Многие поля структуры — это просто указатели на процедуры обработки различных пакетов запросов на ввод-вывод (IRP), формируемых диспетчером ввода-вывода на основании вызовов из приложений режима пользователя соответствующих функций или внутрисистемных запросов. Нет необходимости сразу пытаться реализовать все возможные процедуры. Минимально необходимые процедуры, обеспечивающие обработку IRP, формируются на основании вызовов приложением пользователя функций CreateFile (открытие созданного драйвером устройства) и DeviceIoControl (задание устройству посредством драйвера определённых команд на выполнение). Учитывая, что ОС Windows XP в любых её ипостасях является одной из ведущих операционных систем на рынке автоматизации, есть смысл глубже изучить вопрос строения этой ОС и овладеть техникой написания драйверов для неё. Удачи Вам! ●

Литература

1. В. Пирогов. Ассемблер для Windows. — 3-е изд. — СПб. : БХВ-Петербург, 2005.
2. М. Руссинович, Д. Соломон. Внутреннее устройство Microsoft Windows: Windows Server 2003, Windows XP, Windows 2000. — 4-е изд. — М. : Русская Редакция, Питер, 2005.
3. Валерий Яковлев. Написание пользовательской DLL доступа к универсальному OPC-серверу Fastwel // Современные технологии автоматизации. 2005. № 3. С. 74-81.

Автор — сотрудник фирмы ПРОСОФТ
Телефон: (812) 448-0444
Факс: (812) 448-0339
E-mail: info@spb.prosoft.ru