

СОДЕРЖАНИЕ

ВВЕДЕНИЕ.....	5
1 ОСНОВНЫЕ ПОНЯТИЯ РАЗРАБОТКИ ДРАЙВЕРОВ	6
1.1 ОСНОВНЫЕ ПОНЯТИЯ	6
1.2 ИНСТРУМЕНТАРИЙ	9
2 СТРУКТУРНОЕ ПРОЕКТИРОВАНИЕ	12
3 ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ	14
4 РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ	23
5 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ	25
6 ТЕСТИРОВАНИЕ	27
7 ЗАКЛЮЧЕНИЕ	31
8 СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	32
ПРИЛОЖЕНИЕ А	33
ПРИЛОЖЕНИЕ Б	34

ВВЕДЕНИЕ

Драйверная концепция – неотъемлемая часть современных операционных систем. Эта концепция – основа взаимодействия системы (пользователя) с какими бы то ни было устройствами (системными/периферийными, реальными/виртуальными и т.д.). Даже системные программисты далеко не всегда имеют представление об этой концепции, о принципах ее работы, о программировании с использованием этой концепции. А ведь системное программирование – ключ к пониманию основ ИТ.

Написание драйверов – достаточно сложная, но, тем не менее, очень интересная и актуальная отрасль программирования. Знание особенностей технологии написания драйверов открывает огромное количество возможностей – написание драйверов для устройств, уже не поддерживаемых производителем, для устройств, драйвера к которым еще не написаны, исправление ошибок в драйверах, написание драйверов к различным промышленным устройствам и т.д.

У каждой операционной системы есть свои особенности, отсюда вытекает своя специфика написания драйверов под них. То же самое можно сказать и о разных типах оборудования.

Данная работа будет посвящена разработке драйвера файловой системы для ОС Windows 10.

1 ОСНОВНЫЕ ПОНЯТИЯ РАЗРАБОТКИ ДРАЙВЕРОВ

1.1 ОСНОВНЫЕ ПОНЯТИЯ

Изучение программирования драйверов – так же, как и изучение чего бы то ни было – нужно начинать с изучения теоретических основ. Так и поступим.

Прежде всего – базовые понятия. Итак, что такое драйвер? Драйвер – это часть кода операционной системы, отвечающая за взаимодействие с аппаратурой. В данном контексте слово “аппаратура” имеет самый широкий смысл. Под этим словом можно подразумевать как реальные физические устройства, так и виртуальные или логические. Но это уже подводит нас к вопросу о том, какие вообще бывают драйверы (и, соответственно, для каких устройств), а об этом мы поговорим позднее.

С момента своего появления до сегодняшнего дня драйвер непрерывно эволюционировал, и процесс этот до сих пор не закончился. Один из моментов эволюции драйвера – это эволюция концепции драйвера, как легко заменяемой части операционной системы. Как отдельный и довольно независимый модуль, драйвер сформировался не сразу. Да и сейчас многие драйверы практически неотделимы от операционной системы. Во многих случаях это приводит к необходимости переустановки системы (ОС Windows) или пересборки ее (ядра) (в UNIX-системах). Такое же различие есть и между ветками операционной системы Windows: Windows 9x и Windows NT. В первом случае процесс работы с драйверами происходит (практически всегда) как с отдельными “кирпичиками”, а во втором дела обстоят намного хуже (множество (если не большинство) драйверов “вшито” в ядро).

Список основных общих концепций драйверов в Windows- и UNIX-системах выглядит так:

- способ работы с драйверами как файлами;
- драйвер, как легко заменяемая часть ОС (учитывая сказанное выше);
- существование режима ядра.

Объясню подробнее первый пункт. Способ работы с драйверами как файлами означает, что функции, используемые при взаимодействии с файлами, практически идентичны таковым при взаимодействии с драйверами (имеется в виду лексически): **open**, **close**, **read** и т. д. О режиме ядра я расскажу позднее.

И напоследок стоит отметить (добавить к нашему списку) идентичность механизма IOCTL (Input/Output Control Code, код управления вводом/выводом) – запросов. Теперь рассмотрим классификацию типов драйверов (замечу, довольно условную) для ОС Windows NT:

- ❑ драйверы пользовательского режима (User-Mode Drivers):
 - драйверы виртуальных устройств (Virtual Device Drivers, VDD) – используются для поддержки программ MS-DOS;
 - драйверы принтеров (Printer Drivers);
- ❑ драйверы режима ядра (Kernel-Mode Drivers):
 - драйверы файловой системы (File System Drivers) – осуществляют ввод/вывод на локальные и сетевые диски;
 - унаследованные драйверы (Legacy Drivers) – написаны для предыдущих версий Windows NT;
 - драйверы видеоадаптеров (Video Drivers) – реализуют графические операции;
 - драйверы потоковых устройств (Streaming Drivers) – осуществляют ввод/вывод потокового видео и звука;
 - WDM-драйверы (Windows Driver Model, WDM) – поддерживают технологию Plug&Play и управления электропитанием.

Далее стоит отметить, что драйверы бывают одно- и многоуровневыми. Если драйвер является многоуровневым, то обработка запросов ввода/вывода распределяется между несколькими драйверами, каждый из которых выполняет свою часть работы. Между этими драйверами можно “поставить” любое количество фильтр-драйверов (filter-drivers). Также сейчас необходимо запомнить два термина – вышестоящие (higher-level) и нижестоящие (lower-level) драйверы. При обработке запроса данные идут от вышестоящих драйверов к нижестоящим, а при возврате результатов – наоборот. Ну и, понятно, одноуровневый (monolithic) драйвер просто является противоположностью многоуровневому.

Для технологии Plug&Play существуют три уровня-типа драйверов:

- шинные драйверы;
- фильтр-драйверы;
- функциональные драйверы.

Упомянем о таком базисном понятии, как уровни запросов прерываний (IRQL). Как известно, прерывания обрабатываются в соответствии с их приоритетом. В Windows NT используется особая схема прерываний, называемая *уровнями запросов прерываний*. Всего уровней IRQL 32, самый низкий – 0 (passive), самый высокий – 31 (high). Прерывания с уровня 0 по 2 (DPC\dispatch) являются программными, а с 3 по 31 – аппаратными. Существуют специальные функции ядра, позволяющие узнать текущий уровень IRQL, а также сменить (понизить или повысить) его. Это довольно непростое,

однако, дело, в котором есть множество своих нюансов (с каких уровней какие операции можно производить и т. д.).

После того как мы более или менее разобрались с общими понятиями, мы уже можем приступить к обсуждению каких-то более сложных технологий. В частности, о технологии Plug and Play, которую я упоминал несколькими абзацами выше.

Технология Plug&Play (в условном переводе – “подключи и работай”) – это технология, состоящая как из программной, так и из аппаратной поддержки механизма, позволяющего подключать/отключать, настраивать и т. д. применительно к системе все устройства, подключаемые к ней (конечно же, при условии, что подключаемые устройства поддерживают Plug&Play-технологию). В идеале весь этот процесс осуществляет только механизм Plug&Play, и какие-то действия со стороны пользователя вообще не требуются. Для каких-то устройств это так и происходит, для других – проблем, к сожалению, может быть гораздо больше. Кроме того, для успешной работы Plug&Play необходима не только поддержка этой технологии со стороны устройств, но также, конечно, со стороны драйверов и системного ПО.

Какие возможности предоставляет системное ПО (вместе с драйверами), поддерживающее технологию Plug and Play?

- автоматическое распознавание подключенных к системе устройств;
- распределение и перераспределение ресурсов (таких как, например, порты ввода/вывода и участки памяти) между запросившими их устройствами;
- загрузка необходимых драйверов;
- предоставление драйверам необходимого интерфейса для взаимодействия с технологией Plug&Play;
- реализация механизма, позволяющего драйверам и приложениям получать информацию касательно изменений в наборе устройств, подключенных к системе устройств, и совершить необходимые действия.

Главное перечислили. А теперь перейдем к рассмотрению структуры механизма Plug&Play.

Система Plug&Play состоит из двух компонентов, находящихся соответственно в пользовательском режиме и режиме ядра – менеджера Plug&Play пользовательского режима и менеджера Plug&Play “ядерного” режима.

Менеджер Plug&Play режима ядра работает с ОС и драйверами для конфигурирования, управления и обслуживания устройств. Менеджер

Plug&Play пользовательского режима же взаимодействует с установочными компонентами пользовательского режима для конфигурирования и установки устройств. Также, при необходимости, менеджер Plug&Play взаимодействует с приложениями.

PnP (сокращенное обозначение Plug&Play) может успешно работать со следующими типами устройств:

- физические устройства;
- виртуальные устройства;
- логические устройства.

А сейчас сделаем небольшой обзор инструментов, которые мы будем использовать при написании драйвера.

1.2 ИНСТРУМЕНТАРИЙ

Среди языков программирования, популярных среди разработчиков драйверов под ОС Windows, выделяются языки C, C++ и Assembly. В качестве языка, используемого для написания приложения, выбран язык C, так как синтаксис и некоторые средства C изучались автором на протяжении четырех семестров обучения в университете, а также существует большое количество источников для изучения процесса написания драйверов под платформу Windows.

Теперь рассмотрим инструменты, которые использовались при создании и проверки работы драйвера.

- **Visual Studio Community 2019 v.16.5.4**

Интегрированная среда разработки Visual Studio – это стартовая площадка для написания, отладки и сборки кода, а также последующей публикации приложений. Интегрированная среда разработки (IDE) представляет собой многофункциональную программу, которую можно использовать для различных аспектов разработки программного обеспечения. Помимо стандартного редактора и отладчика, которые существуют в большинстве сред IDE, Visual Studio включает в себя компиляторы, средства автозавершения кода, графические конструкторы и многие другие функции для упрощения процесса разработки.

- **WDK v.10.0.18362.1**

WDK (от англ. *Windows Driver Kit*) – набор из средств разработки, заголовочных файлов, библиотек, утилит, программного кода примеров и документации, который позволяет программистам создавать драйверы для устройств по определённой технологии или для определённой платформы (программной или программно-аппаратной). Название произошло от более общего термина SDK (англ. *Software Development Kit*), которым обозначают комплекты для разработки программ вообще, не только драйверов.

Создание драйвера возможно и без использования WDK, однако WDK содержит средства, упрощающие разработку драйвера (например, готовые примеры и шаблоны кода), обеспечивающие совместимость драйвера с операционной системой (символические определения констант, определения интерфейсных функций ОС, определения, зависящие от типа и версии ОС), а также установку и тестирование драйвера.

Продукт доступен для бесплатной загрузки через сайт Microsoft Connect и содержит в себе средства построения программ как режима ядра, так и пользовательского режима.

- **DebugView v.4.90**

DebugView – это приложение, которое позволяет отслеживать отладочную информацию в вашей локальной системе, или любом компьютере в сети, к которому можете подключиться по протоколу TCP / IP. Он способен отображать отладочную информацию как режима ядра, так и Win32, поэтому вам не нужен специализированный отладчик, чтобы поймать необходимую информацию о вашем приложении и драйвере, а также нет необходимости в том, чтобы изменять нестандартные выходные интерфейсы ваших приложений и драйверов.

В первую очередь устанавливаем Visual Studio 2019 с официального сайта по адресу <https://visualstudio.microsoft.com>. При запуске **Visual Studio Installer** перейдите в раздел **Workloads**, затем в пункте **Installation Details** выберете пункт **Universal Windows Platform development** и начните установку.

После того, как Visual Studio Installer завершил установку, необходимо скачать пакет **WDK** для ОС Windows версии 1903 официально сайта по адресу <https://go.microsoft.com/fwlink/?linkid=2085767>. В конце процесса установки WDK перед вами появится диалоговое окно, которое предложит вам установить расширение WDK в установленную вами ранее Visual Studio 2019.

После выполнения вышеописанных действий, вы будете готовы приступить к написанию драйверов по Windows.

Однако для проверки работоспособности драйвера нам необходимо скачать утилиту **DebugView** с официального сайта по адресу <https://download.sysinternals.com/files/DebugView.zip>.

Теперь у вас есть весь необходимый набор для начала написания драйвера-фильтра файловой системы.

2 СТРУКТУРНОЕ ПРОЕКТИРОВАНИЕ

Для начала разберемся с общей иерархией драйверов и их предназначениями в ОС Windows. В Unix-подобных системах и в Windows драйвера подразделяются на две большие группы: драйверы режима ядра – это те драйвера, которые непосредственно принимают участие в корректной работе операционной системы; драйверы пользовательского режима – это драйвера, обеспечивающие корректную работу операционной системы со сторонними устройствами, подключаемыми к компьютеру, например, драйвера принтеров, виртуальных устройств и т. д.

Опираясь на вышеописанное, вполне очевидно, что данный драйвер относится к первой группе – драйверы режима ядра.

Драйвер-фильтр файловой системы – это прослойка между менеджером ввода/вывода и самим драйвером файловой системы. Он позволяет изменять запрос менеджера и ответ драйвера файловой системы, тем самым дает возможность переопределять поведение файловой системы компьютера. На рисунке 1 представлена упрощенная схема взаимодействия менеджера файловой системы, драйвера-фильтра файловой системы и драйвера файловой системы.

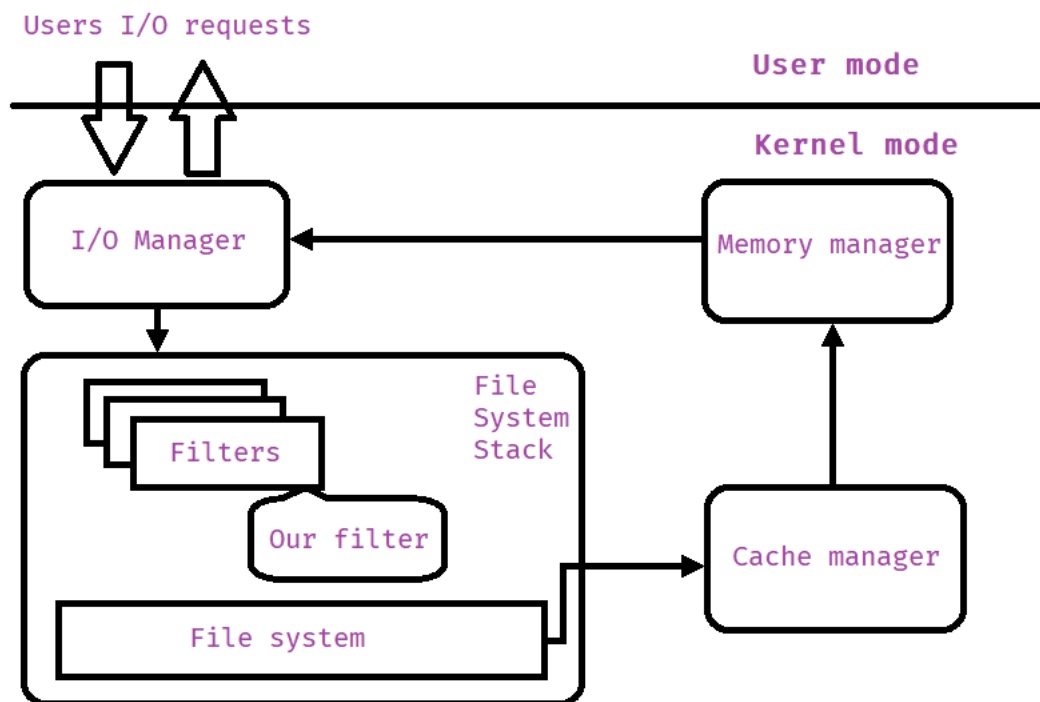


Рис. 1. Схема работы драйвера-фильтра файловой системы.

Теперь перейдем к созданию структуры фильтр-драйвера. Как и любой драйвер, драйвер-фильтр должен предоставлять функции загрузки и выгрузки драйвера, затем в обязательном порядке все драйвера файловой системы должны зарегистрировать координирующую таблицу ввода/вывода и сам механизм регистрации обращений к файловой системе, который и будет выводить в отладочный модуль ядра активность файловой системы. Также важно организовать передачу запроса другим драйверам, состоящих в цепи обработки запроса. Это все основные структурные единицы драйвера-фильтра, которые необходимы для корректной работы драйвера.

Детальное рассмотрение внутренней реализации драйвера будет приведено в следующем разделе.

3 ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ

В данном разделе будет рассмотрен функционал разрабатываемого ПО.

3.1. Main.c

File system filter driver entry. Это точка доступа для каждого драйвера, включая фильтр-драйвера файловой системы. Самое первое, что мы должны сделать, – это объявить **DriverObject** как глобальную переменную (мы будем ее использовать позже):

```
////////////////////////////////////  
// Global data  
  
PDRIVER_OBJECT g_fsFilterDriverObject = NULL;  
////////////////////////////////////  
// DriverEntry - Entry point of the driver  
  
NTSTATUS DriverEntry(  
    __inout PDRIVER_OBJECT DriverObject,  
    __in PUNICODE_STRING RegistryPath  
)  
{  
    UNREFERENCED_PARAMETER(RegistryPath);  
    NTSTATUS status = STATUS_SUCCESS;  
    ULONG i = 0;  
    //ASSERT(FALSE); // This will break to debugger  
    //  
    // Store our driver object.  
    //  
    g_fsFilterDriverObject = DriverObject;  
    ...  
}
```

Setting the IRP dispatch table. Следующий шаг в разработке фильтр-драйвера файловой системы является заполнение dispatch table указателями на функции-обработчики IRP. Нам нужно реализовать в драйвере сквозной обработчик IRP, который будет посылать запросы дальше по цепочке. Также необходимо реализовать обработчик для **IRP_MJ_CREATE**, чтобы получать имена открытых файлов. Реализацию IRP-обработчиков мы рассмотрим позже.

```
////////////////////////////////////  
// DriverEntry - Entry point of the driver  
  
NTSTATUS DriverEntry(  
    __inout PDRIVER_OBJECT DriverObject,  
    __in PUNICODE_STRING RegistryPath  
)  
{  
    ...  
    //  
    // Initialize the driver object dispatch table.  
    //
```

```

for (i = 0; i <= IRP_MJ_MAXIMUM_FUNCTION; ++i)
{
    DriverObject->MajorFunction[i] = FsFilterDispatchPassThrough;
}

DriverObject->MajorFunction[IRP_MJ_CREATE] = FsFilterDispatchCreate;
...
}

```

Setting fast I/O dispatch table. Драйвер-фильтр файловой системы обязательно должен иметь fast I/O dispatch table (fast I/O dispatch table – это таблица, в которой хранятся точки входа в процедуры быстрой отправки ввода-вывода). Если таблица не будет инициализирована, то это повлечет за собой сбой в работе ОС. Fast I/O – это другой способ запуска операций ввода-вывода, который быстрее, чем IRP. Операции быстрого ввода-вывода всегда синхронны. Если обработчик быстрого ввода-вывода возвращает **FALSE**, то мы не можем использовать операции быстрого ввода-вывода. В этом случае IRP будет создан.

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Global data
FAST_IO_DISPATCH g_fastIoDispatch =
{
    sizeof(FAST_IO_DISPATCH),
    FsFilterFastIoCheckIfPossible,
    ...
};
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// DriverEntry - Entry point of the driver
NTSTATUS DriverEntry(
    __inout PDRIVER_OBJECT DriverObject,
    __in     PUNICODE_STRING RegistryPath
)
{
    ...
    //
    // Set fast-io dispatch table.
    //
    DriverObject->FastIoDispatch = &g_fastIoDispatch;
    ...
}

```

Registering notifications about file system changes. Разрабатывая драйвер-фильтр файловой системы, мы должны регистрировать уведомления об изменениях внутри файловой системы. Крайне важно отслеживать: активируется или отключается файловая система, для выполнения подключения/отключения драйвера-фильтра файловой системы. Ниже вы можете увидеть, как отслеживать эти изменения.

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// DriverEntry - Entry point of the driver
NTSTATUS DriverEntry(
    __inout PDRIVER_OBJECT DriverObject,
    __in     PUNICODE_STRING RegistryPath
)

```

```

    )
{
    ...
    //
    // Registered callback routine for file system changes.
    //
    status = IoRegisterFsRegistrationChange(DriverObject,
FsFilterNotificationCallback);
    if (!NT_SUCCESS(status))
    {
        return status;
    }
    ...
}

```

Setting driver unload routine. Финальной частью инициализации фильтра драйвера файловой системы является функция выгрузки драйвера. Данная функция поможет нам загружать и выгружать данный драйвер из файловой системы не прибегая к перезагрузке системы. Тем не менее, этот драйвер действительно выгружается только для отладки, так как безопасно выгружать фильтры файловой системы невозможно. Не рекомендуется производить выгрузку фильтра в промышленном коде.

```

////////////////////////////////////
// DriverEntry - Entry point of the driver
NTSTATUS DriverEntry(
    __inout PDRIVER_OBJECT DriverObject,
    __in PUNICODE_STRING RegistryPath
)
{
    ...
    //
    // Set driver unload routine (debug purpose only).
    //
    DriverObject->DriverUnload = FsFilterUnload;
    return STATUS_SUCCESS;
}

```

File system driver unload routine. Функция выгрузки драйвера удаляет их и освобождает память, выделенную под их хранение. Следующим шагом в разработке нашего драйвера является отключение регистрации уведомлений об изменениях файловой системы.

```

////////////////////////////////////
// Unload routine
VOID FsFilterUnload(
    __in PDRIVER_OBJECT DriverObject
)
{
    ...
    //
    // Unregistered callback routine for file system changes.
    //
}

```

```

IoUnregisterFsRegistrationChange(DriverObject,
FsFilterNotificationCallback);
...
}

```

После отключения регистрации уведомлений, мы должны пройтись в цикле по всем созданным объектам, отсоединить и удалить их. Затем подождите 5 секунд, пока не завершатся оставшиеся IRP. Обратите внимание, что это решение только для отладки. Этот прием работает в большинстве случаев.

```

////////////////////////////////////
// Unload routine
VOID FsFilterUnload(
    __in PDRIVER_OBJECT DriverObject
)
{
    ...
    for (;;)
    {
        IoEnumerateDeviceObjectList(
            DriverObject,
            devList,
            sizeof(devList),
            &numDevices);

        if (0 == numDevices)
        {
            break;
        }

        numDevices = min(numDevices, RTL_NUMBER_OF(devList));

        for (i = 0; i < numDevices; ++i)
        {
            FsFilterDetachFromDevice(devList[i]);
            ObDereferenceObject(devList[i]);
        }

        KeDelayExecutionThread(KernelMode, FALSE, &interval);
    }
}

```

3.2. IrpDispatch.c

Dispatch pass-through. Единственной задачей данного IRP-обработчика является передача запросов к следующему драйверу. Следующий объект драйвера хранится в нашем объектном расширении.

```

////////////////////////////////////
// PassThrough IRP Handler
NTSTATUS FsFilterDispatchPassThrough(
    __in PDEVICE_OBJECT DeviceObject,
    __in PIRP Irp
)
{

```

```

PFSFILTER_DEVICE_EXTENSION pDevExt =
(PFSFILTER_DEVICE_EXTENSION)DeviceObject->DeviceExtension;

IoSkipCurrentIrpStackLocation(Irp);
return IoCallDriver(pDevExt->AttachedToDeviceObject, Irp);
}

```

Dispatch create. Каждый раз при создании файла файловая система вызывает данный IRP-обработчик. После получения имени файла из **PFILE_OBJECT** мы выводим его в канал отладки ядра. После этого мы вызываем обработчик, который мы описали выше. Обратите внимание, что актуальное имя файла существует в **PFILE_OBJECT** только до завершения операции создания файла. Также существует относительное открытие файла, а также открытие по идентификатору файла. В сторонних ресурсах вы можете найти более подробную информацию о поиске имен файлов в этих случаях.

```

////////////////////////////////////
////////////////////////////////////
// IRP_MJ_CREATE IRP Handler
NTSTATUS FsFilterDispatchCreate(
    __in PDEVICE_OBJECT DeviceObject,
    __in PIRP Irp
)
{
    PFILE_OBJECT pFileObject = IoGetCurrentIrpStackLocation(Irp)->
FileObject;

    DbgPrintEx(DPFLTR_IHVDRIVER_ID, DPFLTR_ERROR_LEVEL, "%wZ\n",
&pFileObject->FileName);
    DbgPrint("%wZ\n", &pFileObject->FileName);

    return FsFilterDispatchPassThrough(DeviceObject, Irp);
}

```

3.3. FastIo.c

Так как не все функции/операции быстрого ввода-вывода должны быть реализованы файловой системой, то мы должны проверить актуальность нашей диспатч-таблицы быстрых операций ввода-вывода, используя следующий макрос:

```

////////////////////////////////////
// Macro to test if FAST_IO_DISPATCH handling routine is valid
#define VALID_FAST_IO_DISPATCH_HANDLER(_FastIoDispatchPtr, _FieldName) \
    (((_FastIoDispatchPtr) != NULL) && \
    (((_FastIoDispatchPtr)->SizeOfFastIoDispatch) >= \
    (FIELD_OFFSET(FAST_IO_DISPATCH, _FieldName) + sizeof(void *))) && \
    ((_FastIoDispatchPtr)->_FieldName != NULL))

```

Fast I/O pass-through. В отличие от IRP-запросов, для отправки запроса, используя операции быстрого ввода-вывода, требуется огромное количество кода потому, что каждая функция из набора имеет свой собственный набор параметров. Ниже приведен пример общей функции отправки запроса:

```

/////////////////////////////////////////////////////////////////
BOOLEAN FsFilterFastIoQueryBasicInfo(
    __in PFILE_OBJECT      FileObject,
    __in BOOLEAN           Wait,
    __out PFILE_BASIC_INFORMATION Buffer,
    __out PIO_STATUS_BLOCK IoStatus,
    __in PDEVICE_OBJECT     DeviceObject
)
{
    //
    // Pass through logic for this type of Fast I/O
    //

    PDEVICE_OBJECT nextDeviceObject =
    ((PFSFILTER_DEVICE_EXTENSION)DeviceObject-> DeviceExtension)->
    AttachedToDeviceObject;
    PFAST_IO_DISPATCH fastIoDispatch = nextDeviceObject->DriverObject->
    FastIoDispatch;

    if (VALID_FAST_IO_DISPATCH_HANDLER(fastIoDispatch,
    FastIoQueryBasicInfo))
    {
        return (fastIoDispatch->FastIoQueryBasicInfo)(
            FileObject,
            Wait,
            Buffer,
            IoStatus,
            nextDeviceObject);
    }

    return FALSE;
}

```

Fast I/O detach device. Устройство отсоединения – это специальный запрос быстрого ввода-вывода, который мы должны обработать без вызова следующего драйвера. Мы должны удалить наш драйвер-фильтр из стека файловой системы. Ниже приведен пример кода, который показывает, как можно легко обработать данный запрос:

```

/////////////////////////////////////////////////////////////////
VOID FsFilterFastIoDetachDevice(
    __in PDEVICE_OBJECT     SourceDevice,
    __in PDEVICE_OBJECT     TargetDevice
)
{
    //
    // Detach from the file system's volume device object.
    //

    IoDetachDevice(TargetDevice);
    IoDeleteDevice(SourceDevice);
}

```


3.4. Notification.c

Файловая система состоит из устройств управления (control devices) и устройств хранения (volume devices). Устройства хранения прикреплены к стеку запоминающего устройства. Устройства контроля регистрируется в качестве файловой системы.

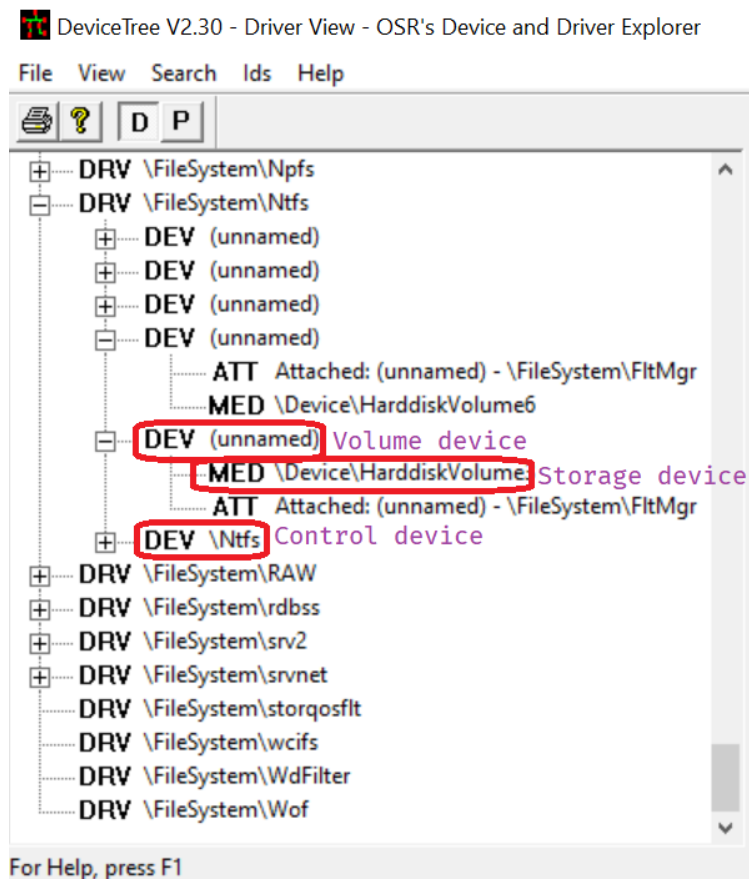


Рис. 2. Устройство файловой системы.

Каждый раз для всех файловых систем, находящихся на вашем компьютере, при их попытке загрузиться или выгрузиться она посылает коллбэк. Это отличное место, чтобы реализовать механизм установки и отсоединения нашего фильтра. Когда файловая система активируется, мы устанавливаем наш фильтр-драйвер (если он уже не установлен) и к каждому ее устройству хранения и подключаемся к ним тоже. Во время деактивации файловой системы, мы проверяем ее систему управления, находим наше устройство и отсоединяем его. Отсоединение от устройств хранения файловой системы осуществляется в рамках описанной нами ранее функции `Fsfilterfastiodetachdevice`.

```

////////////////////////////////////
// This routine is invoked whenever a file system has either registered or
// unregistered itself as an active file system.

VOID FsFilterNotificationCallback(
    __in PDEVICE_OBJECT DeviceObject,
    __in BOOLEAN FsActive
)
{
    //
    // Handle attaching/detaching from the given file system.
    //

    if (FsActive)
    {
        FsFilterAttachToFileSystemDevice(DeviceObject);
    }
    else
    {
        FsFilterDetachFromFileSystemDevice(DeviceObject);
    }
}

```

3.5. AttachDetach.c

Данный файл содержит функции-помощники для присоединения, отсоединения и проверки состояния драйвера-фильтра.

Attaching. Чтобы присоединить наш драйвер, нам нужно функцию **IoCreateDevice** для создания нового device-объекта с расширением device. Затем мы скопировать флаги (**DO_BUFFERED_IO**, **DO_DIRECT_IO**, **FILE_DEVICE_SECURE_OPEN**) у device-объекта, к которому мы пытаемся присоединиться. После этого, необходимо в цикле с задержкой на случай неудачи вызвать функцию **IoAttachDeviceToDeviceStackSafe**. Наш запрос на присоединение может не выполниться в том случае, если инициализация device-объекта еще не завершена. После присоединения, мы сохраняем device-объект, к которому мы присоединились, и очищаем флаг **DO_DEVICE_INITIALIZING**. Ниже вы можете увидеть device-расширение:

```

////////////////////////////////////
// Structures

typedef struct _FSFILTER_DEVICE_EXTENSION
{
    PDEVICE_OBJECT AttachedToDeviceObject;
} FSFILTER_DEVICE_EXTENSION, *PFSFILTER_DEVICE_EXTENSION;

```

Detaching. Процесс отсоединения куда проще. Сначала мы из device-расширения получаем device, к которому мы присоединились, и вызываем функции **IoDetachDevice** и **IoDeleteDevice**.

```

////////////////////////////////////
void FsFilterDetachFromDevice(

```

```

    __in PDEVICE_OBJECT DeviceObject
    )
    {
        PFSFILTER_DEVICE_EXTENSION pDevExt =
        (PFSFILTER_DEVICE_EXTENSION)DeviceObject->DeviceExtension;

        IoDetachDevice(pDevExt->AttachedToDeviceObject);
        IoDeleteDevice(DeviceObject);
    }

```

Checking if our device is attached. Чтобы проверить, присоединен ли наш драйвер или нет, нам нужно пройти по всему стеку устройств, используя функции **IoGetAttachedDeviceReference** и **IoGetLowerDeviceObject**. Мы можем определить нашу устройство, просто сравнив device driver object с **g_fsFilterDriverObject**.

```

////////////////////////////////////
// Misc

BOOLEAN FsFilterIsMyDeviceObject(
__in PDEVICE_OBJECT DeviceObject
)
{
    return DeviceObject->DriverObject == g_fsFilterDriverObject;
}

```

Вот вы уже и ознакомились со структурой драйвера-фильтра файловой системы. В следующей главе мы остановимся на двух произвольных алгоритмах и сделаем пошаговый разбор.

4 РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ

В данном разделе будут рассмотрены алгоритмы, используемые в разрабатываемом системном ПО.

4.1. Выгрузка драйвера из системы

Блок-схема алгоритма приведена в приложении А.

Шаг 1. Начало алгоритма.

Шаг 2. Объявить и проинициализировать переменные.

Шаг 3. Отключаем регистрацию фильтром-драйвером файловой системы изменений в файловой системе при помощи функции `IoUnregisterFsRegistrationChange`.

Шаг 4. Заходим в бесконечный цикл.

Шаг 5. Получаем массив device-объектов.

Шаг 6. Если переменная `numDevices` равна нулю, то переходим к шагу 13.

Шаг 7. Иначе инициализируем переменную `numDevices` минимальным значением device-объектов.

Шаг 8. Заходим в цикл от 0 до `numDevices`.

Шаг 9. Отсоединяем от каждого подключенного устройства драйвер-фильтр при помощи функции `FsFilterDetachFromDevice`.

Шаг 10. Удаляем временный объект, созданный драйвером, при помощи макроса `ObDereferenceObject`.

Шаг 11. Останавливаем поток, в котором выполняется данный алгоритм на 5 секунд для того, чтобы все отправленные IRP-пакеты успели завершиться, с помощью функции `KeDelayExecutionThread`.

Шаг 12. Возвращаемся к шагу 3ю

Шаг 13. Конец алгоритма.

4.2. Проверка подключения драйвера

Блок-схема алгоритма приведена в приложении Б.

Шаг 1. Начало алгоритма.

Шаг 2. Создаем и инициализируем переменные, и получаем указатель на объект устройства самого высокого уровня в стеке драйверов и увеличиваем счетчик ссылок на этот объект с помощью функции `IoGetAttachedDeviceReference`.

Шаг 3. Заходим в цикл, который будет продолжаться до тех пор, пока переменная `currentDevObj` не будет равна `NULL`.

Шаг 4. Проверяем, нашли ли мы наш драйвер в списке, при помощи функции `FsFilterIsMyDeviceObject`. Если мы его нашли то переходим на шаг 5, иначе – на шаг 7.

Шаг 5. Уменьшаем количество ссылок на объект с помощью макроса **ObDereferenceObject**, так как в шаге 2 мы это количество увеличили на единицу.

Шаг 6. Возвращаем значение **TRUE** и переходим на шаг 12.

Шаг 7. Инициализируем переменную **nextDevObj** объектом более низкого уровня при помощи функции **IoGetLowerDeviceObject**.

Шаг 8. Уменьшаем количество ссылок на объект с помощью макроса **ObDereferenceObject**, так как в шаге 2 мы это количество увеличили на единицу.

Шаг 9. Перезаписываем в переменную **currentDevObj** значение из переменной **currentDevObj**.

Шаг 10. Проверяем не равна ли переменная **currentDevObj** **NULL**-у. Если нет, то переходим на одиннадцатый шаг, иначе – на шаг 3.

Шаг 11. Возвращаем значение **FALSE**.

Шаг 12. Конец алгоритма.

5 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

Минимальные системные требования:

- ОС Windows 10, минимальная версия 1507.

Комплект поставляемого ПО:

- Fsfilter.cat;
- FsFilter.inf;
- FsFilter.sys.

Установка драйвера:

1. Распакуйте архив с названием FsFilter.7z;
2. Так как на драйвере отсутствует цифровая подпись, то вам нужно отключить проверку операционной системой цифровой подписи драйверов. Вот подробные шаги:
 - 2.1. Нажмите на клавишу “win”;
 - 2.2. Нажмите на кнопку “Параметры”;
 - 2.3. Перейдите в раздел “Обновления и безопасность”;
 - 2.4. Затем выберите подраздел “Восстановление”;
 - 2.5. Найдите параграф под названием “Особые варианты загрузки”;
 - 2.6. Ниже будет кнопка “Перезагрузить сейчас”, нажмите на нее;
 - 2.7. На появившемся экране выберите действие “Поиск и устранение неисправностей”;
 - 2.8. Выберите пункт “Дополнительные параметры”;
 - 2.9. Затем “Параметры загрузки”;
 - 2.10. На появившемся окне найдите пункт под названием “Отключить обязательную проверку подписи драйверов” и нажмите на клавишу с номером, который совпадает с номером пункта (скорее всего это будет клавиша “7”);
 - 2.11. После этого компьютер перезагрузится.
3. После перезагрузки компьютера, откройте командную строку с доступом администратора.
4. Введите команду `sc create FsFilter type= filesys binPath=c:\path\to\folder\which\contains\FsFilter\FsFilter.sys`, где `c:\path\to\folder\which\contains\FsFilter` – это путь к директории, где хранится файл FsFilter.sys. В случае, если все пройдет успешно, вы увидите сообщение: `[SC] CreateService SUCCESS`.
5. После создания сервиса (установки драйвера), необходимо его запустить при помощи команды `sc start FsFilter`. В случае, если

все пройдет успешно, вы увидите сообщение:

```
SERVICE_NAME: FsFilter
      TYPE                : 2  FILE_SYSTEM_DRIVER
      STATE                : 4  RUNNING
                        <STOPPABLE,NOT_PAUSABLE,IGNORES_SHUTDOWN>
      WIN32_EXIT_CODE      : 0  <0x0>
      SERVICE_EXIT_CODE   : 0  <0x0>
      CHECKPOINT          : 0x0
      WAIT_HINT           : 0x0
      PID                 : 0
      FLAGS                :
```

6. Драйвер установлен.

Удаление драйвера:

1. Остановите работу драйвера, выполнив команду **sc stop FsFilter**.

В случае, если все пройдет успешно, вы увидите сообщение:

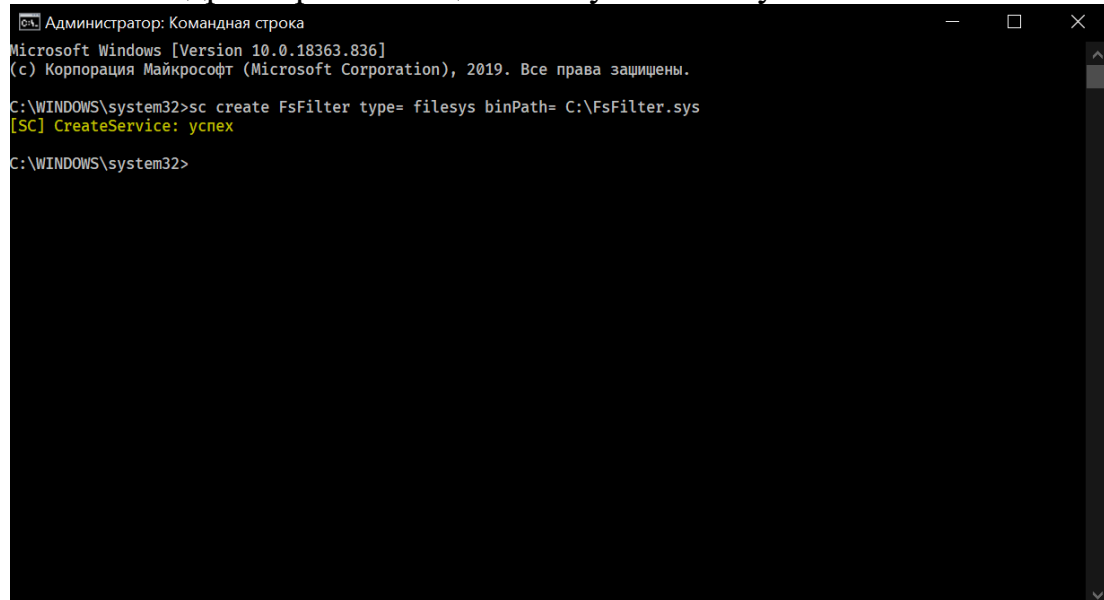
```
SERVICE_NAME: FsFilter
      TYPE                : 2  FILE_SYSTEM_DRIVER
      STATE                : 1  STOPPED
                        <STOPPABLE,NOT_PAUSABLE,IGNORES_SHUTDOWN>
      WIN32_EXIT_CODE      : 0  <0x0>
      SERVICE_EXIT_CODE   : 0  <0x0>
      CHECKPOINT          : 0x0
      WAIT_HINT           : 0x0
```

2. Для удаления сущности драйвера введите команду **sc delete FsFilter**. В случае, если все пройдет успешно, вы увидите сообщение: **[SC] DeleteService SUCCESS**.

Чтобы посмотреть сообщения, которые выводит драйвер-фильтр, вам необходимо скачать утилиту DebugView^[1].

6 ТЕСТИРОВАНИЕ

- Установка драйвера. Сообщение об успешной установке:



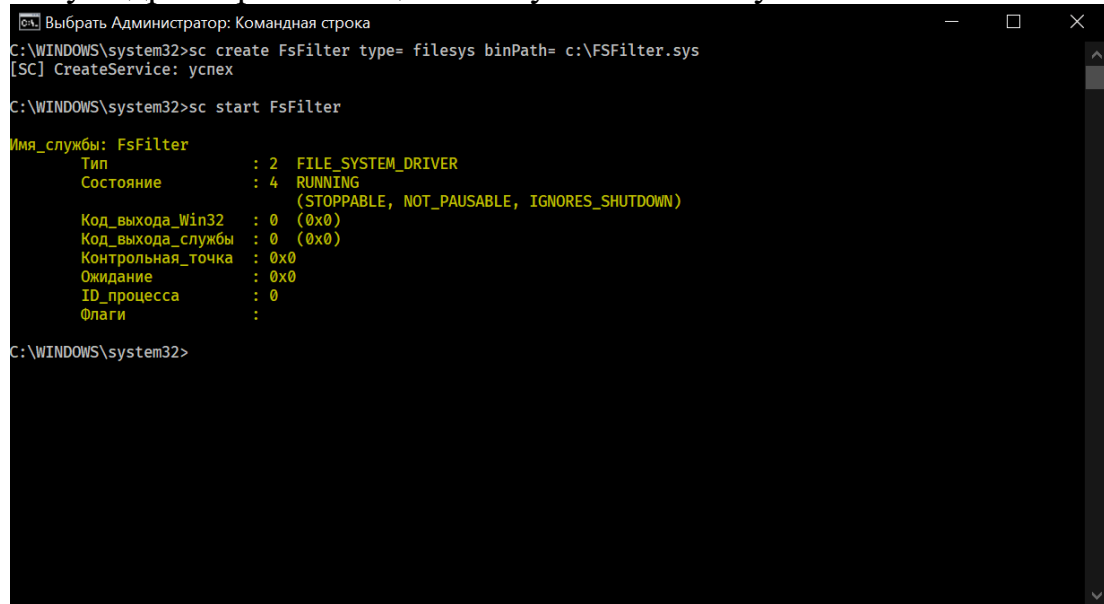
```
Администратор: Командная строка
Microsoft Windows [Version 10.0.18363.836]
(c) Корпорация Майкрософт (Microsoft Corporation), 2019. Все права защищены.

C:\WINDOWS\system32>sc create FsFilter type= filesys binPath= C:\FsFilter.sys
[SC] CreateService: ycneх

C:\WINDOWS\system32>
```

Рис. 3. Сообщение об успешной установке драйвера.

- Запуск драйвера. Сообщение об успешном запуске:



```
Выбрать Администратор: Командная строка
C:\WINDOWS\system32>sc create FsFilter type= filesys binPath= c:\FSFilter.sys
[SC] CreateService: ycneх

C:\WINDOWS\system32>sc start FsFilter

Имя_службы: FsFilter
Тип          : 2  FILE_SYSTEM_DRIVER
Состояние     : 4  RUNNING
               (STOPPABLE, NOT_PAUSABLE, IGNORES_SHUTDOWN)
Код_выхода_Win32 : 0  (0x0)
Код_выхода_службы : 0  (0x0)
Контрольная_точка : 0x0
Ожидание      : 0x0
ID_процесса    : 0
Флаги         :

C:\WINDOWS\system32>
```

Рис. 4. Сообщение об успешном запуске драйвера.

- Проверяем, что драйвер действительно установлен:

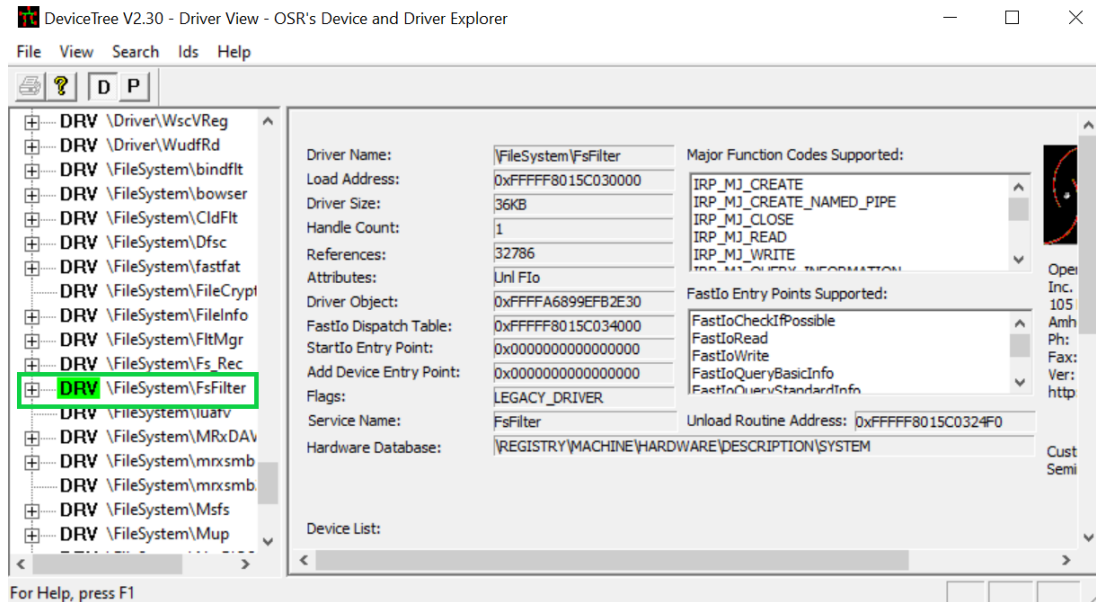


Рис. 5. Расположение драйвера в дереве устройств.

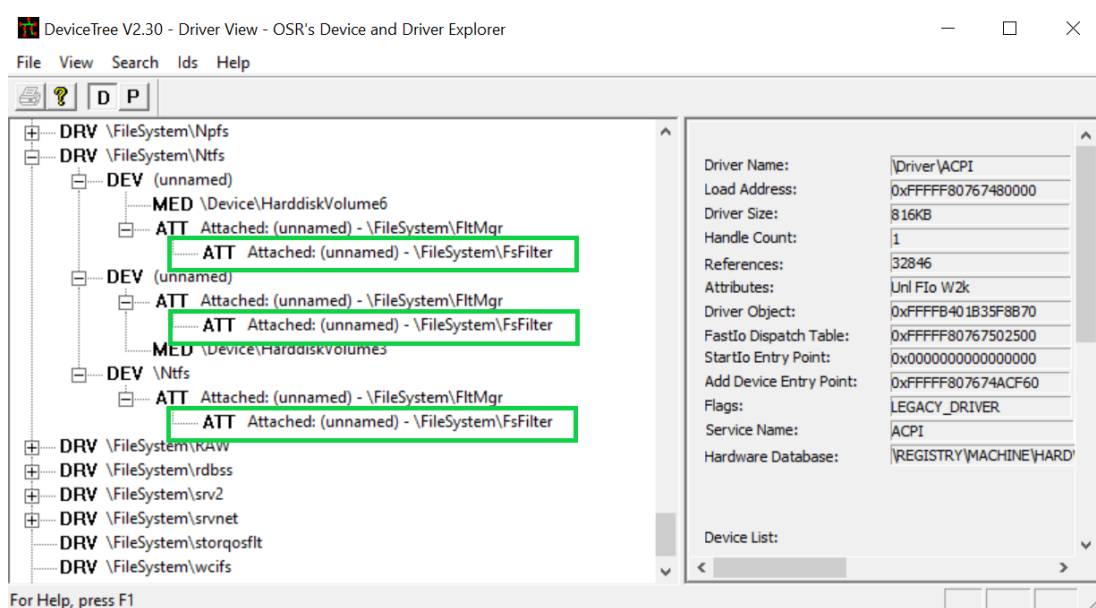


Рис. 6. Дополнительные объекты, созданные драйвером.

- Проверяем работоспособность (то, что драйвер действительно выводит сообщения в лог ядра Windows):

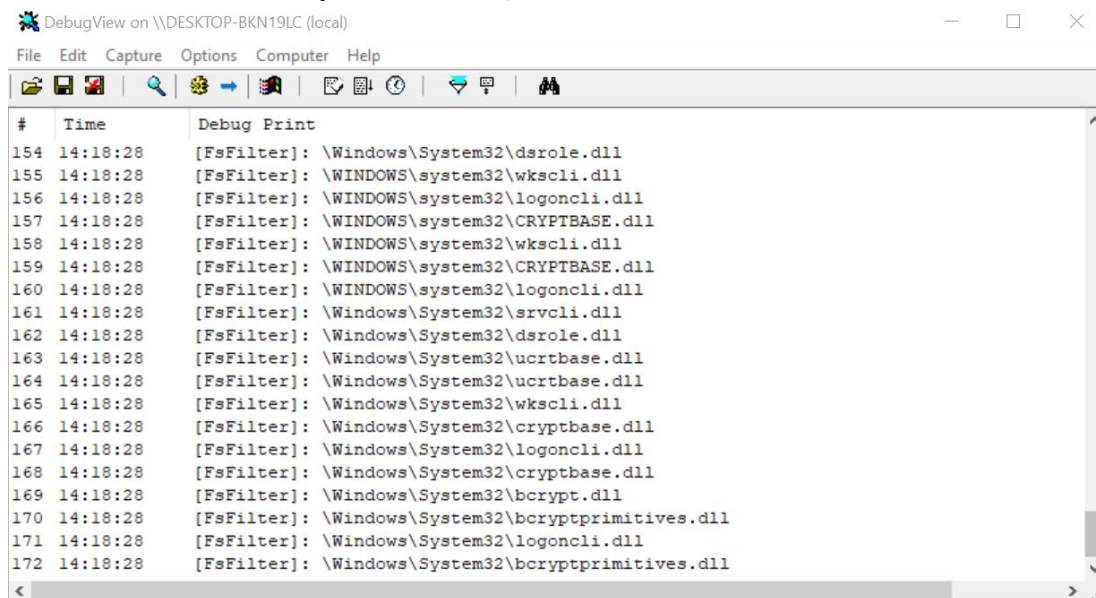


Рис. 7. Сообщения от драйвера об открытых файлах и директориях.

- Прекращаем работу драйвера. Сообщение об успешной установке:

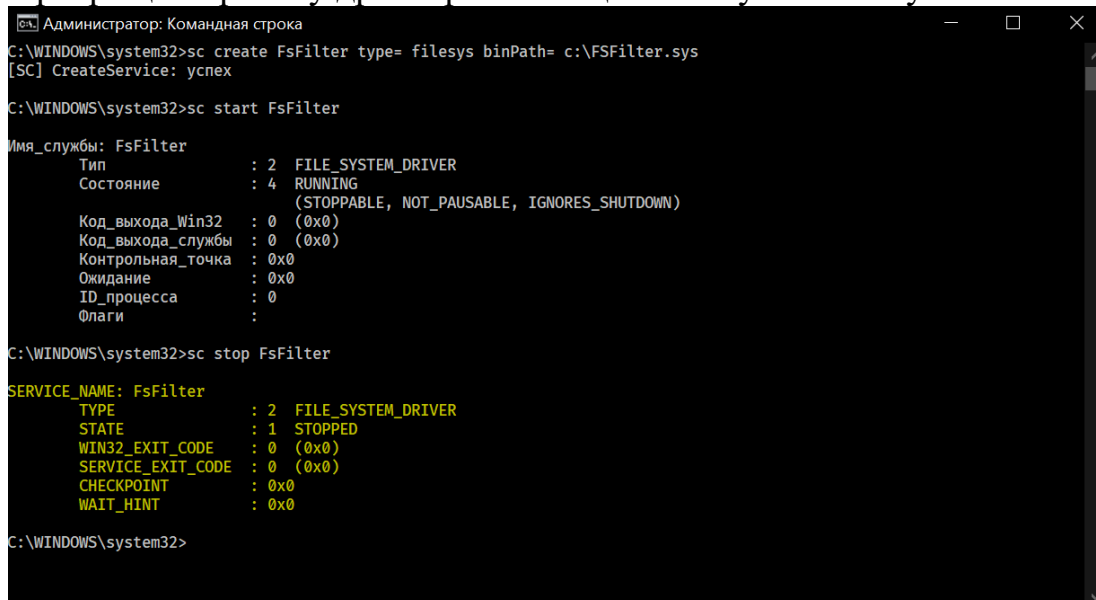


Рис. 8. Сообщение об успешной остановке работы драйвера.

- Удаление драйвера. Сообщение об успешном удалении:

```

Администратор: Командная строка
C:\WINDOWS\system32>sc create FsFilter type= filesystem binPath= c:\FsFilter.sys
[SC] CreateService: ycnex

C:\WINDOWS\system32>sc start FsFilter

Имя_службы: FsFilter
Тип          : 2  FILE_SYSTEM_DRIVER
Состояние    : 4  RUNNING
              (STOPPABLE, NOT_PAUSABLE, IGNORES_SHUTDOWN)
Код_выхода_Win32 : 0  (0x0)
Код_выхода_службы : 0  (0x0)
Контрольная_точка : 0x0
Ожидание      : 0x0
ID_процесса   : 0
Флаги        :

C:\WINDOWS\system32>sc stop FsFilter

SERVICE_NAME: FsFilter
TYPE               : 2  FILE_SYSTEM_DRIVER
STATE              : 1  STOPPED
WIN32_EXIT_CODE     : 0  (0x0)
SERVICE_EXIT_CODE : 0  (0x0)
CHECKPOINT         : 0x0
WAIT_HINT          : 0x0

C:\WINDOWS\system32>sc delete FsFilter
[SC] DeleteService: ycnex

C:\WINDOWS\system32>
  
```

Рис. 9. Сообщение об успешном удалении драйвера.

- Проверяем, что драйвер действительно удален:

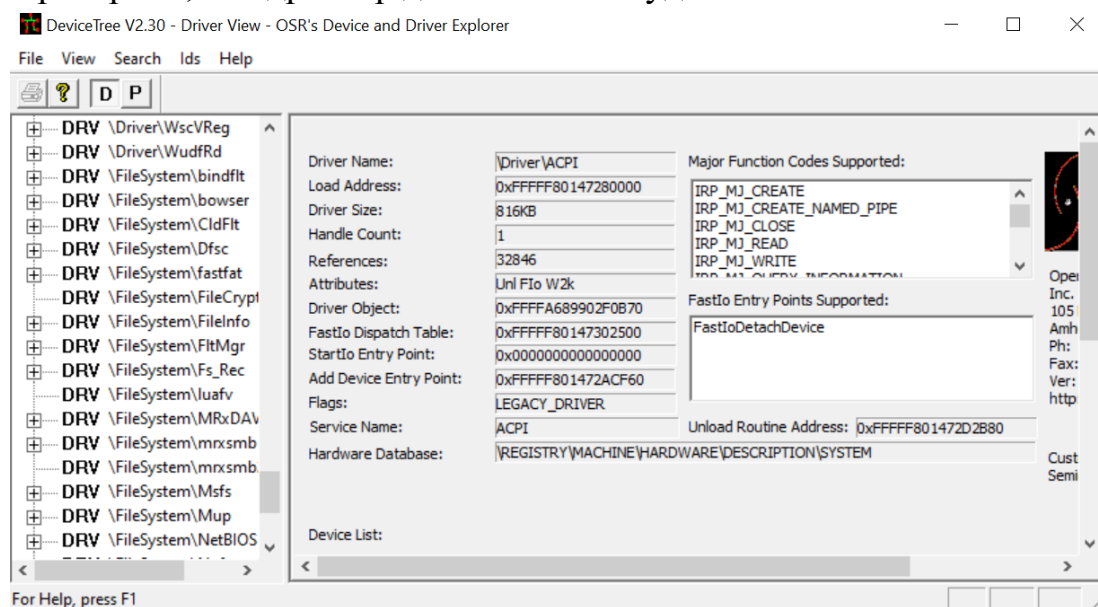


Рис. 10. Проверка отсутствия драйвера в дереве объектов.

Как было показано выше, драйвер-фильтр файловой системы работает согласно нашим ожиданиям, следовательно, все необходимые тесты он прошел.

7 ЗАКЛЮЧЕНИЕ

В ходе работы над курсовым проектом был сконструирован драйвер-фильтр файловой системы. Процесс разработки задокументирован в данной пояснительной записке.

Данный драйвер выводит сообщения о всех действиях файловой системы в лог ядра.

В дальнейшем планируется расширения функциональности драйвера (например, проверка безопасности и целостности файлов, с которыми взаимодействует файловая система(-ы)).

Хочется отметить, что процесс разработки принёс автору массу нового опыта, а также спровоцировал автора искать решения проблем в различных источниках, разбираться в исходном коде примеров WDK. Изучены основные принципы создания фильтр-драйверов.

8 СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Habr – Habr [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://habr.com/>.

[2] Орвик, П. Windows® Driver Foundation: разработка драйверов: Пер. с англ. / П. Орвик, Г. Смит. – М.: Издательство “Русская Редакция”; СПб.: “БХВ-Петербург”, 2008. – 880 с.

[3] Комиссарова В., Программирование драйверов для Windows. – СПб.: “БВХ-Петербург”, 2007. – 256 с.

[4] Windows hardware developer documentation – Учебник по разработке драйверов под ОС Windows для начинающих и продвинутых [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://docs.microsoft.com/ru-ru/windows-hardware/drivers/>.

[5] ORS Online [Электронный ресурс]. – Электронные данные. – Режим доступа: <http://www.osronline.com/>.

[6] YouTube-канал “Programming LoL” [Электронный ресурс]. – Электронные данные. – Режим доступа: https://www.youtube.com/channel/UCj-ldSP0XA_rHekmYUeFx_A.

ПРИЛОЖЕНИЕ А

(Обязательное)

Блок-схема алгоритма выгрузки драйвера

ПРИЛОЖЕНИЕ Б

(Обязательное)

Блок-схема алгоритма проверки подключения драйвера