Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей
Кафедра информатики
Дисциплина: Операционные среды и системное программирование»

К защите допустить:

И.О. Заведующего кафедрой информатики
_____С. И. Сиротко

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к курсовому проекту
на тему

АНАЛИЗ КОНФИГУРАЦИИ И ПОДКЛЮЧЕННЫХ УСТРОЙСТВ, ПРОИЗВОДИТЕЛЬНОСТИ ПОДСИСТЕМ КОМПЬЮТЕРА, ОГРАНИЧЕННОЕ УПРАВЛЕНИЕ УСТРОЙСТВАМИ НА ОС WINDOWS

БГУИР КП 1-40 04 01 012 ПЗ

Студент Д. С. Кончик

Руководитель Н. Ю. Гриценко

Нормоконтролер Н. Ю. Гриценко

Минск 2024

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В современном мире компьютеры и их компоненты стали неотъемлемой частью повседневной жизни. С каждым годом объем вычислительных задач, стоящих перед ними, неуклонно растет, и вместе с тем возрастает важность оптимизации работы компьютерных систем. Одним из ключевых аспектов оптимизации является анализ конфигурации и производительности подсистем компьютера, а также управление подключенными устройствами.

Цель данной курсовой работы состоит в исследовании и анализе конфигурации компьютерной системы под управлением операционной системы Windows, а также в разработке программных решений для обеспечения удобного и эффективного управления компьютером. В рамках этой задачи планируется создать не только диспетчер устройств, но и средство мониторинга производительности, обеспечивающее контроль и оптимизацию работы основных подсистем компьютера.

Для достижения поставленной цели необходимо решить следующие задачи:

1 Провести анализ конфигурации компьютерной системы, включая выявление основных компонентов и их характеристик.

2 Исследовать производительность основных подсистем компьютера, включая процессор, оперативную память, жесткий диск и графический адаптер.

3 Разработать программное обеспечение в виде диспетчера устройств для операционной системы Windows, а также средство мониторинга производительности, обеспечивающие удобное управление подключенными устройствами и контроль производительности системы.

Данная работа представляет собой попытку синтезировать знания о конфигурации и производительности компьютерных систем, а также навыки программирования для создания полезных инструментов, способствующих оптимизации работы компьютера под управлением операционной системы Windows.

Пояснительная записка оформлена в соответствии с СТП 01-2017 [1].

1 УПРАВЛЕНИЕ УСТРОЙСТВАМИ И АНАЛИЗ ПРОИЗВОДИТЕЛЬНОСТИ ПОДСИСТЕМ КОМПЬЮТЕРА

1.1 Устройства в ОС Windows

Устройство – это физическое или логическое оборудование или программное средство, которое выполняет определенную функцию в компьютерной системе. Физические устройства представляют собой конкретные аппаратные компоненты, такие как клавиатура, мышь, принтер, жесткий диск и другие компоненты, которые можно увидеть и осязать. Логические устройства, с другой стороны, создаются программно и могут эмулировать функциональность аппаратных устройств, например, виртуальные дисководы или виртуальные сетевые адаптеры.

Устройства могут выполнять различные функции, такие как ввод данных (например, через клавиатуру или сканер), вывод данных (например, на монитор или принтер), хранение данных (например, на жестком диске или флеш-накопителе) и обмен данными между компьютером и другими устройствами через сеть или интерфейсы передачи данных.

В компьютерной системе устройства работают взаимодействуя с операционной системой и другими программными компонентами через драйверы устройств. Драйверы предоставляют программный интерфейс для управления и взаимодействия с конкретными устройствами, позволяя программам использовать их функциональность без необходимости знать детали их работы. Устройства играют ключевую роль в обеспечении работы компьютерной системы и являются неотъемлемой частью её функциональности.

Устройства можно разделить на две основные категории: устройства ввода и устройства вывода (рисунок 1.1) [2].

Устройства ввода предназначены для ввода данных в компьютерную систему. Они позволяют пользователю передавать информацию компьютеру для обработки. Примеры устройств ввода включают в себя:

– клавиатура, используется для ввода текста и команд;

– мышь, позволяет пользователю управлять указателем на экране и взаимодействовать с графическим интерфейсом;

– сенсорный экран, позволяет пользователю взаимодействовать с устройством, касаясь экрана пальцем или стилусом;

– сканер, используется для преобразования бумажных документов или

изображений в цифровой формат;

– микрофон, позволяет записывать звуковые сигналы и голосовые команды.

Устройства вывода предназначены для вывода информации из компьютерной системы. Они отображают результаты обработки данных на удобном для восприятия пользователем устройстве. Примеры устройств вывода включают в себя:

– монитор, отображает графический интерфейс операционной системы, приложений, веб-страниц и другую информацию;

– принтер, используется для создания физических копий документов и изображений на бумаге;

– динамики или наушники, позволяют воспроизводить звуковые сигналы, музыку, аудиофайлы и голосовые сообщения;

– проектор, используется для вывода изображения на большой экран или поверхность, например, на стену или экран.

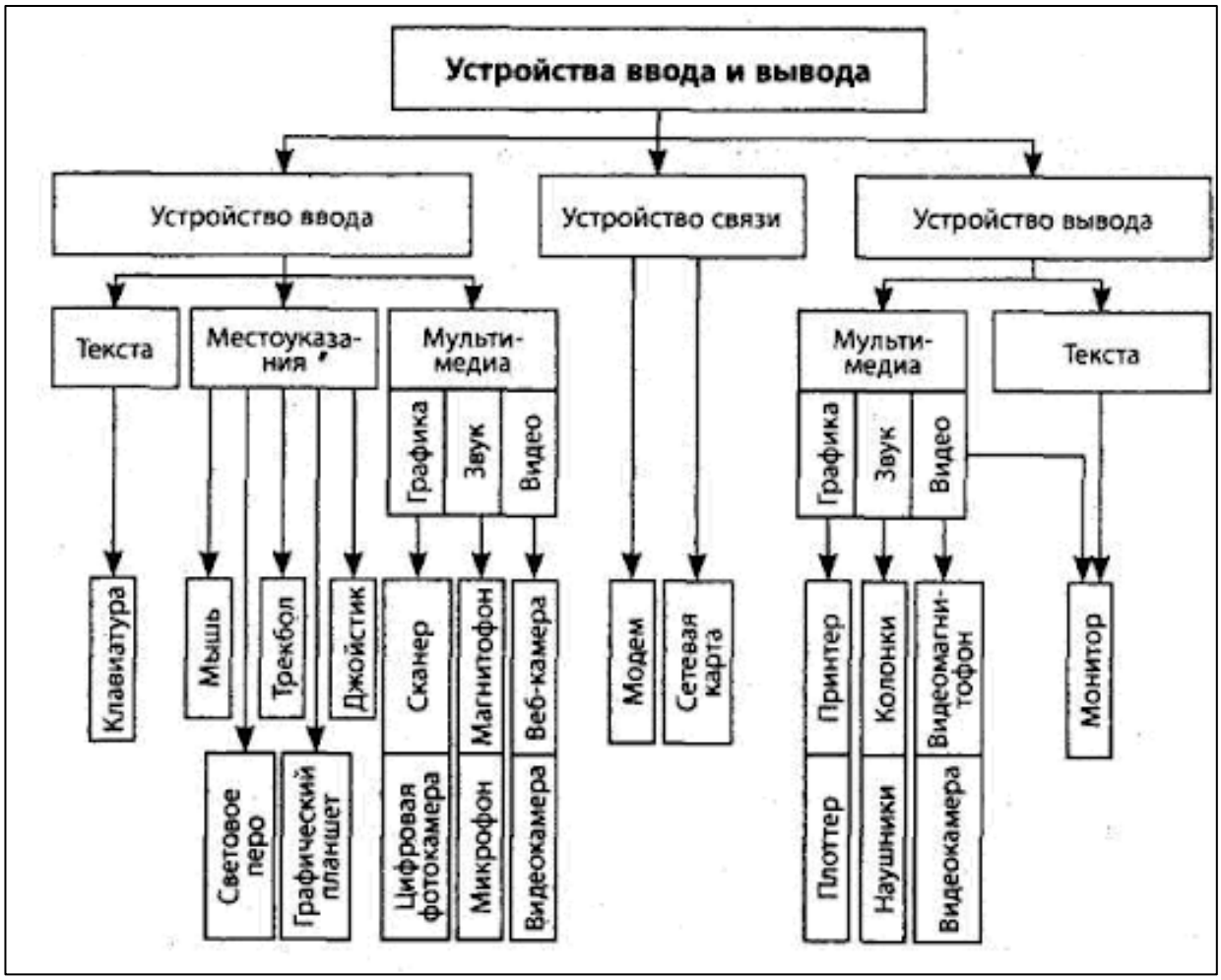

Рисунок 1.1 – Виды устройств

Устройства ввода и вывода являются необходимыми компонентами компьютерной системы, которые позволяют пользователям взаимодействовать с компьютером и получать результаты его работы.

В операционной системе Windows под устройствами обычно понимаются аппаратные компоненты компьютера и периферийные устройства, которые используются для ввода, вывода или хранения данных. Вот некоторые из наиболее распространенных типов устройств в Windows:

1 Процессор (CPU) – главный вычислительный компонент компьютера, который выполняет инструкции программ и управляет всеми операциями в системе. В Windows управление процессором и мониторинг его работы осуществляется с помощью различных инструментов, таких как Диспетчер задач и Планировщик задач.

Видеокарта (GPU) – отвечает за обработку графики и вывод изображения на экран монитора. Windows предоставляет драйверы и инструменты для настройки и управления параметрами видеокарты, а также мониторинга ее работы.

Хранилище (Storage) – используется для хранения данных на постоянной основе. В Windows устройства хранения обычно представлены в виде жестких дисков (HDD) или твердотельных накопителей (SSD). Для управления и мониторинга дисков используются инструменты, такие как Диспетчер дисков или Управление дисками.

Оперативная память (RAM) – используется для временного хранения данных и исполняемых программ во время работы компьютера. В Windows управление памятью происходит автоматически, но пользователи могут мониторить использование памяти с помощью Диспетчера задач.

Звуковая карта (Audio Adapter) – отвечает за воспроизведение и запись звука на компьютере. В Windows можно настраивать параметры звуковой карты, выбирать аудиоустройства воспроизведения и записи, а также контролировать громкость звука.

Сетевые адаптеры (Network Devices) – используются для подключения компьютера к сети, будь то локальная сеть LAN или интернет. В Windows пользователи могут настраивать параметры сетевых адаптеров, управлять сетевыми подключениями и мониторить состояние сети с помощью различных инструментов, таких как Центр управления сетями и общим доступом.

Это лишь небольшой обзор различных типов устройств, которые используются в операционной системе Windows. Каждое из них имеет свои уникальные характеристики и функциональные возможности, и управление

ими обычно осуществляется через специальные инструменты и драйверы, предоставляемые операционной системой.

Операционная система Windows взаимодействует с устройствами компьютера через драйверы устройств и системные службы. Вот основные шаги, по которым Windows работает с устройствами [3]:

1 Обнаружение устройств. При запуске компьютера или подключении нового устройства Windows проводит процесс обнаружения устройств. Операционная система сканирует шину данных (например, шину PCI или USB) для поиска подключенных устройств. Для этого Windows использует информацию, предоставленную BIOS (или UEFI) и конфигурационными файлами устройств. Если новое устройство обнаружено, Windows регистрирует его и начинает процесс загрузки драйвера.

2 Загрузка драйверов. Когда Windows обнаруживает устройство, она ищет соответствующий драйвер в своей базе данных драйверов. Если драйвер не найден на компьютере, Windows может попытаться загрузить его из Интернета через службу Windows Update. После загрузки драйвера Windows загружает его в оперативную память и инициирует процесс его установки.

3 Инициализация устройства. После загрузки драйвера операционная система инициализирует устройство. Это включает в себя установку базовых параметров и режимов работы, инициализацию оборудования и подготовку к взаимодействию с другими компонентами системы.

4 Управление ресурсами. Windows управляет ресурсами, необходимыми для работы каждого устройства. Это включает в себя выделение процессорного времени, оперативной памяти, прерываний и портов ввода-вывода. Операционная система оптимизирует использование ресурсов, чтобы обеспечить стабильную работу всех устройств и приложений.

5 Обработка запросов устройств. Когда приложение или пользователь взаимодействует с устройством, Windows передает запросы соответствующему драйверу устройства. Драйвер обрабатывает эти запросы, выполняя необходимые действия, например, считывая данные с диска или отправляя данные на принтер.

6 Мониторинг и управление. Windows постоянно мониторит состояние устройств и может выполнять различные действия для обеспечения их стабильной работы. Например, если обнаруживается сбой или конфликт, Windows может перезагрузить драйвер устройства или отключить его для предотвращения дальнейших проблем. Также Windows может перевести устройство в режим ожидания (например, выключить монитор), чтобы сэкономить энергию.

7 Обновление и обслуживание. Windows отвечает за обновление и обслуживание драйверов устройств. Это включает в себя автоматическое скачивание и установку обновлений драйверов через службу Windows Update или действия пользователя при необходимости. Обновление драйверов помогает обеспечить совместимость с новым оборудованием и исправить ошибки, повышая стабильность и производительность системы.

Plug and Play (PnP), дословно переводится как «Подключил и играй (работай)» – технология, предназначенная для быстрого определения и конфигурирования устройств в компьютере и других технических устройствах. PnP встроен в Windows и позволяет компьютерной системе автоматически адаптироваться к изменениям в подключаемом оборудовании с минимальным участием пользователя. Это означает, что пользователь может добавлять или удалять устройства без необходимости вручную настраивать систему или иметь глубокие знания оборудования компьютера. Например, можно подключить портативный компьютер к клавиатуре, мыши и монитору, и компьютер автоматически определит их без необходимости ручной настройки.

Для работы PnP необходима поддержка со стороны оборудования, системного программного обеспечения и драйверов. Индустрия разрабатывает стандарты для облегчения идентификации компонентов системы и плат надстроек, что обеспечивает более легкий процесс обмена и обновления компонентов компьютера.

Поддержка системного программного обеспечения для PnP вместе с драйверами PnP обеспечивает следующие возможности:

1 Автоматическое и динамическое распознавание установленного оборудования.

Выделение аппаратных ресурсов (и перераспределение). Диспетчер PnP определяет аппаратные ресурсы, запрашиваемые каждым устройством (например, порты ввода-вывода, запросы прерываний, каналы прямого доступа к памяти и расположения памяти), и назначает аппаратные ресурсы соответствующим образом. При необходимости диспетчер PnP перенастраивает назначения ресурсов, например при добавлении в систему нового устройства, которому требуются уже используемые ресурсы. Драйверы для устройств PnP не назначают ресурсы. Вместо этого запрошенные ресурсы для устройства определяются при перечислении устройства. Диспетчер PnP получает требования для каждого устройства во время выделения ресурсов. Ресурсы не настраиваются динамически для устаревших устройств, поэтому диспетчер PnP сначала назначает ресурсы

устаревшим устройствам.

Загрузка соответствующих драйверов.

Программный интерфейс для взаимодействия драйверов с системой PnP. Интерфейс включает подпрограммы диспетчера ввода-вывода, Plug and Play незначительные IRP (Input/Output Request Packet), обязательные стандартные подпрограммы драйвера и сведения в реестре.

Механизмы, которые позволяют драйверам и приложениям узнавать об изменениях в аппаратной среде и принимать соответствующие меры. PnP позволяет драйверам и коду пользовательского режима регистрироваться для определенных событий оборудования и получать уведомления о ней.

Чтобы драйвер квалифицируется как PnP, он должен предоставить необходимые точки входа PnP, обработать необходимые PnP IRP и следовать рекомендациям PnP.

Архитектура системы Plug and Play (включай и работай) включает в себя три важнейших компонента:

1 Операционная система, поддерживающая технологию типа Plug and Play, которая берет на себя управление всеми внешними устройствами, загружает необходимые драйверы, реагирует на все изменения в аппаратуре компьютера.

2 Система BIOS типа Plug and Play, которая может взаимодействовать с контроллерами ориентированными на Plug and Play и чипсетом системной платы компьютера;

3 Аппаратные средства компьютера и адаптеры поддерживающие Plug and Play.

Платы адаптеров Plug and Play информируют системную BIOS и операционную систему о необходимых им ресурсах. В свою очередь, BIOS и операционная система, по возможности, предотвращают конфликты и передают платам адаптеров информацию о конкретных выделенных ресурсах. После этого плата адаптера сама настраивается под выделенные ей ресурсы.

Автоматическое конфигурирование системы осуществляется во время выполнения расширенной процедуры самопроверки при выполнении POST (Power-On-Self-Test) [4]. BIOS идентифицирует, определяет расположение в слотах, и, по возможности, настраивает платы адаптеров Plug and Play. Эти действия выполняются в несколько этапов.

Отключаются настраиваемые узлы на системной плате и на платах адаптеров. При использовании плат расширения, удовлетворяющих спецификации Plug and Play, после включения компьютера, платы ожидают код инициализации от BIOS. Устройства после включения электропитания не

отвечают на обращения к пространству памяти и ввода-вывода, они доступны в это время только для операций конфигурационного чтения и записи.

Отыскиваются все устройства типа Plug and Play. Управляющие программные средства могут теперь с помощью команды активизации опросить плату, а с помощью другой команды все остальные платы переключить в “изолированное” состояние. В изолированном состоянии программные средства Plug and Play устанавливают связь только с одной активизированной платой. Эта плата передает программам Plug and Play свои характеристики. На основе этих данных осуществляется идентификация плат. По завершении процесса идентификации устанавливается связь между аппаратными и программными компонентами компьютера. При этом запрашиваются и назначаются необходимые конфигурационные параметры. Каждое устройство шины использует область пространства конфигурации (заголовок определенного формата, назначение оставшихся байтов области пространства конфигурации зависит от конкретного устройства и в спецификации не описывается). Область конфигурации доступна системе в любое время. В операциях конфигурационного чтения и записи становится доступной информация о потребностях устройства в системных ресурсах и возможных диапазонах их перемещения.

Создается исходная карта распределения ресурсов: портов, прерывании, каналов ПДП и памяти.

Активируются устройства ввода-вывода. Сканируются ПЗУ в устройствах. Конфигурируются устройства начальной загрузки (IPL – Initial Program Load), т.е. те, что используются для загрузки ОС компьютера.

В область конфигурации устройств записывается информация о выделенных системных ресурсах и режимах работы – это переводит их в рабочее состояние и становится возможным доступ по командам обращения к памяти и портам ввода-вывода контроллеров.

После тестирования и конфигурирования (включающего настройку устройств PnP), POST инициализирует загрузку операционной системы. Запускается начальный загрузчик. Загружаются необходимые компоненты системы, драйверы устройств. Определяется новая настройка конфигурации системы. Управление передается операционной системе. В процессе загрузки операционная система принимает на себя дальнейшую диагностику аппаратных средств – каждое подключенное устройство проходит проверку, и с него запрашиваются данные о всех параметрах его настройки. Перед выбором конфигурации старые параметры настройки проверяются вместе с параметрами настройки нового устройства, после чего они сравниваются и

при необходимости согласовываются. Если операционная система установит, что два компонента аппаратуры имеют одно и то же ресурсное обеспечение (например, одинаковые базовые адреса портов), то эти компоненты должны выдать операционной системе альтернативные ресурсы системы, с которыми они тоже могут работать.

Основная задача Plug and Play – компонента ОС – сообщить о конфликтах, которые не были устранены BIOS. В зависимости от возможностей операционной системы, вы можете попытаться конфигурировать адаптеры программно (с помощью экранного меню) или выключить компьютер и установить перемычки и переключатели на платах вручную. При следующем включении системы или ее перезагрузке будет проведена повторная проверка и выведены сообщения об оставшихся (или новых) конфликтах. После нескольких "заходов" все конфликты, как правило, устраняются.

Для каждого вновь подключенного в систему устройства (или отключенного устройства) процесс автоконфигурации повторяется. В компьютере имеется область энергонезависимой памяти ESCD для поддержки динамического конфигурирования системы Plug and Play, которая может автоматически обновляться при каждой перезагрузке компьютера. Этот процесс динамического конфигурирования и является причиной «задумчивости» при перезагрузке даже мощных компьютеров, имеющих средства PnP, а также не всегда предсказуемого поведения программного обеспечения, вызванного изменением распределения ресурсов по инициативе той же системы PnP.

Многие операционные системы поддерживают функцию автодетектирования устройств, т. е. автоматически определяется тип платы, ее конфигурация, номер используемого прерывания, базовый адрес портов ввода-вывода, канал прямого доступа. В операционных системах Windows важным средством управления всеми компонентами аппаратных средств является программа диспетчер устройств (Device-Manager), с помощью которой можно получить информацию об устройстве, выделенных ему ресурсах и выполнить необходимые согласование ресурсов.

1.2 Диспетчер устройств

Диспетчер устройств играет ключевую роль в обеспечении эффективного управления и контроля за аппаратными компонентами компьютера в операционной системе Windows. Его функциональность включает в себя не только просмотр информации о подключенных устройствах, но и управление ими с целью оптимизации работы компьютерной системы.

Основные функции Диспетчера устройств включают [5]:

1 Просмотр информации о подключенных устройствах. Диспетчер устройств предоставляет не только основные характеристики устройств, но и дополнительные данные, такие как версия драйвера, доступные режимы работы и даже серийные номера. Пользователи могут видеть, какие ресурсы использует каждое устройство, что помогает выявить конфликты и оптимизировать распределение ресурсов. Диспетчер устройств может показывать историю событий, связанных с каждым устройством, включая ошибки, предупреждения и изменения состояния.

2 Управление драйверами устройств. Диспетчер устройств позволяет создавать резервные копии драйверов и восстанавливать их при необходимости, что обеспечивает быстрое восстановление после сбоев или обновлений. Пользователи могут временно или постоянно блокировать автоматические обновления драйверов через Диспетчер устройств, чтобы избежать конфликтов или несовместимостей.

3 Иерархическая структура устройств. Диспетчер устройств позволяет быстро найти связанные устройства, например, устройства, которые используют один и тот же драйвер или подключены к одному контроллеру. Устройства группируются по типу (например, звуковые устройства, сетевые адаптеры), что упрощает ориентацию в структуре и облегчает поиск конкретных компонентов.

4 Решение проблем с устройствами. Диспетчер устройств может автоматически пытаться восстановить работоспособность устройства после выявления проблемы, например, путем перезагрузки драйвера или переназначения ресурсов. Пользователи могут использовать инструменты Диспетчера устройств для интерактивного решения проблем, такие как переустановка драйверов, изменение настроек или сброс настроек устройства.

Диспетчер устройств может предоставлять информацию о текущем использовании ресурсов системы каждым устройством, таким как процессорное время, объем оперативной памяти и пропускная способность

шины данных.

Пользователи могут управлять режимами энергосбережения для каждого устройства через Диспетчер устройств, что позволяет улучшить эффективность работы ноутбуков и других мобильных устройств.

Таким образом, Диспетчер устройств является неотъемлемым инструментом для обеспечения стабильной и эффективной работы компьютерной системы под управлением операционной системы Windows. Он облегчает процесс управления аппаратными компонентами, улучшает производительность и обеспечивает безопасность функционирования компьютера.

Диспетчер устройств был впервые внедрен в операционную систему Windows с выпуском Windows 95 в 1995 году (рисунок 1.2). Это событие ознаменовало значительное улучшение в возможностях управления аппаратными компонентами компьютера для пользователей Windows.

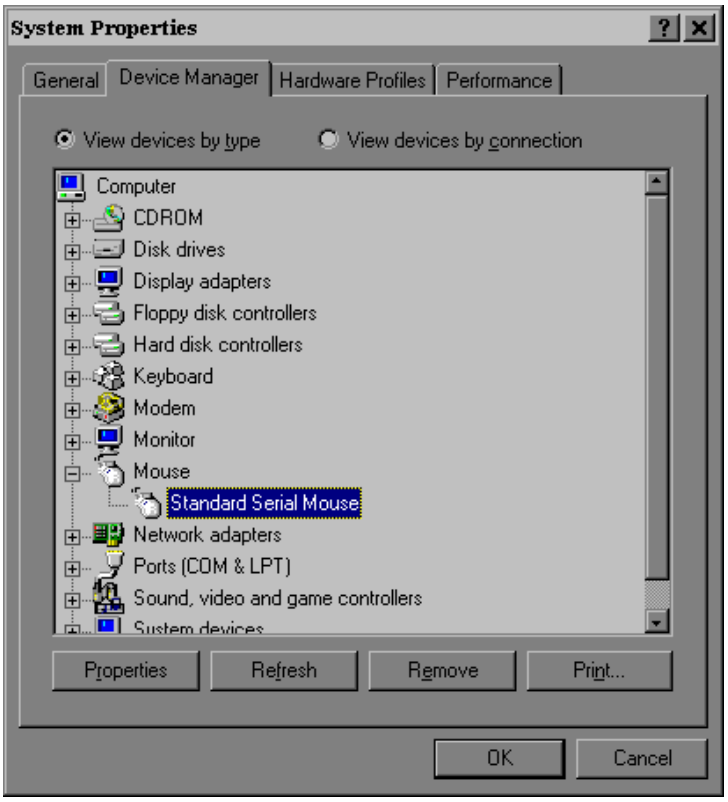

Рисунок 1.2 – Диспетчер устройств в Windows 95

В операционной системе Windows поддержка Plug and Play начала активно развиваться с выпуском Windows 95 в 1995 году. Это был значительный шаг вперед в облегчении процесса установки и настройки новых устройств для пользователей Windows.

С появлением Windows 95 была внедрена поддержка Plug and Play, которая позволила операционной системе автоматически обнаруживать и конфигурировать новые устройства при их подключении к компьютеру. Это означало, что пользователи больше не должны были выполнять ручную установку драйверов или проводить сложные настройки, чтобы начать использовать новое оборудование.

С каждым последующим выпуском Windows технология Plug and Play становилась более надежной и удобной. В более поздних версиях Windows, таких как Windows XP, Windows 7, Windows 10 и далее, Plug and Play стала неотъемлемой частью операционной системы, обеспечивая быстрое и простое подключение и использование новых устройств.

До появления Диспетчера устройств пользователи Windows сталкивались с рядом проблем и ограничений в управлении своим оборудованием. Вот несколько аспектов жизни без Диспетчера устройств:

1 Ограниченные средства диагностики. В отсутствие Диспетчера устройств пользователи сталкивались с ограниченными возможностями для выявления и диагностики проблем с устройствами. Они могли испытывать трудности в определении причин неисправности, так как не имели доступа к подробной информации о состоянии аппаратных компонентов. Чтобы выяснить причину проблемы, пользователю приходилось полагаться на собственные знания или обращаться к документации к устройству, что часто было недостаточно информативным. Это могло привести к длительным перерывам в работе и лишним затратам времени.

2 Ручная установка драйверов. Без Диспетчера устройств пользователи вынуждены были самостоятельно искать и устанавливать драйверы для подключаемого оборудования. Это могло быть особенно затруднительным для тех, кто не обладал достаточным опытом работы с компьютерами или не имел специализированных знаний в этой области. Ручная установка драйверов требовала от пользователя поиска нужных файлов драйверов в интернете или на дисках, а затем последовательного процесса установки, что могло быть непонятным и трудоемким.

3 Ограниченные возможности управления. Без Диспетчера устройств пользователи имели ограниченные возможности для изменения конфигурации своих компьютеров. Добавление нового оборудования или настройка существующих устройств могла быть сложной задачей, требующей вмешательства в конфигурационные файлы и реестр операционной системы. Это могло привести к риску ошибок и нестабильной работы системы при попытках вручную изменить конфигурацию устройств.

4 Частые конфликты и ошибки. В отсутствие централизованного механизма управления устройствами могли возникать частые конфликты и ошибки. Неправильная установка драйверов или несовместимость устройств могли привести к неполадкам и отказам в работе системы. Решение таких проблем требовало глубокого технического понимания работы аппаратного обеспечения и опыта в решении подобных проблем.

5 Зависимость от технических специалистов. При возникновении сложных проблем с оборудованием пользователи часто вынуждены были обращаться к техническим специалистам или сервисным центрам. Это могло быть затратным и времязатратным процессом, а также приводило к зависимости от внешней помощи при решении проблем, что могло замедлить процесс работы и повысить риски для безопасности данных.

В целом, жизнь без Диспетчера устройств в Windows означала более сложный и менее удобный процесс управления и поддержки аппаратного обеспечения компьютера. Появление Диспетчера устройств стало значительным шагом вперед, обеспечивая пользователям более простые и эффективные средства управления и контроля за своим оборудованием.

В Windows 98 (1998 год) был значительно улучшен Диспетчер устройств по сравнению с предыдущими версиями операционной системы (рисунок 1.3).

Рисунок 1.3 – Диспетчер устройств в Windows 98

До появления Windows 98 пользователи часто должны были ручным образом удалять или обновлять драйверы устройств через различные средства, например, путем поиска и удаления файлов драйверов в системных папках. С появлением Windows 98 они получили возможность обновлять и удалять драйверы прямо из интерфейса Диспетчера устройств, что сделало процесс управления драйверами более удобным и простым для пользователей. Также в Windows 98 были добавлены дополнительные сведения о свойствах устройств, доступные через Диспетчер устройств. Это включало в себя информацию о ресурсах устройства, его текущем состоянии, а также другие характеристики, которые помогали пользователям лучше понимать и настраивать своё оборудование.

В операционной системе Windows 2000, выпущенной в 2000 году, Диспетчер устройств был улучшен и стал более мощным инструментом для управления аппаратными компонентами компьютера [6]. Несмотря на то, что базовый интерфейс остался примерно таким же, как в предыдущих версиях Windows, были внесены значительные изменения в функциональность.

В Windows 2000 были расширены возможности диагностики устройств. Теперь Диспетчер устройств предоставлял более подробную информацию о состоянии и характеристиках каждого устройства. Это позволяло пользователям лучше понимать проблемы с оборудованием и быстрее находить способы их решения. Пользователи получили расширенные возможности настройки параметров устройств. Диспетчер устройств предоставлял доступ к различным настройкам и ресурсам устройств, позволяя пользователям изменять их в соответствии с их потребностями. Это включало в себя настройку прерываний, адресов ввода-вывода, параметров питания и многих других параметров. Windows 2000 предоставляла более широкие возможности управления устройствами через Диспетчер устройств. Пользователи могли проще и эффективнее управлять драйверами, обновлять их, отключать или переустанавливать устройства.

Эти улучшения в Диспетчере устройств Windows 2000 сделали его более гибким и функциональным инструментом для управления аппаратными компонентами компьютера, что способствовало повышению производительности и эффективности работы пользователей.

В операционной системе Windows XP, выпущенной в 2001 году, Диспетчер устройств был усовершенствован с целью улучшения интерфейса и навигации, а также расширения функциональности (рисунок 1.4).

Диспетчер устройств в Windows XP получил обновленный и более интуитивно понятный интерфейс, что делал его использование более удобным

для пользователей. Навигация по различным категориям устройств стала более простой и интуитивной, что позволяло пользователям легко находить нужные устройства и информацию о них.

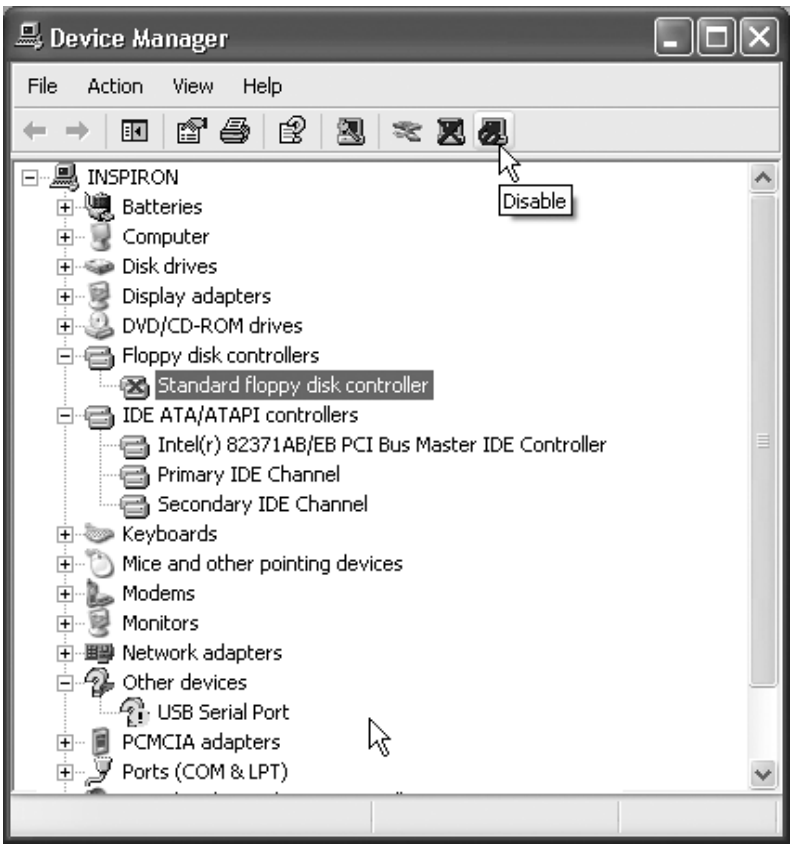

Рисунок 1.4 – Диспетчер устройств в Windows XP

В Windows XP была добавлена функция отображения скрытых устройств в Диспетчере устройств. Это позволяло пользователям просматривать и управлять устройствами, которые могли быть скрыты или неактивны по различным причинам. Такие устройства могли включать в себя запасные порты, отключенные устройства или устройства, которые ранее были подключены к компьютеру.

Были добавлены дополнительные функции управления устройствами через Диспетчер устройств. Это включало возможность обновления драйверов, изменения параметров устройств и управления энергосбережением. Пользователи могли легко выполнять эти действия без необходимости использования сторонних программ или инструментов.

Эти улучшения в Диспетчере устройств Windows XP сделали его более функциональным и удобным инструментом для управления аппаратными компонентами компьютера. Улучшенная навигация и дополнительные

функции управления помогали пользователям более эффективно контролировать все компоненты своего компьютера.

В операционной системе Windows Vista, выпущенной в 2006 году, Диспетчер устройств претерпел изменения, которые сделали его более удобным и информативным для пользователей (рисунок 1.5).

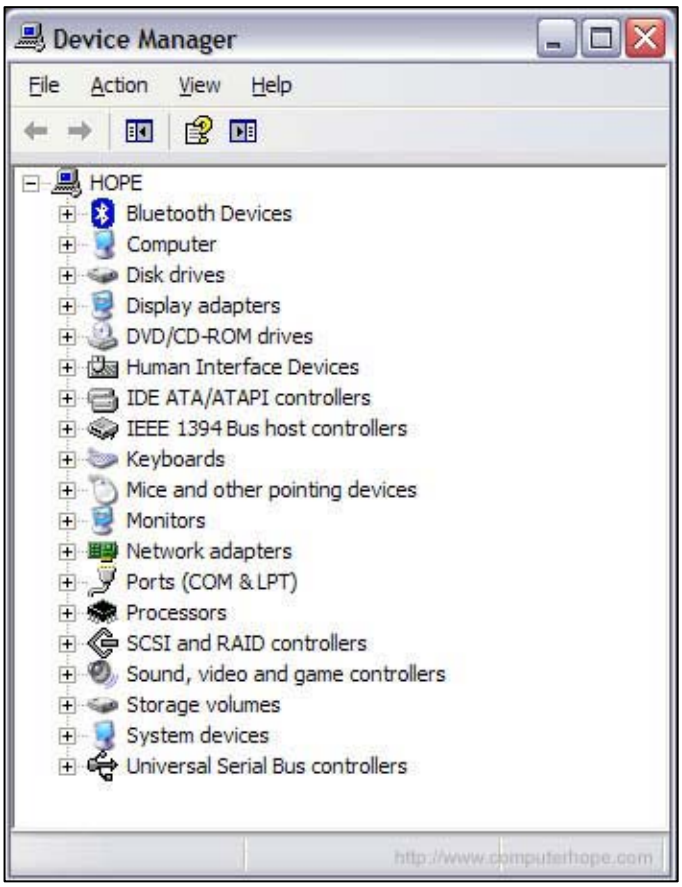

Рисунок 1.5 – Диспетчер устройств в Windows Vista

Для повышения удобства использования были внесены изменения в интерфейс Диспетчера устройств. Это включало обновленный дизайн, более интуитивную навигацию и улучшенную организацию информации о устройствах. Пользователи могли легко находить нужные устройства и получать к ним доступ без лишних усилий.

Были добавлены дополнительные сведения о свойствах устройств, что позволяло пользователям получать более подробную информацию о каждом компоненте своего компьютера. Это включало технические характеристики, статус работы устройства и другие полезные данные, которые помогали лучше понять его функциональность и состояние.

В Windows Vista была улучшена система управления драйверами

устройств [7]. Это включало более удобный и надежный механизм обновления и установки драйверов, что позволяло пользователям легче поддерживать свое оборудование в актуальном состоянии и избегать проблем с совместимостью.

Эти изменения в Диспетчере устройств Windows Vista сделали его более удобным и информативным инструментом для управления аппаратными компонентами компьютера. Улучшенный интерфейс, расширенные сведения о свойствах устройств и улучшенная система управления драйверами помогли пользователям эффективнее контролировать своё оборудование и решать возникающие проблемы.

В операционной системе Windows 7, выпущенной в 2009 году, Диспетчер устройств получил ряд значительных улучшений, которые сделали его более функциональным и удобным для пользователей (рисунок 1.6).

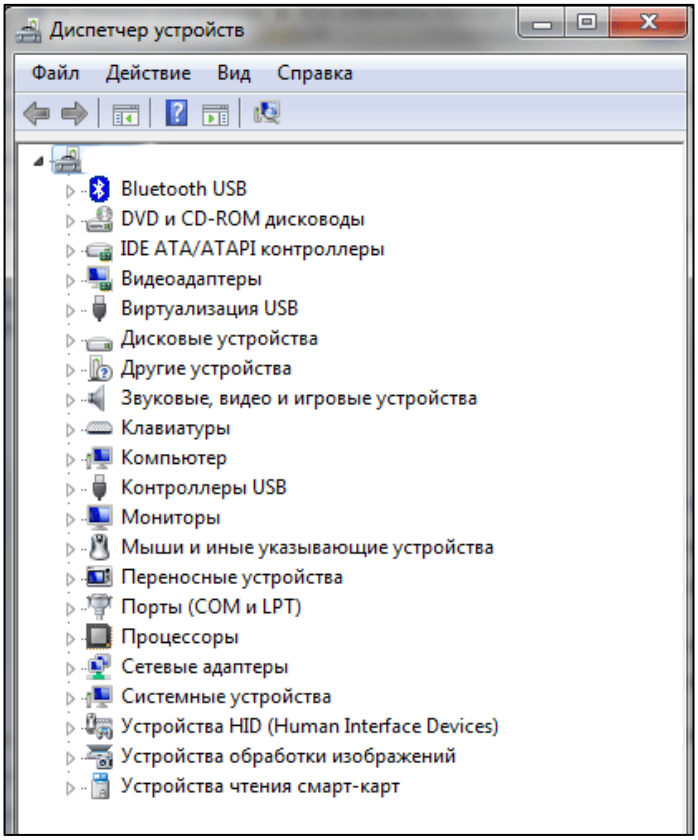

Рисунок 1.6 – Диспетчер устройств в Windows 7

Улучшенное управление устройствами Plug and Play: Windows 7 предложил улучшенное управление устройствами Plug and Play, что позволило более надежно и эффективно обрабатывать подключаемые устройства. Это включало в себя более быстрое обнаружение новых устройств, автоматическую установку необходимых драйверов и более

стабильное подключение и отключение устройств без перезагрузки компьютера.

Windows 7 предоставил дополнительные инструменты для управления драйверами устройств. Это включало возможность быстро обновлять, устанавливать и откатывать драйверы для устройств, что помогало пользователям решать проблемы совместимости и повышать производительность своего оборудования.

Диспетчер устройств в Windows 7 был лучше интегрирован с другими системными инструментами, что упростило процесс диагностики и управления устройствами. Это включало возможность быстро переходить к другим инструментам, таким как Диспетчер задач или Панель управления, для выполнения дополнительных действий по управлению устройствами.

Windows 7 предложил расширенную поддержку для мультимедийных и сетевых устройств, что позволяло пользователям более эффективно управлять своими аудио-, видео- и сетевыми устройствами. Это включало в себя дополнительные опции настройки и расширенные возможности контроля за работой таких устройств.

Эти улучшения сделали Диспетчер устройств в Windows 7 более мощным инструментом для управления аппаратными компонентами компьютера. Улучшенное управление устройствами Plug and Play, инструменты для управления драйверами, улучшенная интеграция с системными инструментами и расширенная поддержка мультимедийных и сетевых устройств помогли пользователям лучше контролировать своё оборудование и решать возникающие проблемы.

В операционной системе Windows 8, выпущенной в 2012 году, Диспетчер устройств получил несколько небольших изменений в интерфейсе и функциональности (рисунок 1.7), чтобы лучше соответствовать новому дизайну операционной системы, который был ориентирован на сенсорные экраны и мобильные устройства.

Диспетчер устройств в Windows 8 был обновлен в соответствии с новым интерфейсом Metro (теперь называемым Modern UI). Интерфейс стал более плоским и простым, с большим упором на использование крупных плиток и жестов с сенсорными устройствами.

Windows 8 была разработана с учетом увеличенного использования сенсорных экранов, поэтому Диспетчер устройств был адаптирован для более удобного использования на устройствах с сенсорными интерфейсами. Это включало в себя увеличение размеров кнопок и элементов управления для улучшения нажатия пальцем.

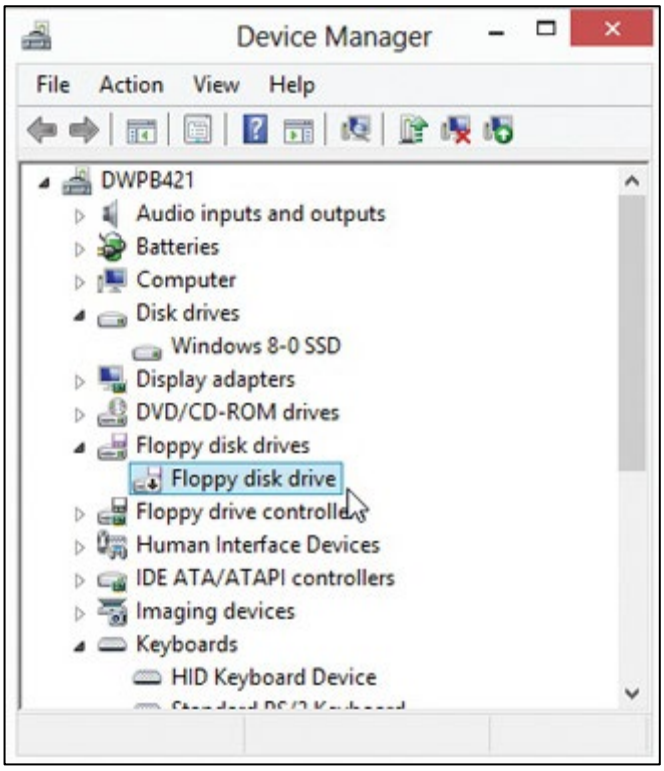

Рисунок 1.7 – Диспетчер устройств в Windows 8

Несмотря на изменения в интерфейсе, основные возможности Диспетчера устройств остались прежними. Пользователи по-прежнему могли просматривать список установленных устройств, проверять их состояние и ресурсы, а также управлять драйверами устройств и решать проблемы совместимости.

Диспетчер устройств в Windows 8 был тесно интегрирован с другими функциями операционной системы, такими как Центр управления, что обеспечивало более удобный доступ к управлению устройствами и настройкам системы [8].

Хотя изменения в Диспетчере устройств Windows 8 были в основном косметическими и ориентированными на новый пользовательский интерфейс, они помогли обеспечить более единый и интуитивно понятный опыт использования операционной системы на всех устройствах, включая сенсорные устройства.

В операционной системе Windows 10, выпущенной в 2015 году, Диспетчер устройств остался ключевым инструментом для управления аппаратными компонентами компьютера. В этой версии были внесены некоторые улучшения в интерфейс и функциональность с целью улучшения удобства использования (рисунок 1.8).

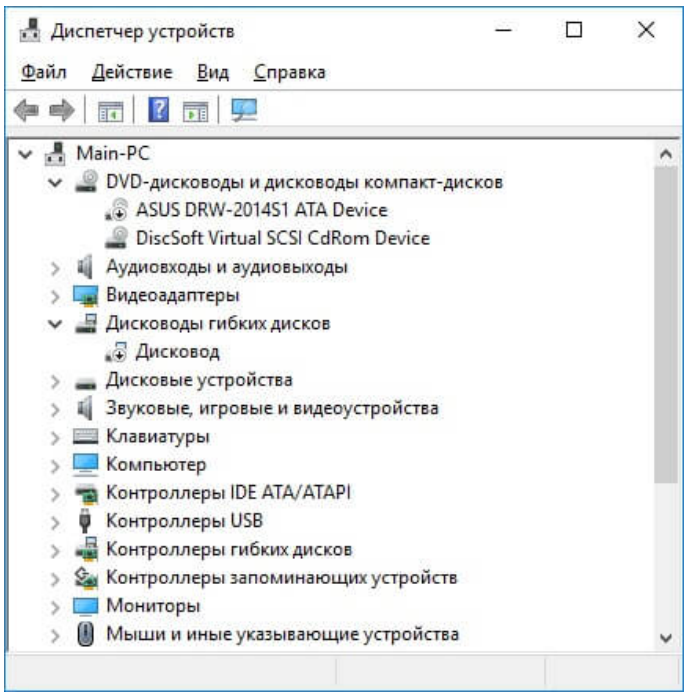

Рисунок 1.8 – Диспетчер устройств в Windows 10

Диспетчер устройств в Windows 10 получил некоторые косметические улучшения, такие как обновленный дизайн и более современный стиль. Это включало в себя изменения в цветовой палитре, шрифтах и стилях элементов управления, что делало интерфейс более приятным для пользователей.

В Windows 10 была внедрена улучшенная система поиска и фильтрации устройств в Диспетчере устройств. Это позволяло пользователям быстро находить нужные устройства с помощью различных критериев, таких как тип устройства, производитель или статус.

Windows 10 предоставляла поддержку новых типов устройств, которые стали доступными к моменту ее выпуска. Диспетчер устройств позволял пользователям управлять этими новыми устройствами и обеспечивал совместимость с ними.

Windows 10 включала улучшенную систему управления драйверами, что обеспечивало более стабильную работу аппаратных компонентов компьютера. Диспетчер устройств позволял пользователям устанавливать, обновлять и удалять драйверы для устройств с помощью удобного интерфейса.

Диспетчер устройств был тесно интегрирован с другими функциями операционной системы Windows 10, такими как Центр уведомлений и Параметры. Это обеспечивало более удобный доступ к управлению устройствами и настройкам системы из различных частей операционной системы.

Эти изменения в Диспетчере устройств Windows 10 помогли сделать управление аппаратными компонентами компьютера более эффективным и удобным для пользователей, что способствовало повышению общего опыта использования операционной системы.

Windows 11 была выпущена Microsoft в октябре 2021 года. В этой операционной системе Диспетчер устройств остается важным инструментом для управления аппаратными компонентами компьютера. Несмотря на то, что Windows 11 имеет некоторые изменения в интерфейсе и дизайне по сравнению с предыдущими версиями, основные функциональности и возможности Диспетчера устройств остаются схожими с теми, что присутствовали в Windows 10.

Одним из основных изменений, которые заметны в Windows 11, является обновленный пользовательский интерфейс с новыми элементами дизайна (рисунок 1.9). Диспетчер устройств также получил некоторые косметические улучшения, такие как обновленные иконки и элементы управления, чтобы соответствовать новому дизайну операционной системы.

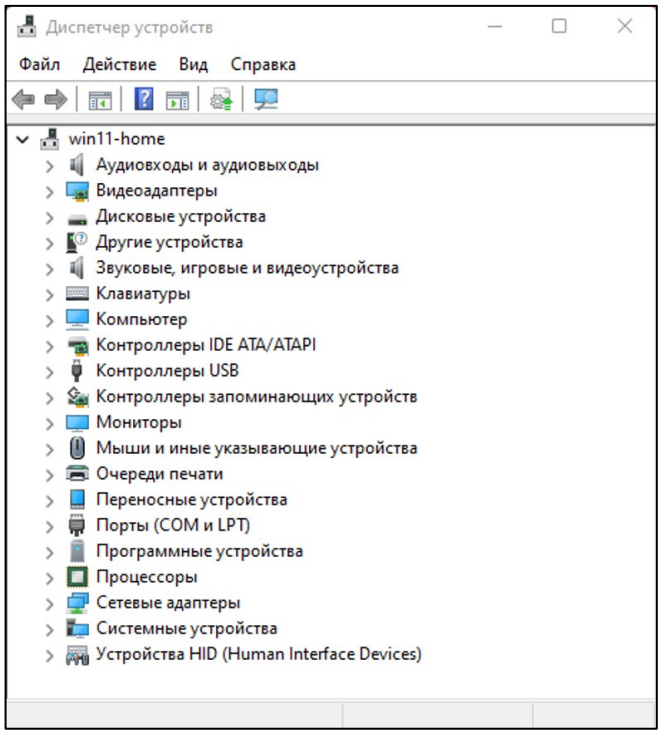

Рисунок 1.9 – Диспетчер устройств в Windows 11

Однако в целом функциональность Диспетчера устройств в Windows 11 остается привычной для пользователей Windows, предоставляя возможность просмотра, управления и настройки аппаратных компонентов компьютера, а также обнаружение и устранение проблем с оборудованием.

1.3 Анализ производительности подсистем компьютера

Подсистемы компьютера представляют собой различные компоненты и уровни аппаратного и программного обеспечения, которые работают вместе для обеспечения функциональности компьютерной системы. Они могут включать в себя [9]:

– центральный процессор (CPU), отвечает за обработку данных и выполнение инструкций, является "мозгом" компьютера;

– оперативная память (RAM), хранит данные и инструкции, к которым быстро требуется доступ процессору;

– жесткий диск (HDD/SSD), используется для долгосрочного хранения данных и программ;

– графический процессор (GPU), отвечает за обработку графики и видео;

– системная шина (System Bus), обеспечивает связь между различными компонентами системы.

Помимо перечисленных, подсистемы также включают в себя подсистемы ввода-вывода (например, клавиатура, мышь, монитор), подсистемы хранения данных (например, жесткие диски, USB-накопители) и другие аппаратные и программные компоненты, которые выполняют различные функции в компьютерной системе.

Можно представить компьютерное оборудование как состоящее из трех отдельных подсистем, как показано на рисунке 1.10.

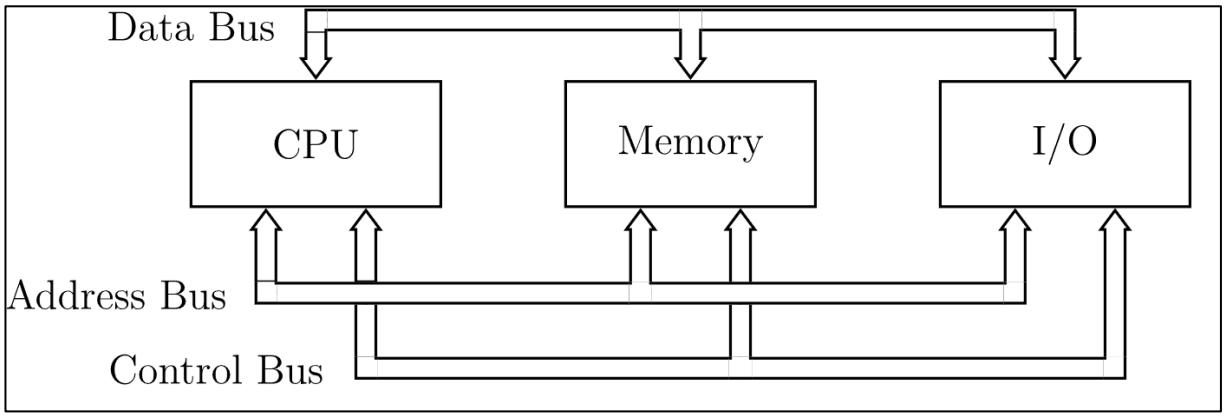

Рисунок 1.10 – Подсистемы компьютера

Центральный процессор (CPU) – управляет большинством функций компьютера, выполняет арифметические и логические операции и содержит небольшой объем очень быстрой памяти.

Память (Memory) – обеспечивает хранение инструкций для

центрального процессора и данных, которыми он управляет.

Ввод/вывод (I/O) – Осуществляет связь с внешним миром и с устройствами массового хранения данных (например, дисками, сетью, USB).

Шина (Bus) – канал связи по протоколу, точно определяющим, как этот канал используется.

Анализ производительности подсистем компьютера представляет собой основательное изучение и оценку работы различных компонентов и функциональных блоков компьютерной системы. Целью этого анализа является определение эффективности, надежности и пропускной способности системы в целом, а также её отдельных компонентов.

Этот процесс включает в себя:

1 Изучение компонентов. Для проведения анализа производительности необходимо начать с изучения каждой подсистемы компьютера, таких как CPU, RAM, GPU, HDD/SSD и других. Это позволяет понять, как каждый компонент взаимодействует с другими и как его работа влияет на общую производительность системы. Например, оценка процессора может включать в себя изучение таких параметров, как частота процессора, количество ядер и потоков, а также использование кэш-памяти.

2 Оценка использования ресурсов. Важной частью анализа является измерение использования различных ресурсов компьютера, таких как процессорное время, объем занятой оперативной памяти, доступное дисковое пространство и пропускная способность сети. Это помогает выявить узкие места и проблемные компоненты, которые могут замедлять работу системы. Например, высокая загрузка процессора или переполненная оперативная память может указывать на проблемы с производительностью.

3 Измерение скорости выполнения операций. Оценка времени, затрачиваемого на выполнение различных операций и задач, позволяет определить, насколько быстро и отзывчиво работает компьютер. Это включает в себя такие задачи, как загрузка операционной системы, запуск приложений, копирование файлов и т. д. Повышение скорости выполнения этих операций может улучшить общее впечатление пользователя от работы с системой.

4 Оценка общей отзывчивости системы. Анализируя общее восприятие пользователя относительно отзывчивости системы при выполнении различных задач, можно выявить области, требующие улучшения. Это включает в себя скорость открытия приложений, реакцию на пользовательские действия и отображение графического контента. Улучшение общей отзывчивости системы способствует повышению удовлетворенности пользователей и эффективности работы.

5 Идентификация проблем и улучшений. На основе результатов анализа производительности выявляются проблемные компоненты и узкие места в системе. Это может включать в себя проблемы с оборудованием, неэффективное использование ресурсов или неоптимизированные настройки. Разработка стратегии улучшения производительности включает в себя обновление оборудования, оптимизацию настроек системы или использование специализированных программных решений, направленных на решение выявленных проблем.

В операционной системе Windows существует несколько инструментов для анализа производительности подсистем компьютера. Вот некоторые из них [10]:

1 Диспетчер задач (Task Manager) – это стандартный инструмент в Windows, который позволяет отслеживать процессы, запущенные на компьютере, и контролировать использование ресурсов, таких как процессор, оперативная память, дисковое пространство и сетевая активность. Для открытия Диспетчера задач можно использовать комбинацию клавиш «Ctrl + Shift + Esc».

2 Монитор ресурсов (Resource Monitor), предоставляет более подробную информацию о процессах и ресурсах компьютера по сравнению с Диспетчером задач. Он позволяет отслеживать активность ЦП, дисков, сети и памяти в реальном времени. Его можно найти в Диспетчере задач, выбрав вкладку «Производительность», а затем «Открыть монитор ресурсов".

3 Проводник ресурсов (Resource Explorer) – это инструмент, который позволяет анализировать файлы, папки, реестр и другие ресурсы компьютера для выявления проблем, таких как утечки памяти, перегруженные диски и так далее. Он предоставляет подробную информацию о потреблении ресурсов различными процессами и приложениями. Resource Explorer – это часть инструментария Windows Assessment and Deployment Kit (ADK), который предоставляет возможности для анализа ресурсов и производительности в Windows.

4 Диагностика ресурсов (Performance Monitor) – это инструмент для мониторинга и анализа производительности компьютера на основе собираемых данных о ресурсах, таких как процессор, память, дисковое пространство и сеть. Он позволяет создавать графики и отчеты о производительности системы с течением времени.

5 Windows PowerShell – это мощный инструмент командной строки, который предоставляет широкие возможности для анализа производительности и управления ресурсами компьютера с помощью команд

и сценариев.

Эти инструменты позволяют пользователям Windows мониторить и анализировать производительность компьютера, выявлять узкие места и проблемы, а также принимать меры для их устранения.

Диспетчер задач (Task Manager) – это инструмент в операционных системах Windows, который позволяет пользователям просматривать и управлять активными процессами и службами на своем компьютере.

Диспетчер задач впервые появился в операционной системе Windows NT 4.0 в 1996 году (рисунок 1.11). Это стало значительным шагом в обеспечении пользователей возможностью более детально контролировать и управлять работой своих компьютеров. Со временем, Диспетчер задач стал одним из наиболее важных и часто используемых инструментов в Windows.

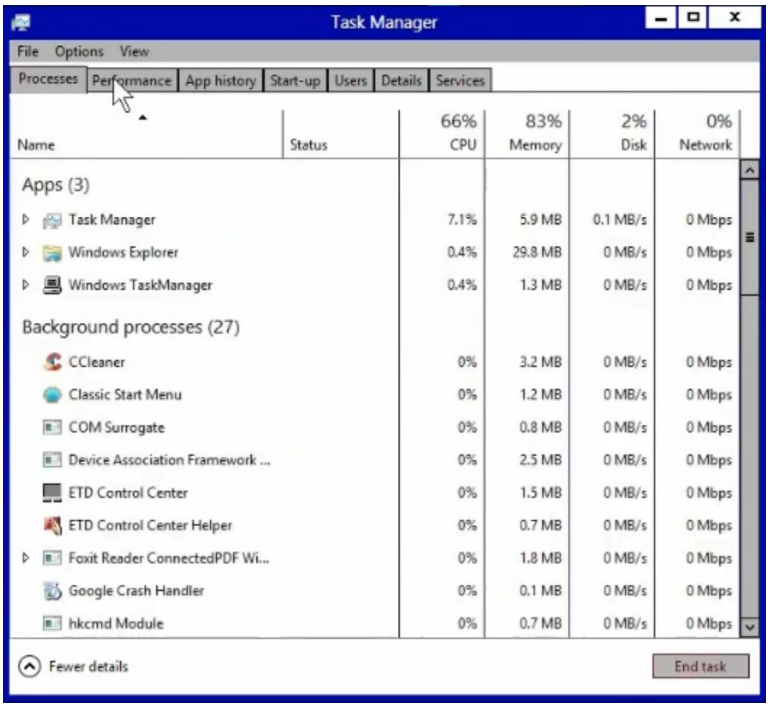

Рисунок 1.11 – Диспетчер задач в Windows NT 4.0

Следующие версии операционной системы Windows, включая Windows 2000, Windows XP, Windows Vista, Windows 7, Windows 8 и Windows 10, все включали в себя Диспетчер задач, каждая из которых принесла свои улучшения в функциональность и интерфейс этого инструмента. Например, с появлением Windows Vista и Windows 7 были добавлены дополнительные функции, такие как анализатор производительности и управление автозагрузкой, что позволило пользователям получить более полный контроль над своей системой. А с появлением Windows 8 и Windows 10 интерфейс

Диспетчера задач был пересмотрен, чтобы соответствовать современным требованиям дизайна, а также были добавлены новые функции, такие как возможность поиска онлайн-информации о процессах.

Функции, которые предоставляет Диспетчер задач:

1 Отображение активных процессов. Диспетчер задач предоставляет подробную информацию о всех процессах, запущенных на компьютере. Это включает в себя название процесса, уникальный идентификатор процесса (PID), а также информацию о том, сколько центрального процессора, оперативной памяти, дисковых и сетевых ресурсов использует каждый процесс. Это позволяет пользователям легко определить, какие приложения и процессы потребляют больше всего ресурсов и могут приводить к замедлению работы системы.

2 Завершение процессов. В случае, если какой-либо процесс перестал отвечать или замедляет работу системы, пользователи могут завершить его с помощью Диспетчера задач. Просто выбрав нужный процесс и нажав кнопку "Завершить задачу", пользователи могут освободить занятые ресурсы и улучшить производительность компьютера.

3 Мониторинг ресурсов. Диспетчер задач позволяет отслеживать использование ресурсов компьютера в реальном времени. Это включает в себя мониторинг использования CPU, оперативной памяти, жесткого диска и сетевого трафика. Благодаря этой функции пользователи могут наблюдать за тем, какие процессы потребляют больше всего ресурсов, и принимать соответствующие меры для оптимизации работы системы.

4 Управление автозагрузкой. Во вкладке "Автозагрузка" Диспетчера задач пользователи могут управлять программами и службами, которые автоматически запускаются при загрузке операционной системы. Это позволяет оптимизировать время загрузки компьютера и предотвратить запуск ненужных или вредоносных приложений.

5 Анализ производительности (рисунок 1.12). В некоторых версиях Windows Диспетчер задач содержит дополнительные инструменты для анализа производительности. Например, он может показывать графики использования ресурсов во времени, что помогает пользователям наглядно оценить изменения производительности и выявить возможные проблемы.

6 Поиск онлайн-информации. В последних версиях Windows Диспетчер задач иногда предлагает возможность искать онлайн-информацию о выбранном процессе. Это позволяет пользователям узнать больше о процессе, его связи с другими программами и возможных рисках для системы.

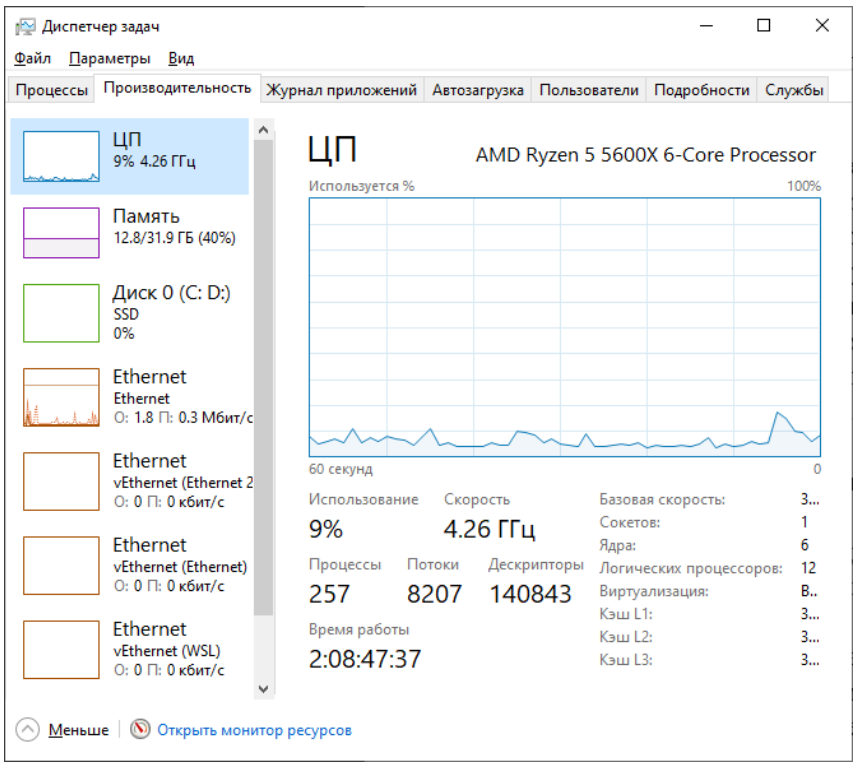

Рисунок 1.12 – Вкладка «Производительность» в Диспетчере задач

Диспетчер задач является важным инструментом для мониторинга и управления компьютерной системой в реальном времени, что делает его незаменимым помощником для пользователей Windows.

Монитор ресурсов (рисунок 1.13) был впервые введен в операционной системе Windows Vista в 2006 году [11]. Он был создан в качестве усовершенствованной версии предыдущего инструмента – Диспетчера ресурсов, который появился в Windows XP. Монитор ресурсов получил ряд улучшений и новых функций, сделав его более мощным и информативным инструментом для мониторинга производительности компьютера.

Функциональность Монитора ресурсов:

1 Отслеживание процессора (CPU). Монитор ресурсов предоставляет подробную информацию о загрузке процессора и активности каждого запущенного процесса. Пользователи могут видеть процент использования CPU и идентифицировать приложения или процессы, которые потребляют больше всего ресурсов. Это позволяет выявить приложения, вызывающие значительную нагрузку на процессор, и принять меры для оптимизации работы системы, например, завершить или приостановить эти процессы.

2 Мониторинг оперативной памяти (RAM). Инструмент отображает текущее использование оперативной памяти системой и каждым запущенным процессом. Пользователи могут видеть объем выделенной оперативной

памяти, используемый каждым процессом, и общий объем доступной и свободной памяти. Это позволяет оптимизировать использование оперативной памяти, выявлять утечки памяти и определять, какие процессы требуют дополнительной памяти для более эффективной работы.

3 Отслеживание жестких дисков (HDD/SSD). Монитор ресурсов предоставляет информацию о работе жестких дисков, включая активность чтения и записи данных, а также уровень загрузки дисков. Пользователи могут видеть количество операций чтения и записи на диске, скорость передачи данных и задержку в доступе к диску. Это позволяет выявить процессы, которые могут быть причиной долгого времени отклика системы из-за интенсивной дисковой активности, и оптимизировать использование дисковых ресурсов.

4 Мониторинг сетевого трафика. Монитор ресурсов также позволяет отслеживать сетевую активность компьютера, включая количество переданных и полученных данных, скорость передачи и задержку сети. Пользователи могут видеть сетевую активность по каждому процессу или приложению, что помогает контролировать сетевую нагрузку, выявлять проблемы с сетевым подключением и оптимизировать использование сетевых ресурсов.

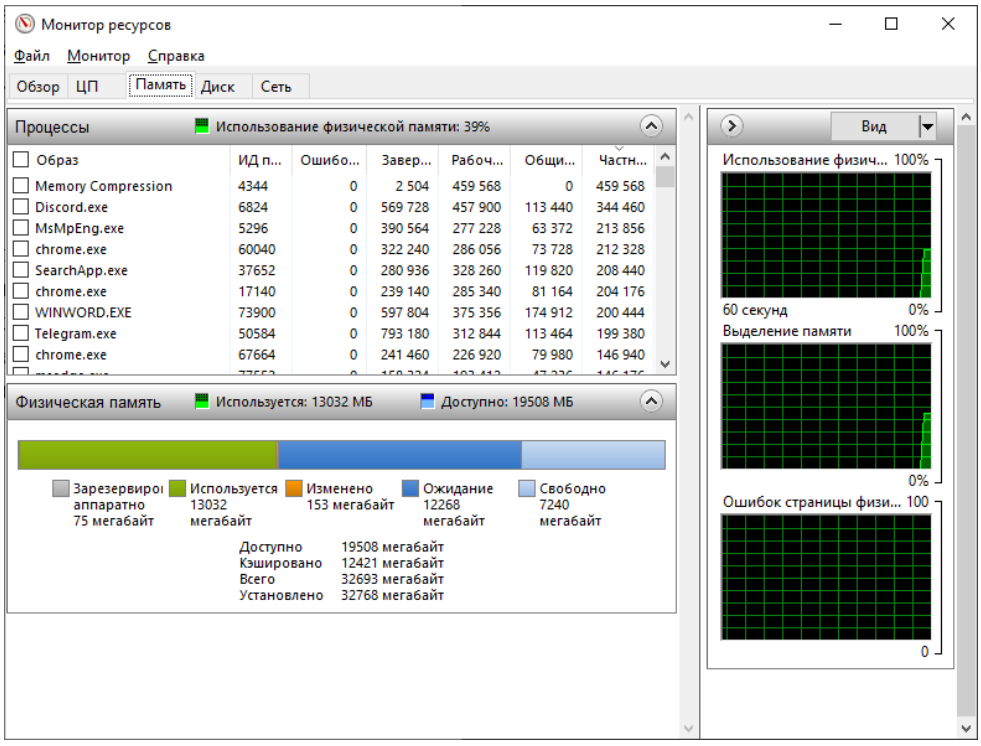

Рисунок 1.13 – Монитор ресурсов

Монитор ресурсов является важным инструментом для анализа и оптимизации производительности компьютера в операционной системе Windows. Он предоставляет пользователю детальную информацию о том, как используются ресурсы системы, что помогает выявлять и устранять проблемы, связанные с нагрузкой на процессор, память, диски и сеть.

Анализ производительности с помощью PowerShell позволяет пользователям получать информацию о различных аспектах работы компьютерной системы, включая процессор, оперативную память, дисковое пространство, сетевую активность и другие ресурсы. PowerShell предоставляет широкий спектр командлетов и инструментов для сбора и анализа данных о производительности системы.

Ниже приведены некоторые примеры командлетов PowerShell для анализа производительности [12]:

1 Get-Counter. Этот командлет позволяет получить данные о различных счетчиках производительности, таких как загрузка процессора, использование памяти, скорость передачи данных по сети и другие. Например: «*Get-Counter '\Processor(*)\% Processor Time' -Continuous».*

Get-Process/ Этот командлет позволяет получить информацию о запущенных процессах, включая их идентификаторы, загрузку процессора, объем занимаемой памяти и другие параметры. Например: *«Get-Process | Sort-Object CPU -Descending | Select-Object -First 5».*

Get-WmiObject: Этот командлет позволяет получить информацию о различных аспектах системы, используя Windows Management Instrumentation (WMI). Например, можно получить информацию о дисковых разделах: *«Get-WmiObject Win32_LogicalDisk | Select-Object DeviceID, Size, FreeSpace».*

Measure-Command: Этот командлет позволяет измерить время выполнения команд или скриптов в PowerShell. Это может быть полезно для оценки производительности выполнения определенных операций. Например: *«Measure-Command { Get-Process }».*

Test-Connection: Этот командлет позволяет проверить доступность удаленного компьютера и измерить время отклика. Например: «*Test-Connection -ComputerName google.com -Count 5»*.

Эти командлеты и многие другие инструменты PowerShell предоставляют широкие возможности для анализа производительности компьютерной системы и выявления проблем, которые могут повлиять на её работу.

1.4 Ограниченное управление устройствами

Ограниченное управление устройствами в операционной системе Windows представляет собой механизм контроля и ограничения доступа к аппаратным компонентам компьютера. Этот подход позволяет администраторам и пользователям ограничивать доступ к определенным устройствам для обеспечения безопасности и контроля за использованием ресурсов.

Ограниченное управление устройствами предоставляет следующие возможности:

1 Контроль доступа. Ограниченное управление устройствами позволяет администраторам устанавливать гибкие политики доступа к устройствам для различных пользователей или групп пользователей. Например, можно разрешить определенным пользователям доступ к принтерам, сканерам или съемным USB-устройствам, в то время как другим пользователям этот доступ будет ограничен. Это помогает предотвратить несанкционированный доступ к чувствительным данным или ресурсам компьютера.

2 Ограничение функциональности. Администраторы могут ограничивать функциональность некоторых устройств для повышения безопасности и предотвращения потенциальных угроз. Например, можно запретить запись на USB-накопителях или блокировать доступ к определенным портам ввода-вывода, чтобы предотвратить передачу вредоносных программ или утечку конфиденциальной информации.

3 Управление ресурсами. Ограниченное управление устройствами позволяет оптимизировать использование ресурсов компьютера путем ограничения доступа к устройствам, которые потребляют больше всего ресурсов. Например, высокоскоростные принтеры или сетевые адаптеры могут потреблять значительное количество пропускной способности сети или оперативной памяти. Путем ограничения доступа к этим устройствам администраторы могут предотвратить перегрузку сети или системы и обеспечить более стабильную работу компьютера.

4 Мониторинг и аудит использования устройств. Некоторые средства управления устройствами в Windows позволяют администраторам мониторить и аудитировать использование устройств. Это позволяет отслеживать активность пользователей, определять потенциальные угрозы безопасности и выявлять несанкционированные попытки доступа к устройствам.

5 Централизованное управление. В некоторых версиях Windows предусмотрены инструменты централизованного управления устройствами,

которые позволяют администраторам настраивать и управлять политиками безопасности для нескольких компьютеров или пользователей из одного центра управления. Это обеспечивает единое и эффективное управление устройствами в корпоративной среде.

Ограниченное управление устройствами в Windows достигается с помощью различных методов и инструментов:

1 Групповые политики. Групповые политики – это мощный инструмент управления доступом к устройствам на уровне компьютера или пользовательских учетных записей в среде Windows. Администраторы могут создавать и применять политики, определяющие, какие устройства разрешено использовать для конкретных пользователей или групп пользователей. Например, можно разрешить доступ к USB-накопителям только определенным пользователям или запретить использование определенных сетевых принтеров. Групповые политики применяются ко всем компьютерам в домене и обеспечивают единое и централизованное управление доступом к устройствам.

2 Драйверы устройств. Некоторые устройства могут поставляться с драйверами, которые включают функции ограничения доступа или контроля за использованием устройства. Например, драйверы принтера могут предоставлять возможность администраторам ограничивать доступ к определенным функциям принтера, таким как двусторонняя печать или печать цветных документов. Это позволяет администраторам более тонко настраивать использование устройств и обеспечивать соответствие политикам безопасности.

3 Центр управления устройствами. Windows предоставляет центр управления устройствами, который позволяет пользователям и администраторам просматривать и управлять установленными устройствами. С помощью этого инструмента можно просматривать список устройств, устанавливать драйверы, изменять параметры устройств и применять политики безопасности. Центр управления устройствами предоставляет удобный интерфейс для управления устройствами и обеспечивает простоту настройки и конфигурирования устройств в системе Windows.

Ограниченное управление устройствами (Device Management) в операционной системе Windows имеет долгую историю, начиная с ранних версий Windows NT и Windows 95. В это время компьютеры были менее сетевыми, и угрозы безопасности были менее развитыми, поэтому функции управления устройствами были простыми и ограниченными. Однако с появлением сетевых технологий и повышением требований к безопасности,

этот аспект стал более значимым для обеспечения безопасности и управления ресурсами в корпоративных средах.

В ранних версиях Windows, таких как Windows NT и Windows 95, механизмы ограниченного управления устройствами были простыми и позволяли администраторам устанавливать основные правила доступа к устройствам и контролировать их использование в некоторой степени. Однако с развитием технологий и повышением требований к безопасности, данные механизмы стали совершенствоваться и становиться более эффективными.

В Windows XP были расширены и усовершенствованы механизмы ограниченного управления устройствами. Администраторы получили возможность более тонко настраивать правила доступа к устройствам, контролировать их использование и применять политики безопасности на уровне группы или отдельных пользователей.

Windows Vista внесла изменения в интерфейс и функциональность ограниченного управления устройствами, чтобы сделать его более удобным и информативным для пользователей. Были внедрены новые возможности, такие как расширенные сведения о свойствах устройств и улучшенная система управления драйверами.

С появлением Windows 7 были добавлены новые возможности в ограниченное управление устройствами. Появилось улучшенное управление устройствами Plug and Play, а также инструменты для управления драйверами, что позволило пользователям более эффективно контролировать свое оборудование и решать проблемы, связанные с ним.

В Windows 10 ограниченное управление устройствами остается ключевым инструментом для управления аппаратными компонентами компьютера. Были внесены некоторые улучшения в интерфейс и функциональность для повышения удобства использования, такие как улучшенная система поиска и фильтрации устройств.

В настоящее время ограниченное управление устройствами остается важной составляющей стратегии информационной безопасности и управления IT-ресурсами в корпоративных средах. С развитием облачных технологий, мобильных устройств и интернета вещей, этот аспект становится еще более актуальным, поскольку администраторам необходимо контролировать доступ к различным типам устройств и обеспечивать безопасность корпоративных данных в различных средах использования.

2 ПЛАТФОРМА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

2.1 Структура и архитектура платформы

В рамках курсового проекта используется операционная система Windows 10, которая является одной из наиболее распространенных и широко используемых операционных систем в мире. Windows 10 предоставляет обширный набор функций и инструментов для разработки и запуска различных программ и приложений.

Структура и архитектура платформы Windows 10 позволяют эффективно управлять ресурсами компьютера, обеспечивать безопасность данных и обеспечивать высокую производительность системы. Эта операционная система поддерживает широкий спектр аппаратного и программного обеспечения, что делает ее идеальным выбором для выполнения курсовых проектов в области информационных технологий.

В контексте курсового проекта структура и архитектура Windows 10 играют важную роль для понимания основных принципов функционирования операционной системы и работы с ее компонентами. Например, изучение структуры помогает разобраться в том, как создать приложение, анализирующее и управляющее устройствами с помощью Диспетчера устройств. Также понимание архитектуры операционной системы важно для разработки инструментов мониторинга ресурсов, которые используют знания о структуре системы для эффективного отслеживания и анализа производительности компьютера.

Windows NT (New Technology) – это семейство операционных систем, разработанных и выпущенных корпорацией Microsoft. Windows NT была создана как полностью новая операционная система, отличающаяся от более ранних версий Windows, таких как Windows 95 и Windows 98, и предназначенная для использования в корпоративной среде и на серверах [13].

Архитектура Windows NT присуща семейству операционных систем (ОС) на ядре Windows NT. Это следующие операционные системы: Windows NT 3.1, Windows NT 3.5, Windows NT 3.51, Windows NT 4.0, Windows 2000, Windows XP, Windows Server 2003, Windows Vista, Windows Server 2008, Windows 7, Windows 8, Windows 10 и Windows 11.

Все они являются операционными системами с вытесняющей многозадачностью, разработаны для работы как с однопроцессорными, так и с симметричными мультипроцессорными компьютерами. Для обработки запросов ввода-вывода используется пакетноуправляемый ввод-вывод,

который применяет пакеты запросов ввода\вывода (IRP) и асинхронный ввод-вывод.

Архитектура Windows NT (рисунок 2.1) имеет модульную структуру и состоит из двух основных уровней – компоненты, работающие в режиме пользователя, и компоненты режима ядра. Программы и подсистемы, работающие в режиме пользователя, имеют ограничения на доступ к системным ресурсам. Режим ядра имеет неограниченный доступ к системной памяти и внешним устройствам. Ядро системы NT называют гибридным ядром или макроядром. Архитектура включает в себя само ядро, уровень аппаратных абстракций (HAL), драйверы и ряд служб (Executives), которые работают в режиме ядра (Kernel-mode drivers) или в пользовательском режиме (User-mode drivers).

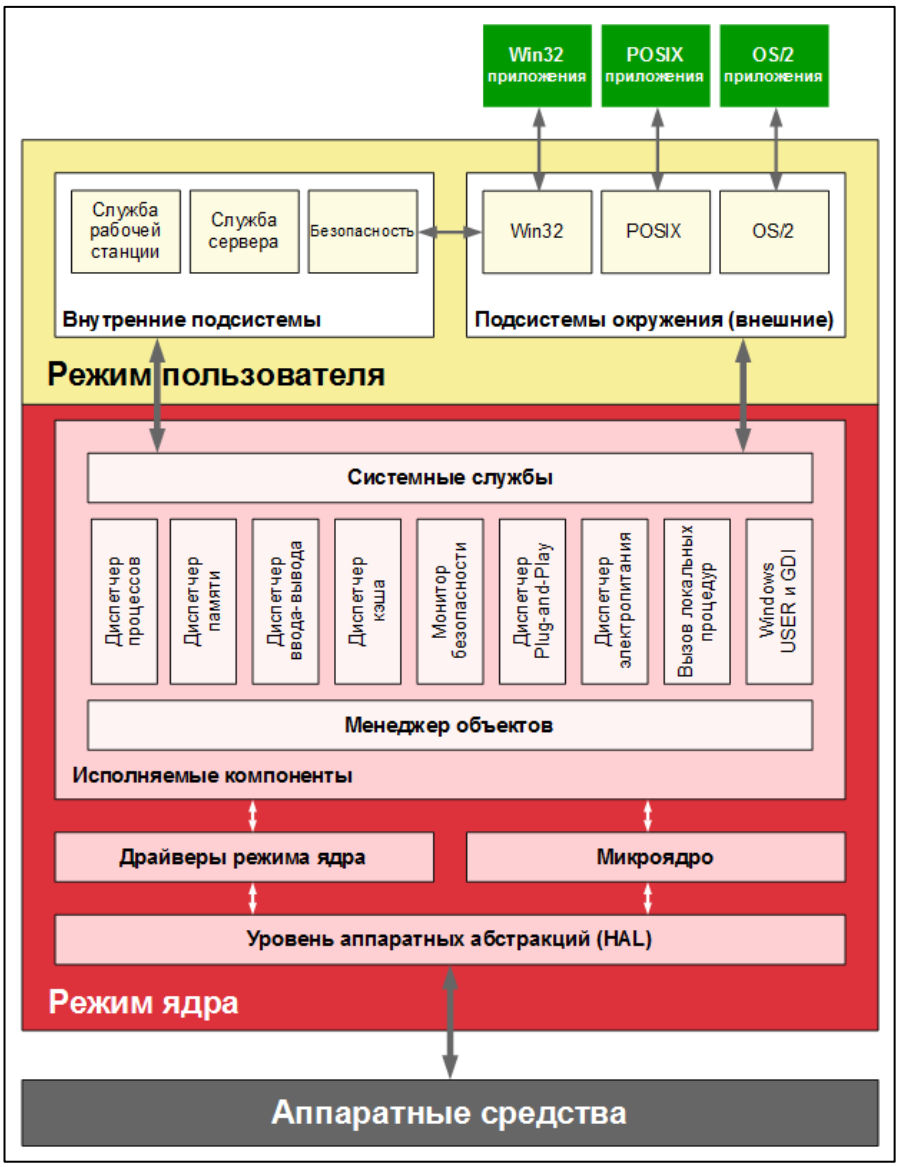

Рисунок 2.1 – Архитектура Windows NT

Режим пользователя состоит из подсистем, которые передают запросы ввода-вывода соответствующему драйверу режима ядра посредством менеджера ввода-вывода. Уровень пользователя состоит из двух подсистем – подсистемы окружения (Environment) и интегральной подсистемы (Integral).

Подсистема окружения разработана для запуска приложений, написанных для разных типов операционных систем. Ни одна из подсистем окружения не имеет прямого доступа к аппаратной части компьютера. Доступ к ресурсам памяти происходит посредством Менеджера виртуальной памяти, который работает в режиме ядра. Также приложения запускаются с меньшим приоритетом, чем процессы режима ядра.

Подсистема окружения состоит из следующих подсистем – подсистема Win32, подсистема OS/2 и подсистема POSIX. Подсистема окружения Win32 запускает 32-разрядные Windows-приложения. Она содержит консоль и поддержку текстового окна, обработку ошибок для всех других подсистем окружения. Поддерживает VDM (Virtual DOS Machine), которая позволяет запускать 16-разрядные DOS и Windows- (Win16) приложения. VDM запускается в своем собственном адресном пространстве и эмулирует систему MS-DOS, запущенную на компьютере с процессором Intel 80486. Программы Win16 запускаются в режиме Win16 VDM. Каждая программа запускается в одном процессе с использованием одного адресного пространства, но для каждой программы используется свой отдельный поток. Однако Windows NT позволяет запускать Win16-программы в отдельных Win16 VDM-процессах, реализуя вытесняющую многозадачность. Процесс подсистемы окружения Win32 – csrss.exe также включает в себя функциональность менеджера окон, то есть обрабатывает входящие события, такие, как нажатие клавиш клавиатуры и мыши, и передает их на обработку соответствующим приложениям. Каждое приложение само производит перерисовку окон в ответ на эти сообщения.

Подсистема окружения OS/2 поддерживает неграфические 16-разрядные приложения операционной системы OS/2 и эмулирует систему OS/2 2.1.x.

Подсистема окружения POSIX поддерживает приложения, написанные в соответствии со стандартом POSIX.1.

Интегрированная подсистема (Integral subsystem) следит за некоторыми функциями операционной системы от имени подсистемы окружения. Состоит из подсистемы безопасности, службы рабочей станции и службы сервера. Служба безопасности обращается с маркерами доступа, разрешает или запрещает доступ к учётной записи пользователя, обрабатывает запросы

авторизации и инициирует процесс входа пользователя в систему. Служба Рабочая станция обеспечивает доступ компьютера к сети – является API для сетевого редиректора (ПО, эмулирующее доступ к удаленной файловой системе как к локальной). Служба Сервер позволяет компьютеру предоставлять сетевые сервисы.

Режим ядра Windows NT имеет полный доступ к аппаратной части компьютера и системным ресурсам. Работает в защищенной области памяти. Управляет памятью и взаимодействием с аппаратной частью. Предотвращает доступ к критическим областям памяти со стороны приложений и служб пользовательского режима. Для выполнения подобных операций процесс пользовательского режима должен попросить режим ядра выполнить её от своего имени [14].

Архитектура x86 поддерживает 4 уровня привилегий – от 0 до 3, но используются только 0-й и 3-й уровень. Режим пользователя использует уровень 3, а режим ядра – 0. Это было сделано для возможности переноса на платформу RISC, которая использует только два уровня привилегий. Режим ядра состоит из исполнительных служб, которые представляют собой различные модули, выполняющие определённые задачи, драйвера ядра, само ядро и уровень аппаратных абстракций HAL.

Исполнительная подсистема работает с вводом-выводом, менеджером объектов, управлением над процессами и безопасностью. Неофициально делится на несколько подсистем – менеджер кэша, менеджер конфигурации, менеджер ввода/вывода, вызов локальных процедур, менеджер памяти, монитор безопасности. Системные службы, то есть системные вызовы, реализованы на этом уровне, за исключением нескольких вызовов, которые вызывают непосредственно ядро для большей производительности. В данном контексте термин «служба» относится к вызываемым подпрограммам, или набору вызываемых подпрограмм. Они отличаются от служб, выполняемых в режиме пользователя, которые в какой-то мере являются аналогом демонов в UNIX-подобных системах.

Менеджер объектов – это исполнительная подсистема, к которой обращаются все остальные модули исполнительной подсистемы, в частности, системные вызовы, когда им необходимо получить доступ к ресурсам Windows NT. Менеджер объектов служит для уменьшения дублирования объектов, что может привести к ошибкам в работе системы. Для менеджера объектов каждый ресурс системы является объектом – будь то физический ресурс типа периферийного устройства, файловой системы или логический ресурс – файл и др. Каждый объект имеет свою структуру, или тип объекта.

Создание объекта делится на две стадии – создание и вставка. Создание – создается пустой объект и резервируются необходимые ресурсы, например, имя в пространстве имен. Если создание пустого объекта произошло успешно, то подсистема, ответственная за создание объекта, заполняет его. Если инициализация успешна, то подсистема заставляет менеджер объектов произвести вставку объекта – то есть сделать его доступным по своему имени или дескриптору.

2.2 Программные интерфейсы платформы

Программные интерфейсы (API) Windows – это набор функций и методов, предоставляемых операционной системой Windows для взаимодействия с приложениями и другими компонентами системы. Они представляют собой наборы команд и инструкций, которые разработчики могут использовать для создания приложений, которые взаимодействуют с различными аспектами операционной системы Windows.

Программные интерфейсы Windows включают широкий спектр функций и возможностей, таких как управление файлами и папками, работа с процессами и потоками, работа с реестром системы, сетевое взаимодействие, графический интерфейс пользователя (GUI), управление устройствами и многое другое.

Native API. Для прикладных программ системой Windows NT предоставляется несколько наборов API. Основной из них – так называемый «родной» API (NT Native API), реализованный в динамически подключаемой библиотеке *`ntdll.dll`* и состоящий из двух частей: системные вызовы ядра NT (функции с префиксами *`Nt`* и *`Zw`*, передающие выполнение функциям ядра ntoskrnl.exe с теми же названиями) и функции, реализованные в пользовательском режиме (с префиксом *`Rtl`*). Часть функций второй группы использует внутри себя системные вызовы; остальные целиком состоят из непривилегированного кода и могут вызываться не только из кода пользовательского режима, но и из драйверов. Кроме функций Native API, в *`ntdll`* также включены функции стандартной библиотеки языка Си.

Официальная документация на Native API весьма скудна, но сообществам энтузиастов удалось методом проб и ошибок собрать достаточно обширные сведения об этом интерфейсе. В частности, в феврале 2000 года опубликована книга Гэри Неббета «Справочник по базовым функциям API Windows NT/2000» (ISBN 1-57870-199-6); в 2002 году она была переведена на русский язык (ISBN 5-8459-0238-X). Источником информации

о Native API может служить Windows DDK, где описаны некоторые функции ядра, доступные посредством Native API, а также изучение кода Windows (обратная разработка) – посредством дизассемблирования либо используя исходные тексты Windows 2000, ставшие доступными в результате утечки, либо используя исходные тексты Windows Server 2003, доступные в рамках программы Windows Research Kernel.

Программы, выполняющиеся до загрузки подсистем, обеспечивающих работу остальных API ОС Windows NT, ограничены использованием Native API. Например, программа *`autochk`*, проверяющая диски при загрузке ОС после некорректного завершения работы, использует только Native API.

Win32 API. Чаще всего прикладными программами для Windows NT используется Win32 API – интерфейс, созданный на основе API ОС Windows 3.1 и позволяющий перекомпилировать существующие программы для 16-битных версий Windows с минимальными изменениями исходного кода. Совместимость Win32 API и 16-битного Windows API настолько велика, что 32-битные и 16-битные приложения могут свободно обмениваться сообщениями, работать с окнами друг друга и т. д. Кроме поддержки функций существовавшего Windows API, в Win32 API был также добавлен ряд новых возможностей, в том числе поддержка консольных программ, многопоточности и объектов синхронизации, таких как мьютексы и семафоры. Документация на Win32 API входит в состав Microsoft Platform SDK и доступна на веб-сайте.

Библиотеки поддержки Win32 API в основном названы так же, как системные библиотеки Windows 3.x, с добавлением суффикса *`32`*: это библиотеки *`kernel32`, `advapi32`, `gdi32`, `user32`, `comctl32`, `comdlg32`, `shell32`* и ряд других. Функции Win32 API могут либо самостоятельно реализовывать требуемую функциональность в пользовательском режиме, либо вызывать описанные выше функции Native API, либо обращаться к подсистеме *`csrss`* посредством механизма LPC, либо осуществлять системный вызов в библиотеку *`win32k`*, реализующую необходимую для Win32 API поддержку в режиме ядра. Четыре перечисленных варианта могут также комбинироваться в любом сочетании: например, функция Win32 API *`WriteFile`* обращается к функции Native API *`NtWriteFile`* для записи в дисковый файл, и вызывает соответствующую функцию csrss для вывода в консоль.

Поддержка Win32 API включена в семейство ОС Windows 9x; кроме того, она может быть добавлена в Windows 3.1x установкой пакета Win32s. Для облегчения переноса существующих Windows-приложений,

использующих для представления строк MBCS-кодировки, все функции Win32 API, принимающие параметрами строки, были созданы в двух версиях: функции с суффиксом A (ANSI) принимают MBCS-строки, а функции с суффиксом W (wide) принимают строки в кодировке UTF-16. В Win32s и Windows 9x поддерживаются только A-функции, тогда как в Windows NT, где все строки внутри ОС хранятся исключительно в UTF-16, каждая A-функция просто преобразует свои строковые параметры в Юникод и вызывает W-версию той же функции. В поставляемых H-файлах библиотеки также определены имена функций без суффикса, и использование A- либо W-версии функций определяется опциями компиляции, а в модулях Delphi до 2010 версии, например, они жёстко завязаны на варианты с суффиксом A. При этом большинство новых функций, появившихся в Windows 2000 или более поздних ОС семейства Windows NT, существует только в Unicode-версии, потому что задача обеспечения совместимости со старыми программами и с ОС Windows 9x уже не стоит так остро, как раньше.

POSIX и OS/2. В отличие от большинства свободных Unix-подобных ОС, Windows NT сертифицирована институтом NIST на совместимость со стандартом POSIX.1, и даже с более строгим стандартом FIPS 151-2. Библиотекой *`psxdll`* экспортируются стандартные функции POSIX, а также некоторые функции Native API, не имеющие аналогов в POSIX – например, для работы с кучей, со структурными исключениями, с Юникодом. Внутри этих функций используются как Native API, так и LPC-вызовы в подсистему psxss, являющуюся обычным Win32-процессом.

Для выполнения 16-битных программ, написанных для OS/2 1.x, в состав Windows NT включены две системных библиотеки OS/2 (*`doscalls`* и *`netapi`)* и консольная программа-эмулятор *`os2`*, которая загружает и использует посредством LPC-вызовов подсистемы *`os2srv`* и *`os2ss`*. Остальные системные библиотеки OS/2, кроме двух названных (*`kbdcalls`, `mailslot`, `moncalls`, `nampipes`, `quecalls`, `viocalls`*), не хранятся как отдельные файлы, а эмулируются. Программы, написанные для OS/2 2.0 и выше, а также оконные программы и программы, напрямую работающие с устройствами компьютера, в том числе драйверы, системой Windows NT не поддерживаются.

Обе эти подсистемы, необязательные для работы большинства приложений, были удалены в Windows XP и последующих выпусках Windows. При помощи манипуляций с реестром их можно было отключить и в предыдущих версиях Windows NT, что рекомендовалось специалистами по компьютерной безопасности в целях сокращения поверхности атаки компьютерной системы [15].

DOS и Win16. Чтобы обеспечить двоичную совместимость с существующими программами для предыдущих семейств ОС от Microsoft, в Windows NT была добавлена программа-эмулятор *`ntvdm`,* реализующая VDM (виртуальную DOS-машину), внутри которой может выполняться программа для DOS. Для каждой выполняемой DOS-программы создаётся собственная VDM, тогда как несколько 16-битных Windows-программ могут выполняться в отдельных потоках внутри одной VDM, которая в этом случае играет роль подсистемы. Для того, чтобы внутри VDM можно было выполнять программы для Windows, в неё сначала должна быть загружена программа *`wowexec`,* устанавливающая связь VDM с платформой WOW («Windows on Win32»), позволяющей использовать 16-битные приложения для Windows наравне с 32-битными. Сама программа-эмулятор *`ntvdm`* выполняется внутри подсистемы Win32, что позволяет Win32-программам обращаться к окнам DOS-программ как к консольным окнам, а к окнам Win16-программ – как к графическим окнам.

Ещё одна технология обеспечения двоичной совместимости, реализованная в Windows NT – это thunks – небольшие секции кода, выполняющие преобразования (например типов) или обеспечивающие вызов 32-разрядного кода из 16-разрядного и наоборот. Thunks позволяют 32-битным программам пользоваться 16-битными DLL-библиотеками (для Windows или OS/2) и наоборот. Thunks для Win16 реализованы в библиотеках *`wow32`* (32-битные точки входа) и *`krnl386`* (16-битные точки входа); thunks для OS/2 – в библиотеке *`doscalls`* (16-битные точки входа). К 16-битным системным библиотекам, включённым в состав Windows NT для использования технологией WOW, относятся *`krnl386`, `gdi`, `user`, `shell`* и др. Поддержка DOS-программ виртуальной DOS-машиной системы Windows NT не ограничена эмуляцией реального режима процессора x86: поддерживается интерфейс DPMI, позволяющий DOS-программам обращаться к расширенной памяти. Однако поддержка программ для DOS и Win16 в Windows NT ограничена требованиями безопасности: программы, напрямую работающие с устройствами компьютера, не поддерживаются.

В связи с аппаратными ограничениями 64-битных платформ поддержка VDM и WOW была исключена из 64-битных версий Windows, запуск 16-битных программ средствами системы на них невозможен, но возможно использование эмуляторов, таких как DOSBox. Основным API этих версий Windows NT является 64-битная версия Win32 API; для запуска 32-битных программ используется технология WOW64, аналогичная традиционной WOW.

2.3 История, версии и достоинства

Windows – одна из ключевых и важнейших операционных систем в истории компьютерной индустрии, начиная с её появления в 1985 году. С этого времени Windows прошла значительный путь развития и сегодня является одной из наиболее распространенных и влиятельных операционных систем в мире. Временная шкала, показывающая выпуски Windows для персональных компьютеров и серверов, показана на рисунке 2.2 [16].

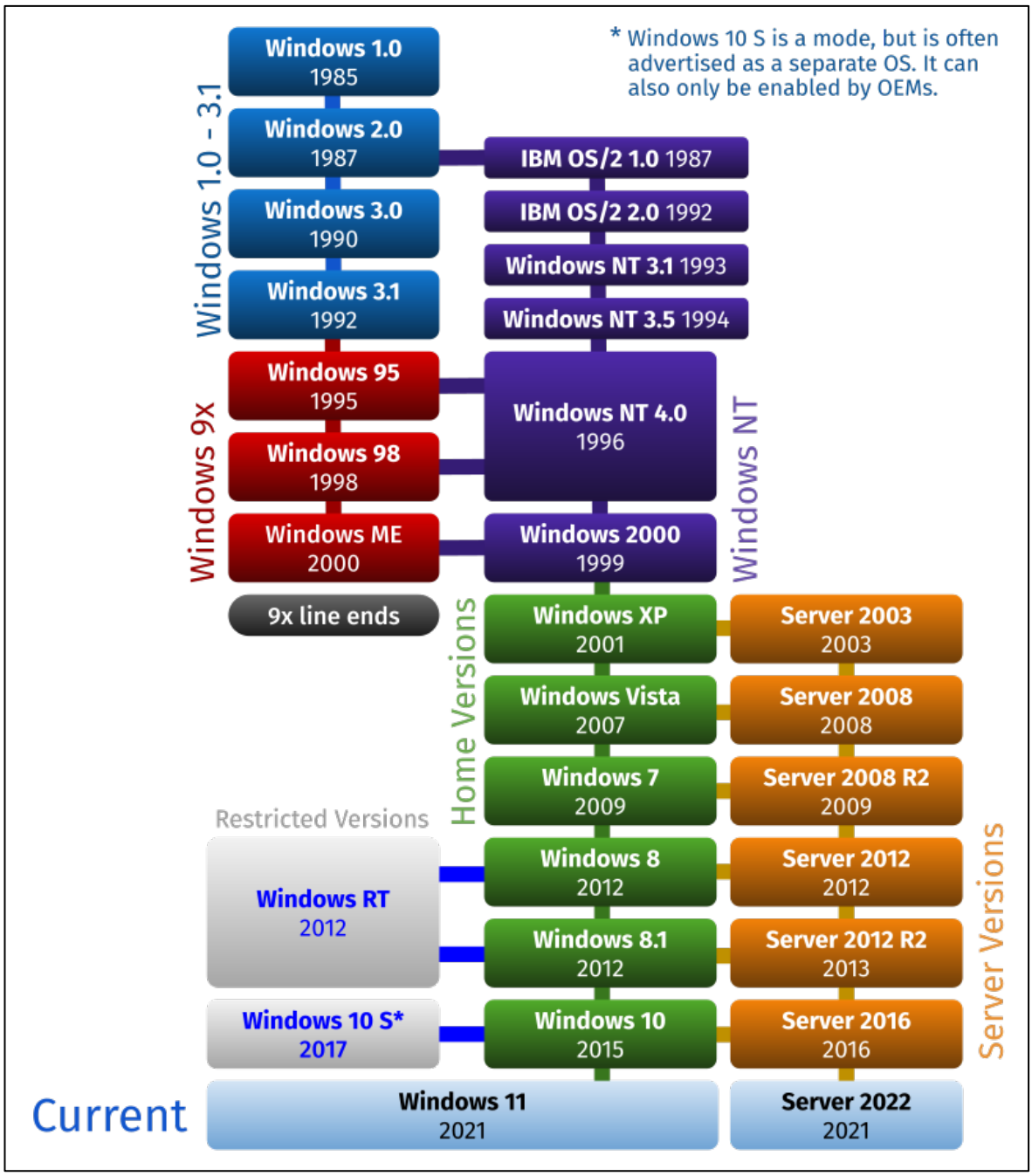

Рисунок 2.2 – История версий Windows

Windows 1 была первой версией операционной системы Windows. Она была выпущена в ноябре 1985 года и стала первой попыткой Microsoft создать графический интерфейс пользователя.

Это означает, что вместо ввода команд пользователи могут указывать и нажимать на значки и кнопки на экране.

Билл Гейтс, основатель Microsoft, руководил разработкой Windows 1. Она была разработана для работы поверх операционной системы командной строки MS-DOS.

Windows 1 в значительной степени полагалась на мышь, которая не была обычным компьютерным устройством ввода. Microsoft включила игру под названием Reversi, в которой требовалась мышь, чтобы помочь пользователям привыкнуть к новому способу взаимодействия с компьютером.

Некоторые ключевые особенности Windows 1:

– графический интерфейс пользователя (GUI) с раскрывающимися меню, мозаичными окнами и поддержкой мыши;

– независимая от устройства графика экрана и принтера;

– совместная многозадачность приложений Windows.

Windows 2 была выпущена Microsoft в декабре 1987 года, через два года после Windows 1. Самым большим улучшением в Windows 2 было то, что окна могли перекрывать друг друга, что упрощало одновременную работу с несколькими программами.

В Windows 2 также появилась возможность сворачивать и разворачивать окна, которая используется до сих пор.

Еще одним большим изменением в Windows 2 стало появление панели управления, которая облегчила поиск системных настроек и опций. Microsoft Word и Excel также были впервые выпущены для Windows 2.

Ключевые особенности Windows 2:

– перекрывающиеся окна;

– сворачивание и развертывание окон;

– панель управления настройками системы;

– Microsoft Word и Excel

Windows 3, выпущенная в 1990 году, стала первой версией Windows, ставшей популярной. Она была предустановлена на многих ПК-совместимых компьютерах и позволяла пользователям запускать несколько программ одновременно или в режиме «многозадачности».

Она также имела более современный и красочный вид и поддерживала 256 цветов. Среди его наиболее выдающихся особенностей была карточная игра «Пасьянс», которая имела большой успех среди пользователей.

Windows 3 имела большой успех: было продано более 10 миллионов копий и стала самым продаваемым графическим пользовательским интерфейсом за всю историю.

Windows 3.1 была выпущена в 1992 году и представляла собой обновление Windows 3.0. Это была первая версия Windows, в которой использовались шрифты TrueType.

В этой версии также был представлен Сапер. Windows 3.1 требовала 1 МБ ОЗУ и позволяла управлять программами MS-DOS с помощью мыши. Это также была первая версия Windows, распространявшаяся на компакт-диске. Всего за два месяца было продано три миллиона копий новой версии.

Основными изменениями по сравнению с Windows 3.0 были улучшенная поддержка мультимедиа и сети, поддержка шрифтов TrueType и улучшенная диагностика ошибок.

Windows 95 была выпущена в 1995 году и содержала несколько новых функций. Самым заметным изменением стала кнопка «Пуск» и меню «Пуск». В операционной системе также появилась концепция «Plug and Play», которая должна была автоматически находить и устанавливать драйверы для новых периферийных устройств, но иногда не работала. Windows 95 также была ориентирована на многозадачность и имела 32-битную среду.

Она был разработана для работы с Windows NT и имела большую обратную совместимость со старыми драйверами и программным обеспечением. В Windows 95 также появился Internet Explorer, но он не был установлен по умолчанию.

Windows 98 была выпущена в июне 1998 года как обновление Windows 95. В ней было представлено несколько новых функций и улучшений, таких как улучшенная поддержка USB-устройств и жестких дисков большего размера, а также улучшения пользовательского интерфейса.

Вот некоторые из новых функций и улучшений:

– улучшенная поддержка USB, упрощающая подключение и использование USB-устройств, таких как мыши и принтеры;

– возможность использовать жесткие диски и файлы большего размера благодаря новой файловой системе FAT32;

– поддержка нескольких мониторов;

– новые программы, такие как Internet Explorer 4, Outlook Express и проигрыватель Windows Media;

– улучшена производительность и сокращено время запуска программ;

– поддержка DVD-ROM и общего доступа к подключению к Интернету (ICS) во втором издании (Windows 98SE).

Windows ME: Операционная система, выпущенная в 2000 году, могла бы быть принята многими лучше, поскольку было известно, что у нее было много проблем. Это была последняя версия Windows, основанная на MS-DOS, и она предназначалась для домашнего использования.

В Windows ME представлены новые функции, такие как инструменты автоматического восстановления системы, IE 5.5, Windows Media Player 7 и Windows Movie Maker. Однако он также был известен своими ошибками и проблемами с установкой.

Windows Me была похожа на Windows 98SE и включала новые возможности для домашней сети, редактирования видео и восстановления системы в случае возникновения чрезвычайной ситуации.

Windows 2000 была выпущена в феврале 2000 года и была основана на бизнес-ориентированной системе Windows NT. Windows 2000 была первой системой, поддерживавшей режим гибернации, и автоматическое обновление Microsoft сыграло важную роль. Она построена на базе Windows NT и доступен в трех вариантах: Professional, Server и Advanced Server.

Windows 2000 предлагала множество улучшений, таких как более надежная интеграция с Интернетом, поддержка до 4 ГБ оперативной памяти и исключение многих сценариев перезагрузки. Она также имела защиту файлов Windows, которая не позволяла установленным приложениям удалять необходимые системные файлы.

Windows XP, выпущенная в 2001 году, объединила в себе лучшие возможности корпоративных и потребительских операционных систем Microsoft. В ее интерфейсе были обновлены меню «Пуск», панель задач, обои Vista и новые визуальные эффекты.

Также были представлены новые инструменты, такие как ClearType, встроенная функция записи компакт-дисков, а также работающие автоматические обновления и инструменты восстановления. Несмотря на то, что долгий срок службы сделал ее популярным, ее недостатки в безопасности, особенно в Internet Explorer, сделали ее уязвимым для взлома и кибератак.

В 2007 году Microsoft выпустила новую операционную систему Windows Vista, которая заменила Windows XP. Она имела свежий вид, с прозрачными элементами и упором на поиск и безопасность.

Однако у нее были некоторые проблемы. Пользователям приходилось одобрять каждое приложение, которое хотело внести изменения, что заставляло людей небрежно относиться к своей безопасности.

Она также работал медленно на старых компьютерах. В Windows Vista представлены такие функции, как Windows Aero, улучшенные инструменты

поиска и мультимедиа, а также улучшения безопасности. Однако это было не просто обновление XP, а совершенно новая система.

Основная поддержка Windows Vista закончилась 10 апреля 2012 г., а расширенная поддержка закончилась 11 апреля 2017 г. На смену ей пришла Windows 7, и по состоянию на февраль 2022 г. только 0,18% компьютеров работают под управлением Windows Vista.

Windows 7 была выпущена в 2009 году как преемница Windows Vista. Она был направлена на устранение проблем и жалоб пользователей, связанных с Vista, и считался огромным улучшением.

Windows 7 была быстрее, стабильнее и проще в использовании, и многие люди перешли на нее с Windows XP, полностью пропустив Vista. Некоторые новые функции Windows 7 включали распознавание рукописного ввода и возможность привязывать окна по бокам экрана для облегчения изменения размера.

Windows 7 – популярная операционная система, имеющая пять версий: Starter, Home Premium, Professional, Enterprise и Ultimate. Однако существуют также специальные версии Windows 7, называемые «N-редакции».

Эти N-версии Windows 7 дают вам свободу выбора собственного медиаплеера и другого программного обеспечения для управления и воспроизведения цифровых медиафайлов, таких как компакт-диски и DVD-диски.

14 января 2020 г. Microsoft прекратила выпуск обновлений безопасности и техническую поддержку для Windows 7 спустя 10 лет. Это означает, что Microsoft больше не поддерживает Windows 7.

Windows 8 была выпущена в 2012 году с совершенно новым интерфейсом, в котором кнопка «Пуск» и меню «Пуск» заменены более сенсорным стартовым экраном. В новом интерфейсе появились значки программ и живые плитки, отображающие информацию в виде «виджетов».

Интерфейс рабочего стола, аналогичный Windows 7, все еще был доступен. Windows 8 была быстрее своих предшественников и представила поддержку устройств USB 3.0.

Был представлен Магазин Windows, предлагающий универсальные приложения для Windows, но сторонние программы могли получить доступ только к традиционному интерфейсу рабочего стола. Однако новый сенсорный интерфейс не приветствовался традиционными пользователями настольных компьютеров, которые предпочитали использовать мышь и клавиатуру.

Windows 8.1 – бесплатное обновление, выпущенное в октябре 2013 года

и направленное на устранение некоторых критических замечаний в адрес ее предшественницы, Windows 8.

Она включал в себя повторное введение кнопки «Пуск», отсутствовавшей в Windows 8, а также возможность загрузки непосредственно в режиме рабочего стола, что лучше подходит для пользователей с мышью и клавиатурой.

С этим выпуском Microsoft также начала переходить к ежегодным обновлениям программного обеспечения. Также была представлена Windows RT, версия для процессоров на базе ARM.

Windows 10 – новейшая версия операционной системы Microsoft, анонсированная в 2014 году как «техническая предварительная версия», доступная для тестирования. Windows 10 предназначена для унификации всех платформ на нескольких устройствах, позволяя загружать универсальные приложения из Магазина Windows и запускать их на всех устройствах с Windows. Она также включает переключение между режимами клавиатуры, мыши и планшета для таких устройств, как Surface Pro 3.

Windows 10 должна была быть выпущена в 2015 году после конференции разработчиков Microsoft.

Windows 365 – это служба облачных вычислений от Microsoft, которая позволяет пользователям транслировать все возможности рабочего стола Windows на любое устройство, подключенное к Интернету. Она предоставляет виртуализированную среду с настраиваемыми параметрами хранения, вычислительной мощности и памяти для удовлетворения потребностей отдельных лиц или организаций.

Windows 11 – последняя версия операционной системы Microsoft, выпущенная в конце 2021 года. Она имеет некоторые новые функции, в том числе изменения во внешнем виде интерфейса, новую систему виджетов для быстрого доступа к информации и возможность привязки окна в разных макетах.

Приложения по умолчанию были обновлены, а обновленное приложение настроек упрощает изменение системных настроек. Пользователи также теперь могут иметь несколько рабочих столов, которые можно использовать для более эффективной организации своей работы.

Teams теперь также интегрирован в систему, что упрощает совместную работу с коллегами. Кроме того, появились новые специальные возможности, в том числе общесистемные субтитры и более естественные голоса диктора [17].

2.4 Обоснование выбора платформы

В процессе курсового проекта, решение о выборе программной платформы стоит на первом плане. В данном контексте, было решено использовать операционную систему Windows 10 в качестве базовой платформы для исследований и разработки программ. Это решение подкреплено несколькими весомыми аргументами:

1 Массовость использования. Windows – одна из самых популярных ОС в мире, обеспечивая огромное количество пользователей и организаций. Это делает исследование на этой платформе актуальным и востребованным, а результаты приобретают широкое практическое применение.

2 Богатые средства разработки. ОС Windows предлагает широкий набор инструментов для разработки, включая популярные IDE, такие как Visual Studio, а также поддержку различных языков программирования. Это значительно упрощает процесс создания и тестирования приложений.

3 Удобство настройки и использования. Windows обладает интуитивным интерфейсом и легко настраивается для разработки и тестирования параллельных приложений. Это позволяет разработчикам сосредоточиться на сути исследования и оптимизации вычислений, минуя сложности настройки среды разработки.

4 Экосистема приложений и сервисов. Операционная система Windows 10 предоставляет доступ к широкой экосистеме приложений и сервисов, что облегчает разработку и интеграцию программ. Разнообразие доступных приложений и инструментов позволяет выбирать оптимальные решения для конкретных задач и ускоряет процесс разработки.

Таким образом, выбор операционной системы Windows 10 в качестве базовой платформы для курсового проекта является обоснованным и целесообразным решением. Массовость использования этой операционной системы, богатые средства разработки, удобство настройки и экосистема приложений и сервисов делают ее привлекательным выбором для проведения исследований и разработки программного обеспечения. Платформа Windows 10 обеспечивает надежную основу для работы с вычислениями, позволяя разработчикам сосредоточиться на сути задачи и достижении поставленных целей.

3 ТЕОРЕТИЧЕСКОЕ ОБОСНОВАНИЕ РАЗРАБОТКИ ПРОГРАММНОГО ПРОДУКТА

3.1 Обоснование необходимости разработки

Разработка программного продукта, представленного в данной работе, обоснована несколькими ключевыми аспектами, которые представляют собой значимые потребности и требования в сфере информационных технологий. Ниже перечислены основные моменты, которые подтверждают необходимость разработки данного программного решения:

1 Повышение эффективности управления устройствами. С увеличением количества подключаемых устройств к компьютерам возникает необходимость в эффективных средствах управления этими устройствами. Разработка программного продукта в виде диспетчера устройств позволит существенно упростить процесс управления и контроля за подключенными устройствами.

2 Необходимость в мониторинге производительности. В контексте постоянного развития информационных технологий и растущих требований к производительности компьютерных систем становится важным иметь инструменты для мониторинга и анализа производительности. Разработка соответствующих функций в программном продукте позволит пользователям эффективно отслеживать и улучшать производительность своих систем.

3 Повышение удобства использования программного продукта – это один из ключевых аспектов разработки, который направлен на создание интерфейса, максимально удобного для пользователей. Это важно, поскольку хорошо спроектированный интерфейс делает программу более доступной и привлекательной для пользователей, увеличивая их удовлетворенность и эффективность использования.

На основе указанных обоснований явно видна необходимость в разработке программного продукта, соответствующего современным требованиям и решающего актуальные задачи в сфере информационных технологий, является крайне важной и обоснованной. Учитывая рост числа подключаемых устройств, увеличение требований к производительности компьютерных систем и необходимость в удобстве использования программных продуктов, создание инструмента, объединяющего эти аспекты, становится необходимостью.

3.2 Технологии программирования, используемые для решения поставленных задач

Развитие программного продукта, который включает в себя диспетчер устройств и систему мониторинга производительности компьютерной системы, требует тщательного выбора технологий. В данном разделе будет проанализирован и обоснован выбор набора инструментов, нацеленных на эффективное достижение целей проекта. Внимание сосредоточено на том, каким образом использование определенных технологий позволяет эффективно решать поставленные задачи и обеспечивать высокое качество разработки программного продукта.

В рамках разработки программного продукта для диспетчера устройств и системы мониторинга производительности под операционную систему Windows было принято решение использовать платформу .NET и язык программирования C#. Данное решение обосновано рядом преимуществ, структурой и функциональными возможностями данных технологий:

1 Кроссплатформенность и переносимость кода. Платформа .NET обеспечивает кроссплатформенность, что означает возможность запуска приложений на различных операционных системах, включая Windows, Linux и macOS. Это обеспечивает гибкость в развертывании и использовании программного продукта на различных платформах.

2 Широкий набор инструментов и библиотек. Платформа .NET предоставляет разработчикам обширный набор инструментов и библиотек для разработки приложений. Это включает в себя интегрированную среду разработки Visual Studio, которая облегчает процесс создания, отладки и тестирования приложений. Богатая экосистема .NET также включает в себя множество сторонних библиотек и фреймворков, расширяющих функциональные возможности приложений.

3 Простота и удобство языка C#. Язык программирования C# отличается простым и интуитивно понятным синтаксисом, что упрощает процесс написания кода и повышает производительность разработчика. Благодаря своей структуре, напоминающей язык программирования Java, C# стал популярным выбором для создания приложений под платформу .NET.

.NET Framework – это платформа разработки программного обеспечения, представленная Microsoft в конце 1990 года под названием NGWS. 13 февраля 2002 года Microsoft выпустила первую версию .NET Framework, известную как .NET Framework 1.0 [18].

.NET Framework (рисунок 3.1) – это виртуальная машина, предоставляющая общую платформу для запуска приложения, созданного с использованием разных языков, таких как C#, VB.NET, Visual Basic и т. д. Она также используется для создания приложений на основе форм, консолей, мобильных и веб-приложение или службы, доступные в среде Microsoft. Более того, платформа .NET является чисто объектно-ориентированной, подобно языку Java. Но это не независимая платформа, как Java. Таким образом, его приложение работает только на платформе Windows.

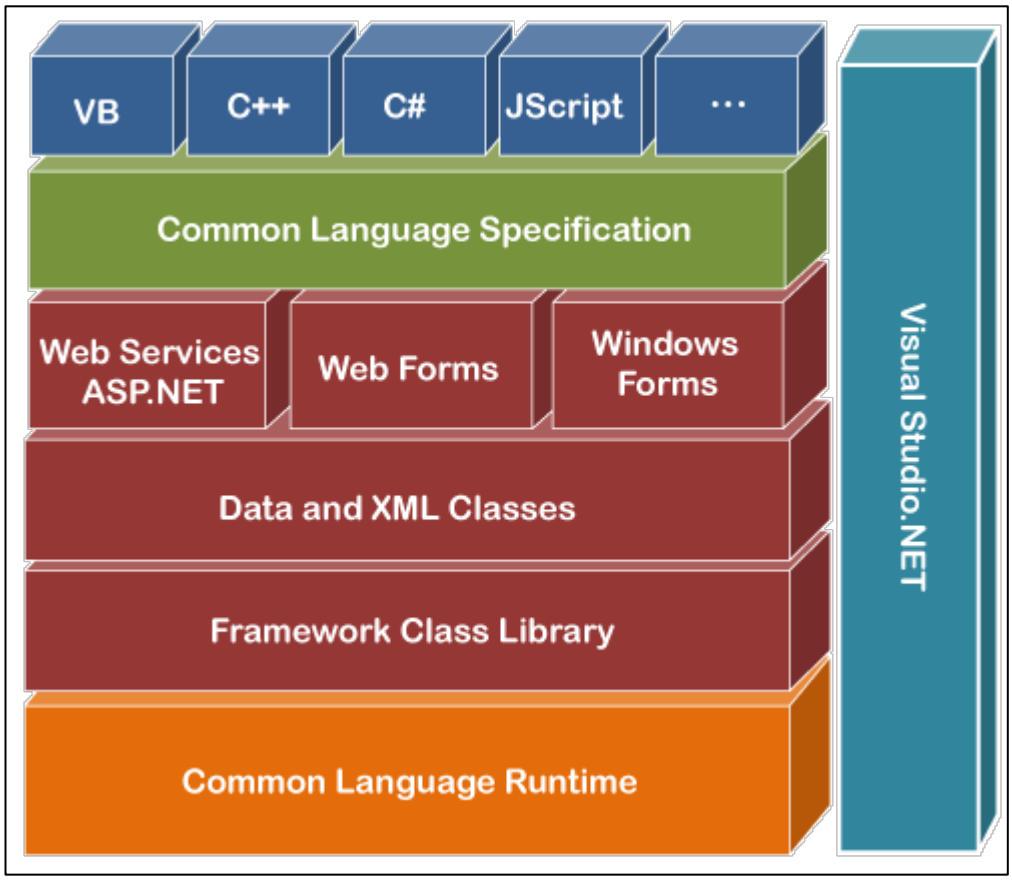

Рисунок 3.1 – Архитектура .NET Framework

CLR (общеязыковая среда выполнения) – это важная часть платформы .NET, которая работает как виртуальный компонент .NET Framework и выполняет программы на разных языках, такие как C# , Visual Basic и т. д. CLR также помогает конвертировать исходный код в байт-код и этот байт-код известен как CIL (Common Intermediate Language) или MSIL (Microsoft Intermediate Language). После преобразования в байт-код CLR во время выполнения использует JIT-компилятор, который помогает преобразовать код CIL или MSIL в машинный или собственный код.

CTS (система общих типов) – определяет стандарт, который представляет, какие типы данных и значений могут быть определены и управляться в памяти компьютера во время выполнения. CTS гарантирует, что программные данные, определенные на разных языках, будут взаимодействовать друг с другом для обмена информацией.

BCL – библиотека базовых классов, имеет богатую коллекцию возможностей и функций библиотек, которые помогают реализовать многие языки программирования в .NET Framework, такие как C#, F# , Visual C++ и другие.

CLS (общеязыковая спецификация) – это подмножество системы общих типов (CTS), определяющее набор правил и положений, которым должен следовать каждый язык, входящий в состав .NET Framework. Другими словами, язык CLS должен обеспечивать межъязыковую интеграцию или совместимость. Например, в языках C# и VB.NET язык C# завершает каждый оператор точкой с запятой, тогда как в VB.NET он не заканчивается точкой с запятой, и когда эти операторы выполняются в .NET Framework, он обеспечивает общую платформу для взаимодействия и обмена информацией друг с другом.

Сборка .NET – это основной строительный блок .NET Framework. Это небольшая единица кода, содержащая логически скомпилированный код в инфраструктуре Common Language (CLI), который используется для развертывания, безопасности и управления версиями. Он состоит из двух частей (процесса) DLL и сборок библиотеки (exe). Когда программа .NET компилируется, она генерирует метаданные с помощью Microsoft Intermediate Language, которые сохраняются в файле с именем Assembly.

C# (C-Sharp) – это мощный и гибкий язык программирования, разработанный компанией Microsoft специально для платформы .NET. Он предоставляет разработчикам широкие возможности для создания разнообразных приложений, от настольных программ до веб-приложений и игр. Вот некоторые ключевые аспекты языка C#:

1 Простота и эффективность. C# был разработан с упором на удобство использования и читаемость кода. Он обладает простым и понятным синтаксисом, что делает его доступным для новичков, но в то же время предоставляет продвинутые возможности для опытных разработчиков.

2 Статическая типизация. C# является языком со статической типизацией, что означает, что типы данных определяются на этапе компиляции. Это повышает безопасность кода и позволяет обнаруживать ошибки на ранних этапах разработки.

3 Обширный набор функциональных возможностей. C# предоставляет разработчикам широкий набор инструментов и функций для создания высокопроизводительных и масштабируемых приложений. Это включает в себя поддержку объектно-ориентированного программирования (ООП), асинхронного программирования, LINQ (Language Integrated Query) для работы с данными, обработку исключений и многое другое.

4 Интеграция с платформой .NET. C# плотно интегрирован с платформой .NET, что обеспечивает высокую производительность и эффективность при разработке приложений для различных платформ. Он может взаимодействовать с другими языками .NET, такими как Visual Basic.NET и F#, а также использовать библиотеки классов .NET (BCL) для выполнения различных задач.

5 Широкая поддержка сообщества и инструментов. C# имеет активное сообщество разработчиков, которое предоставляет множество ресурсов, библиотек и инструментов для упрощения разработки. Кроме того, среда разработки Visual Studio обеспечивает широкий набор инструментов для создания, отладки и развертывания приложений на C#.

В целом, C# – это современный, мощный и универсальный язык программирования, который позволяет разработчикам создавать высококачественные приложения для различных платформ и устройств.

Для разработки графического интерфейса была выбрана технология WinForms (Windows Forms). Windows Forms – интерфейс программирования приложений (API), отвечающий за графический интерфейс пользователя и являющийся частью Microsoft .NET Framework. Данный интерфейс упрощает доступ к элементам интерфейса Microsoft Windows за счет создания обёртки для существующего Win32 API в управляемом коде. Причём управляемый код – классы, реализующие API для Windows Forms, не зависят от языка разработки. То есть программист одинаково может использовать Windows Forms как при написании ПО на C#, C++, так и на VB.Net, J# и др.

Вот некоторые ключевые аспекты WinForms [19]:

1 Простота создания интерфейса. WinForms позволяет разработчикам создавать графический интерфейс пользователя с помощью перетаскивания и размещения элементов управления, таких как кнопки, текстовые поля, списки и т. д. Это делает процесс разработки графического интерфейса быстрым и интуитивно понятным.

2 Гибкость и настраиваемость. В WinForms предоставляется широкий набор стандартных элементов управления, а также возможность создания собственных пользовательских элементов. Разработчики могут легко

настраивать внешний вид и поведение элементов интерфейса, чтобы соответствовать требованиям проекта.

3 Событийно-ориентированная модель программирования. WinForms основан на событийно-ориентированной модели программирования, что означает, что приложение реагирует на события, такие как нажатие кнопки или изменение значения поля ввода. Это делает возможным создание интерактивных и отзывчивых приложений.

4 Интеграция с другими технологиями .NET. WinForms плотно интегрирован с другими технологиями и инструментами .NET, такими как язык программирования C# и среда разработки Visual Studio. Это обеспечивает совместимость с широким спектром инструментов и ресурсов для разработки приложений.

5 Поддержка многоязычности и культур. WinForms предоставляет механизмы для локализации и многоязычной поддержки приложений, что позволяет создавать приложения для различных региональных и языковых рынков.

6 Поддержка различных платформ. Приложения, созданные с использованием WinForms, могут быть запущены на различных платформах, поддерживаемых платформой .NET, включая Windows, Linux и macOS.

Таким образом, WinForms представляет собой надежную и эффективную технологию для создания графического интерфейса пользователя в приложениях на платформе .NET. Ее простота использования, гибкость настройки и широкие возможности интеграции делают ее привлекательным выбором для разработчиков, стремящихся создать интуитивно понятные и функциональные пользовательские интерфейсы.

Для взаимодействия с системными ресурсами и процессами требуются специализированные средства и технологии.

Windows Management Instrumentation (WMI) в дословном переводе – инструментарий управления Windows. WMI – это одна из базовых технологий для централизованного управления и слежения за работой различных частей компьютерной инфраструктуры под управлением платформы Windows.

Технология WMI – это расширенная и адаптированная под Windows реализация стандарта WBEM (Web-Based Enterprise Management), принятого многими компаниями, в основе которого лежит идея создания универсального интерфейса мониторинга и управления различными системами и компонентами распределённой информационной среды предприятия с использованием объектно-ориентированных идеологий и протоколов HTML и XML.

В основе структуры данных в WBEM лежит Common Information Model (CIM), реализующая объектно-ориентированный подход к представлению компонентов системы. CIM является расширяемой моделью, что позволяет программам, системам и драйверам добавлять в неё свои классы, объекты, методы и свойства.

WMI, основанный на CIM, также является открытой унифицированной системой интерфейсов доступа к любым параметрам операционной системы, устройствам и приложениям, которые функционируют в ней.

Важной особенностью WMI является то, что хранящиеся в нём объекты соответствуют динамическим ресурсам, то есть параметры этих ресурсов постоянно меняются, поэтому параметры таких объектов не хранятся постоянно, а создаются по запросу потребителя данных. Хранилище свойств объектов WMI называется репозиторием и расположено в системной папке ОС Windows: `*%SystemRoot%\System32\WBEM\Repository*`.

Так как WMI построен по объектно-ориентированному принципу, то все данные операционной системы представлены в виде объектов и их свойств и методов.

Все классы группируются в пространства имён, которые иерархически упорядочены и логически связаны друг с другом по определённой технологии или области управления. В WMI имеется одно корневое пространство имён Root, которое в свою очередь имеет 4 подпространства: CIMv2, Default, Security и WMI.

Классы имеют свойства и методы и находятся в иерархической зависимости друг от друга, то есть классы-потомки могут наследовать или переопределять свойства классов-родителей, а также добавлять свои свойства.

Свойства классов используются для однозначной идентификации экземпляра класса и для описания состояния используемого ресурса. Обычно все свойства классов доступны только для чтения, хотя некоторые из них можно модифицировать определённым методом. Методы классов позволяют выполнить действия над управляемым ресурсом.

К каждому экземпляру класса можно обратиться по полному пути, который имеет следующую структуру: `*[\\ComputerName\NameSpace][:ClassName][.KeyProperty1=Value1…]*`, где ComputerName – имя компьютера, NameSpace – название пространства имен, ClassName – имя класса, KeyProperty1=Value1 – свойства объекта и значение по которому он идентифицируется.

Пример обращения к процессу с именем «Calc.exe», который запущен на локальной машине: `*\\.\CIMv2:Win32_Process.Name="Calc.exe"*`.

Для доступа к дополнительным функциям и ресурсам операционной системы могут использоваться следующие библиотеки и DLL:

1 *`kernel32.dll`* – эта библиотека содержит множество функций, необходимых для взаимодействия с операционной системой Windows. Например, для получения списка подключенных устройств можно использовать эту библиотеку в сочетании с функцией *`SetupDiEnumDeviceInfo`* из `SetupAPI.dll`. Эта функция позволяет перечислить все устройства, подключенные к компьютеру, и получить информацию о каждом из них.

2 *`user32.dll`* – содержит функции для создания и управления окнами пользовательского интерфейса. С ее помощью можно создавать, изменять и уничтожать окна, а также обрабатывать сообщения, связанные с вводом и выводом (например, MessageBox для вывода диалоговых окон).

3 *`gdi32.dll`* – предоставляет функции для работы с графическими объектами на экране. Она содержит методы для рисования линий, фигур, текста, а также для управления цветами и шрифтами. Например, функция CreatePen используется для создания пера, а TextOut для вывода текста на экран.

4 *`advapi32.dll`* – эта библиотека предоставляет функции для работы с различными системными службами и компонентами безопасности. С ее помощью можно управлять службами Windows, работать с реестром, аутентифицировать пользователей и многое другое. Например, функция RegOpenKeyEx используется для открытия ключа в реестре.

5 *`shell32.dll`* – содержит функции для работы с оболочкой Windows, включая доступ к файловой системе, ярлыкам и контекстным меню. Она позволяет выполнять различные операции с файлами и папками, создавать ярлыки, а также работать с контекстными меню в проводнике Windows.

Таким образом, использование .NET Framework, C#, Windows Forms, WMI и системных DLL для разработки диспетчера устройств и мониторинга производительности представляет собой оптимальный выбор. Эти технологии обеспечивают удобное и эффективное создание приложений под операционную систему Windows, обладают широкими возможностями для работы с системными ресурсами и предоставляют инструменты для создания удобного и интуитивно понятного пользовательского интерфейса. Благодаря этому сочетанию технологий возможна реализация функционала диспетчера устройств и мониторинга производительности, что позволяет эффективно управлять подключенными устройствами и отслеживать работу компьютерных систем [20].

4 ПРОЕКТИРОВАНИЕ ФУНКЦИОНАЛЬНЫХ ВОЗМОЖНОСТЕЙ ПРОГРАММЫ

4.1 Анализ требований

Программное решение, предоставляющее пользователю доступ к просмотру подключенных устройств и анализу производительности компьютерных компонентов, должно представлять собой комплексный инструмент с следующими ключевыми возможностями:

1 Конфигурация системы. Добавление в программу вкладки "Компьютер" позволит пользователю просматривать основные характеристики своей компьютерной системы, такие как установленная операционная система, архитектура процессора, объем оперативной памяти и другие важные параметры. Это дает пользователям полное представление о конфигурации своей системы, что может быть полезно при управлении ресурсами и решении проблем с производительностью.

2 Диспетчер устройств. Программа должна обеспечить возможность просмотра всего списка подключенных устройств с подробной информацией о каждом из них, включая их характеристики и статус. Пользователю следует предоставить функционал для управления подключенными устройствами, позволяя осуществлять операции по их управлению и настройке. Возможна интеграция с операционной системой Windows, благодаря чему пользователь сможет легко управлять подключенными устройствами и выполнять операции по их настройке и управлению, такими как установка драйверов или изменение параметров подключения.

3 Мониторинг производительности. Программа должна предоставить возможность наблюдения за использованием центрального процессора, оперативной памяти, дискового пространства и сетевого трафика в реальном времени. Для удобства восприятия информации пользователем, данные о производительности следует представить в форме графиков или диаграмм.

Для реализации описанных функциональных возможностей необходимо создать программное решение, которое будет представлять собой интуитивно понятный и удобный в использовании инструмент. Таким образом, успешная реализация всех функциональных возможностей программы позволит пользователям получить полное представление о конфигурации своей компьютерной системы, эффективно управлять подключенными устройствами и мониторить производительность компонентов, что сделает использование программы удобным и эффективным.

4.2 Конфигурация системы

Для отображения информации на вкладке «Computer» был разработан класс TaskManagerSystemProps, который отвечает за получение информации о конфигурации компьютерной системы. Фрагмент кода данного класса представлен на рисунке 4.1.

Рисунок 4.1 – Класс TaskManagerSystemProps

Данный класс включает в себя набор свойств и методов, которые позволяют получить различные параметры операционной системы, процессора, BIOS и других компонентов.

Метод Refresh класса TaskManagerSystemProps играет ключевую роль в обновлении информации о конфигурации компьютерной системы. Он использует различные источники данных, такие как реестр Windows и системные API, чтобы получить подробные сведения о системе. В своей работе метод обращается к различным ключам и значениям реестра, чтобы извлечь информацию о версии операционной системы, характеристиках процессора, а также других системных параметрах. Например, для получения

информации о версии операционной системы, метод использует ключи реестра, такие как SOFTWARE\Microsoft\Windows NT\CurrentVersion, где содержится информация о названии, версии и других атрибутах Windows.

Для получения сведений о процессоре метод обращается к ключам в реестре, таким как HARDWARE\DESCRIPTION\System\CentralProcessor\0, где содержится информация о производителе, модели и других характеристиках процессора. Кроме того, метод использует системные API, такие как GetPhysicallyInstalledSystemMemory, чтобы определить общий объем установленной оперативной памяти.

Метод DecodeProductKey в классе TaskManagerSystemProps предназначен для расшифровки кода продукта Windows, который хранится в реестре системы в зашифрованном виде. Этот код представляет собой уникальный идентификатор операционной системы, который можно использовать для активации Windows.

Процесс расшифровки начинается с извлечения зашифрованного кода продукта из реестра Windows. Затем этот код разбивается на отдельные байты, и каждый байт интерпретируется как индекс для определения символа в таблице возможных символов. Таблица содержит символы, которые могут быть использованы в коде продукта Windows, включая буквы и цифры, за исключением некоторых символов, таких как гласные и некоторые согласные, которые не используются в кодах продуктов.

Используя эти индексы, метод производит перевод каждого байта в соответствующий символ из таблицы возможных символов. Последовательность символов образует расшифрованный код продукта Windows, который затем возвращается в виде строки.

Собранная информация сохраняется в различных свойствах класса TaskManagerSystemProps, таких как WindowsName, ProcessorVendor, TotalMemory и другие. Эти свойства позволяют пользователю получить доступ к различным аспектам конфигурации системы и использовать их в приложении для отображения информации о системе.

Таким образом, благодаря методу Refresh и свойствам класса TaskManagerSystemProps пользователь получает возможность получить подробные сведения о конфигурации своей компьютерной системы, а метод Refresh обеспечивает актуальность и точность этой информации, обновляя ее в соответствии с текущим состоянием системы.

4.3 Диспетчер устройств

Для вкладки «Devices» был разработан класс TaskManagerDeviceClass. Этот класс отвечает за загрузку и отображение информации об устройствах, подключенных к компьютеру, а также их характеристик. Фрагмент кода данного класса представлен на рисунке 4.2.

```
[SupportedOSPlatform("windows")]
11 references
internal class TaskManagerDeviceClass : IComparable, IComparable<TaskManagerDeviceClass> {
    private static readonly DEVPROPKEY DEVPKEY_Device_DeviceDesc = new("a45c254e-df1c-4efd-8020-67d146a850e0", 2);
    private static readonly DEVPROPKEY DEVPKEY_Device_Service = new("a45c254e-df1c-4efd-8020-67d146a850e0", 6);
    private static readonly DEVPROPKEY DEVPKEY_Device_ClassGuid = new("a45c254e-df1c-4efd-8020-67d146a850e0", 10);
    private static readonly DEVPROPKEY DEVPKEY_Device_FriendlyName = new("a45c254e-df1c-4efd-8020-67d146a850e0", 14);
    private static readonly DEVPROPKEY DEVPKEY_Device_InstallState = new("a45c254e-df1c-4efd-8020-67d146a850e0", 36);
    private static readonly DEVPROPKEY DEVPKEY_DeviceClass_IconPath = new("259abffc-50a7-47ce-af08-68c9a7d73366", 12);
    private static readonly DEVPROPKEY DEVPKEY_Device_ProblemCode = new("4340a6c5-93fa-4706-972c-7b648008a5a7", 3);
    private static readonly DEVPROPKEY DEVPKEY_Device_InstanceId = new("78c34fc8-104a-4aca-9ea4-524d52996e57", 256);
    private readonly List<TaskManagerDevice> _devices = new();

    1 reference
    public TaskManagerDeviceClass(Guid classId, string description, string iconPath) {
        ClassId = classId;
        Description = description;
        IconPath = iconPath;
    }

    3 references
    public Guid ClassId { get; }
    4 references
    public string Description { get; }
    2 references
    public string IconPath { get; private set; }
    1 reference
    public IReadOnlyList<TaskManagerDevice> Devices => _devices;
```

Рисунок 4.2 – Класс TaskManagerDeviceClass

Класс TaskManagerDeviceClass представляет собой важный компонент для работы вкладки "Devices" в диспетчере задач. Ниже приведено подробное описание его функционала.

Для определения и доступа к различным свойствам устройств, таким как описание устройства, служба, класс GUID и т. д. в классе TaskManagerDeviceClass используются свойства DEVPROPKEY. Каждая константа DEVPROPKEY, определенная в классе, представляет одно из таких свойств и идентифицирует его по GUID и идентификатору свойства. Это позволяет классу TaskManagerDeviceClass правильно интерпретировать и использовать информацию об устройствах при загрузке и отображении.

Структура DEVPROPKEY представляет пару значений, определяющих свойство устройства. Она содержит два поля: fmtid и pid. FormID представляет собой GUID формата, к которому относится свойство. Формат определяет, как интерпретировать значение свойства. Property ID – это идентификатор свойства в пределах данного формата. Он указывает на конкретное свойство внутри формата.

Класс инициализируется с GUID класса устройства, его описанием и путем к иконке. Эти параметры позволяют категоризировать устройства для их последующего отображения. Свойства ClassId, Description и IconPath хранят соответственно GUID класса, его описание и путь к иконке.

Метод выполняет загрузку информации о подключенных устройствах и их категориях. Метод принимает объект фильтра TaskManagerDeviceFilter и возвращает список классов устройств. Для этого метод использует функции API Windows, такие как SetupDiGetClassDevs, SetupDiEnumDeviceInfo, чтобы получить информацию о подключенных устройствах и их характеристиках.

Класс содержит ряд вспомогательных методов, таких как GetStringProperty, GetIntProperty и GetGuidProperty, которые используются для извлечения строковых, числовых и GUID-свойств устройств из API Windows.

Каждый экземпляр класса TaskManagerDeviceClass содержит список экземпляров класса TaskManagerDevice, представляющих отдельные устройства в данной категории. Каждое устройство хранит информацию о его имени, состоянии подключения, идентификаторе экземпляра и других свойствах. Класс TaskManagerDevice представлен на рисунке 4.3.

```
[SupportedOSPlatform("windows")]
16 references
internal class TaskManagerDevice : IComparable, IComparable<TaskManagerDevice> {
    1 reference
    public TaskManagerDevice(TaskManagerDeviceClass cls, string name, SP_DEVINFO_DATA data, bool connected) {
        Class = cls;
        Name = name;
        Data = data;
        ImageKey = data.ClassGuid + "-" + data.DevInst;
        Present = connected;
    }

    9 references
    public string Name { get; }
    1 reference
    public TaskManagerDeviceClass Class { get; }
    1 reference
    public SP_DEVINFO_DATA Data { get; }
    5 references
    public bool Present { get; }
    3 references
    public bool Disabled { get => ProblemCode == 22; }
    8 references
    public string InstanceID { get; set; } = "";
    1 reference
    public string Service { get; set; } = "";
    4 references
    public int ProblemCode { get; set; } = 0;
    12 references
    public string ImageKey { get; }
    0 references
    public override string ToString() => Name;

    0 references
    int IComparable.CompareTo(object? obj) => CompareTo(obj as TaskManagerDevice);
    1 reference
    public int CompareTo(TaskManagerDevice? other) => Name.CompareTo(other?.Name);

}
```

Рисунок 4.3 – Класс TaskManagerDevice

Для управления устройствами в выпадающем меню используются следующие функции:

1 Feature_EnableDisable. Эта функция позволяет включать или отключать выбранное устройство. Она сначала проверяет выбранное устройство и выводит предупреждение, если пользователь пытается отключить устройство. Затем она использует функции API Windows, такие как CM_Locate_DevNode, CM_Enable_DevNode и CM_Disable_DevNode, чтобы найти и изменить состояние устройства. В случае успеха она обновляет список устройств и отображает соответствующее сообщение о статусе операции.

2 Feature_Uninstall. Эта функция предназначена для удаления выбранного устройства из системы. Она также проверяет выбранное устройство и выводит предупреждение, чтобы пользователь мог подтвердить свое намерение удалить устройство. Затем она использует функции API Windows, такие как CM_Locate_DevNode и CM_Uninstall_DevNode, чтобы найти и удалить устройство. После успешного удаления она также обновляет список устройств и выводит сообщение о статусе операции.

3 Feature_Properties. Эта функция открывает свойства выбранного устройства. Она сначала проверяет выбранное устройство и его идентификатор экземпляра, а затем использует Process.Start, чтобы запустить команду rundll32.exe, которая открывает окно свойств устройства в диспетчере устройств Windows.

4.4 Мониторинг производительности

Для мониторинга производительности системы используется класс TaskManagerSystem, который предоставляет различные метрики и статистику о ресурсах компьютера. Изображение класса представлено на рисунке 4.4.

Этот класс включает в себя множество методов и свойств для обновления и отображения информации о производительности системы. Метод Refresh играет центральную роль в обновлении информации о производительности системы. Его основные шаги включают:

1 Вычисление времени работы системы. Метод определяет время работы системы, которое затем используется для вычисления некоторых метрик, таких как время работы ЦП.

2 Получение информации о производительности. Используя различные функции API Windows, такие как NtQuerySystemInformation и GetPerformanceInfo, метод собирает данные о различных аспектах производительности, таких как использование ЦП, памяти, дискового

пространства и сетевого трафика.

3 Вызов событий обновления. После успешного обновления информации метод вызывает события RefreshStarting и RefreshCompleted, чтобы уведомить другие части системы о начале и завершении процесса обновления. Это позволяет другим компонентам реагировать на изменения в производительности системы.

Класс TaskManagerSystem содержит ряд свойств и методов для работы с метриками производительности. Эти метрики предоставляют информацию о различных аспектах производительности системы, таких как использование памяти, диска и сети.

```
[SupportedOSPlatform("windows")]
3 references
internal class TaskManagerSystem : TaskManagerValuesBase {
    private API.SYSTEM_PERFORMANCE_INFORMATION _SPI = new();
    private API.PERFORMANCE_INFORMATION _PI = new();
    private readonly uint Multiplier = 4096U;
    private ulong _sumPageFileTotal, _sumPageFileUsed, _sumPageFilePeak;
    private readonly CpuUsage _Cpu = new();
    private TimeSpan _UpTime;

    1 reference
    public TaskManagerSystem() {
        _PI.cb = (uint)Marshal.SizeOf(_PI);
    }

    public event EventHandler? RefreshStarting;
    public event EventHandler? RefreshCompleted;

    1 reference
    public void Refresh(bool cancellingEvents = false) {
        CancellingEvents = cancellingEvents;
        if (LastUpdate == 0) { LastUpdate = DateTime.Now.Ticks - 10; }
        if (!CancellingEvents) RefreshStarting?.Invoke(this, new());

        // Compute System Uptime
        if (Environment.TickCount > 0) {
            _UpTime = TimeSpan.FromTicks(Environment.TickCount * 10000L);
        } else {
```

Рисунок 4.4 – Класс TaskManagerSystem

Класс TaskManagerSystem определяет обработчики событий для отслеживания изменений в значениях метрик. Эти обработчики позволяют реагировать на изменения производительности системы и обеспечивают возможность уведомления других частей приложения о таких изменениях.

В целом, класс TaskManagerSystem обеспечивает полный и надежный механизм для мониторинга производительности системы и предоставляет разработчикам возможность эффективно отслеживать и управлять ресурсами компьютера.

Для хранения и отображения значений метрик используется класс Metric (рисунок 4.5). Он представляет отдельную метрику и предоставляет методы для установки и инкрементации ее значения, а также форматирования значения для отображения.

Этот класс включает в себя следующие основные элементы:

1 Свойства и методы для работы с метриками. Класс содержит свойства для хранения текущего значения метрики и ее изменения, а также методы для установки и инкрементации значений. Он также предоставляет возможность форматирования значения метрики для отображения в пользовательском интерфейсе.

2 Обработчики событий. Класс определяет обработчики событий для отслеживания изменений значений метрик. Эти обработчики вызываются при изменении значений метрик и позволяют уведомлять другие части системы о таких изменениях.

Рисунок 4.5 – Класс Metric

Таким образом, классы TaskManagerSystem и Metric вместе обеспечивают эффективный мониторинг производительности системы, предоставляя полезную информацию о ресурсах и их использовании. TaskManagerSystem служит в качестве центрального управляющего элемента, который координирует процесс обновления информации о производительности системы и предоставляет интерфейс для получения этих данных.

С другой стороны, класс Metric отвечает за представление и форматирование отдельных метрик производительности. Он обеспечивает удобный интерфейс для работы с отдельными значениями и их изменениями, а также позволяет определить формат отображения этих данных.

5 ВЗАИМОДЕЙСТВИЕ С ПРОГРАММНЫМ ПРОДУКТОМ

5.1 Общая характеристика

Программный продукт представляет собой интегрированное решение, объединяющее функциональность диспетчера устройств и инструменты мониторинга производительности системы. При запуске программы пользователю доступна вкладка «Computer», которая служит входной точкой для получения основной информации о компьютере. Изображение данной вкладки представлено на рисунке 5.1.

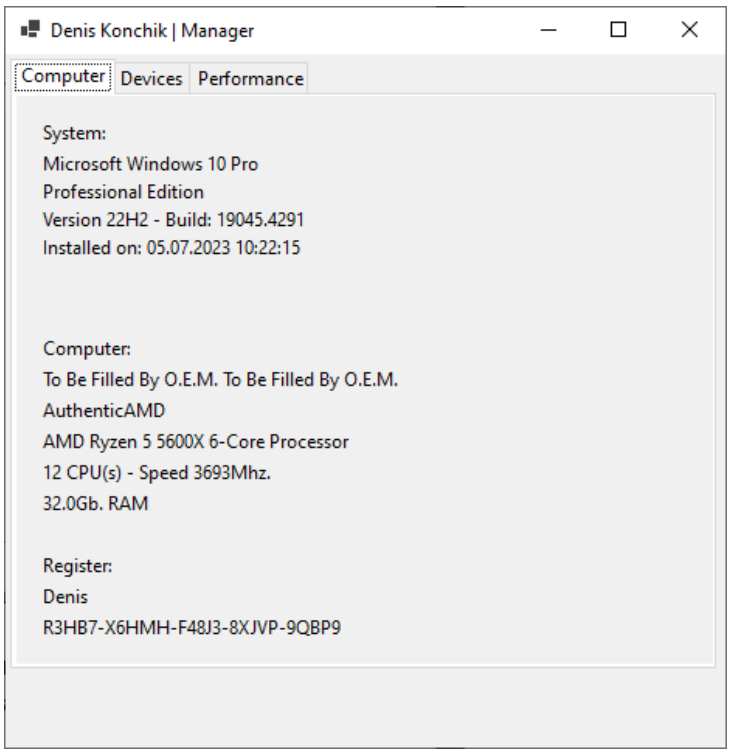

Рисунок 5.1 – Вкладка «Computer»

Данная вкладка предоставляет информацию об установленной операционной системе, имени пользователя, ключе активации и конфигурации компьютера.

Помимо этого, в программе имеются вкладки «Devices» и «Performance», которые предоставляют информацию о подключенных устройствах и производительности подсистем компьютера соответственно. Таким образом, программы обеспечивает пользователям аналогичный функционал, который они могут найти в стандартных инструментах операционной системы Windows.

5.2 Менеджер устройств

На вкладке «Devices» (рисунок 5.2) пользователи могут увидеть иерархическое представление устройств системы в виде дерева, где каждое устройство представлено в своей категории. Это предоставляет удобный способ ознакомиться с различными устройствами, подключенными к компьютеру, а также осуществлять управление ими и настраивать их параметры.

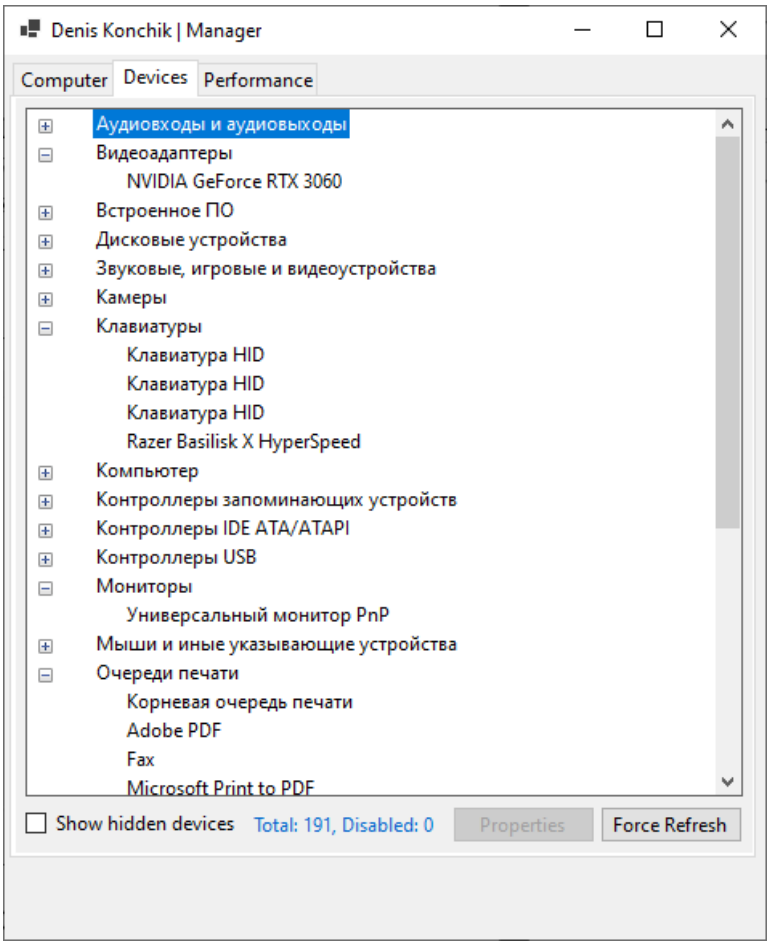

Рисунок 5.2 – Вкладка «Devices»

Чекбокс «Show hidden devices» позволяет пользователю отобразить скрытые устройства в списке на вкладке «Devices». Это полезная функция, которая позволяет пользователю просматривать и управлять устройствами, которые обычно не отображаются в списке из-за настроек системы.

После нажатия на эту кнопку в списке устройств появятся скрытые устройства, что позволит пользователю просматривать их свойства, выполнять действия по управлению и, при необходимости, изменять их настройки.

Кроме того, под списком устройств отображается информация о количестве устройств, как общем, так и отключенных. Например, «Total: 50, Disabled: 10» указывает на то, что общее количество устройств в списке составляет 50, при этом 10 из них отключены. Это предоставляет пользователю общий обзор состояния устройств и их доступности для использования.

Кроме того, в приложении присутствует кнопка «Force Refresh», которая позволяет пользователю принудительно обновить список устройств. Нажатие на эту кнопку инициирует процесс обновления списка устройств, что позволяет отобразить их заново с последними доступными данными.

Эта функция полезна в случае, если пользователь внес изменения в систему, связанные с устройствами, и хочет убедиться, что список отображает актуальную информацию. Например, после установки нового устройства или изменения его параметров пользователь может использовать кнопку «Force Refresh», чтобы увидеть изменения в списке устройств без необходимости закрывать и заново открывать приложение.

При нажатии на устройство правой кнопкой мыши открывается контекстное меню, предоставляющее ряд опций для управления устройством (рисунок 5.3).

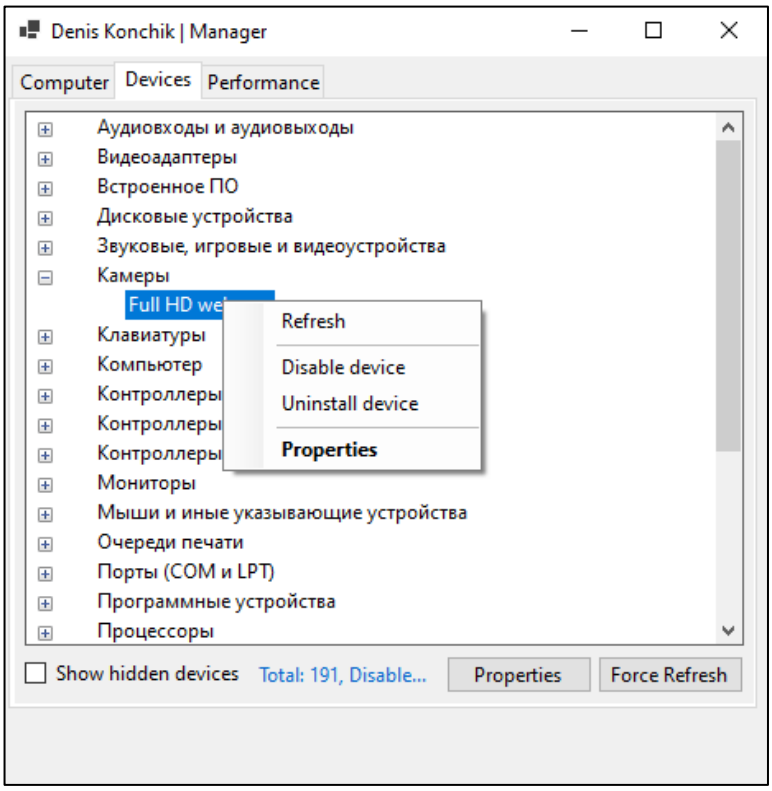

Рисунок 5.3 – Контекстное меню устройства

В этом меню доступны следующие кнопки:

1 Refresh (Обновить). Эта опция позволяет обновить информацию о выбранном устройстве. При выборе этой опции программа перечитывает данные об устройстве, чтобы отобразить актуальное состояние и параметры, такие как статус подключения, наличие драйверов и другие характеристики.

2 Disable Device (Отключить устройство). При выборе этой опции программа временно отключает выбранное устройство. Это означает, что устройство перестает функционировать до тех пор, пока оно не будет снова включено. Отключение устройства может быть полезным в случае, если оно вызывает проблемы или конфликты с другими устройствами.

3 Uninstall Device (Удалить устройство). При выборе этой опции программа инициирует процесс удаления выбранного устройства из системы. Это включает удаление всех связанных с устройством драйверов, файлов и записей в системном реестре. Удаление устройства может быть полезным, если оно больше не используется, повреждено или вызывает серьезные проблемы с системой. Перед удалением устройства рекомендуется убедиться, что для его работы не требуются какие-либо важные программы или функции.

4 Properties. Вызывает стандартное окно Windows (рисунок 5.4) с подробной информацией об устройстве. В этом окне предоставляются различные вкладки, такие как «Общие», «Драйвер», «Сведения», «События», где можно получить дополнительные сведения об устройстве, его состоянии и драйверах.

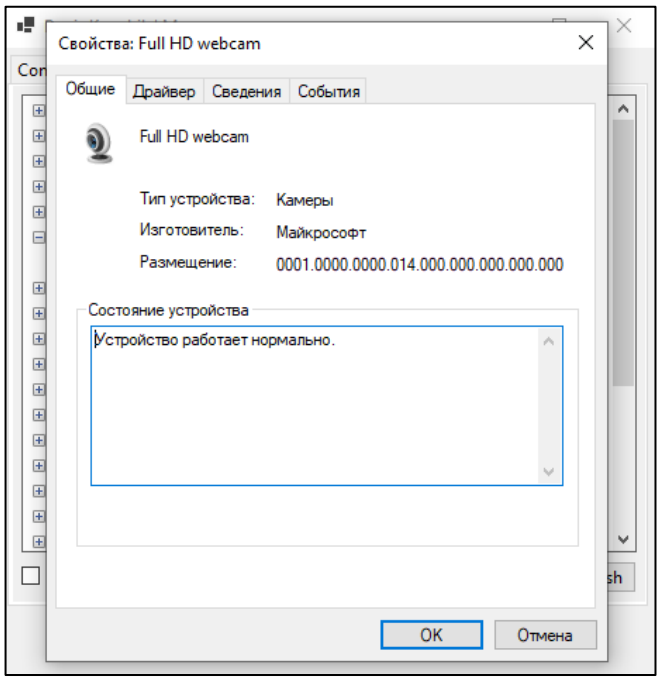

Рисунок 5.4 – Окно с информацией об устройстве

5.3 Производительность подсистем компьютера

В приложении имеется вкладка «Performance», которая предоставляет пользователю информацию о производительности различных подсистем компьютера (рисунок 5.5).

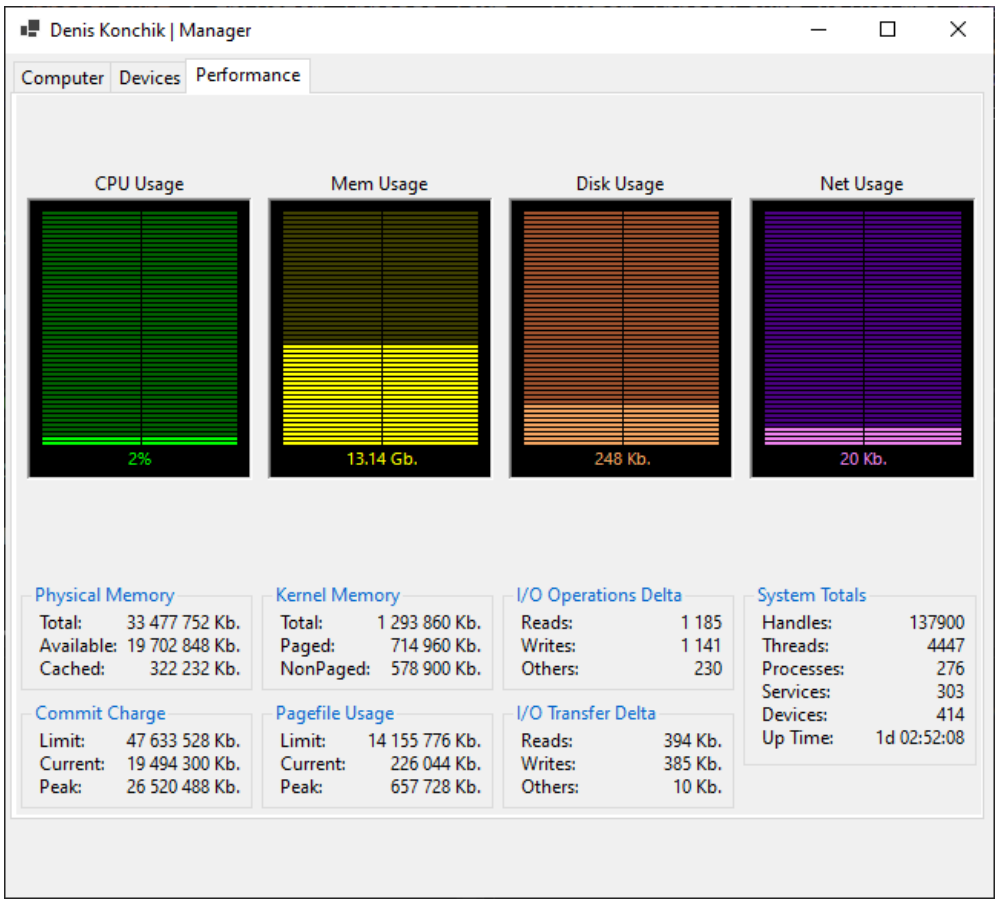

Рисунок 5.5 – Вкладка «Performance»

На этой вкладке данные представлены в виде диаграмм, отображающих основные показатели:

1 Использование процессора (CPU Usage). Данный показатель отображает процент времени, который процессор затрачивает на выполнение задач. Высокое значение может указывать на интенсивную работу приложений или процессов, что может привести к снижению производительности системы.

2 Использование памяти (Mem Usage). Этот показатель отображает объем оперативной памяти, используемой системой. Высокое использование памяти может привести к замедлению работы системы из-за нехватки ресурсов для выполнения задач.

3 Этот показатель отображает активность дискового пространства на компьютере, включая как операции чтения, так и записи. Он предоставляет информацию о загрузке дисковой подсистемы и активности операций ввода/вывода на диск, что позволяет оценить текущую нагрузку на диск и его производительность.

4 Сетевой трафик (Net Usage). Показатель Net Usage предоставляет информацию о сетевой активности компьютера. Он отображает объем переданных и принятых данных по сети, что позволяет оценить загрузку сетевого интерфейса и общую активность сетевого соединения.

Помимо графического отображения производительности подсистем компьютера присутствует отображение некоторой информации в текстовом виде:

1 Physical Memory. Этот раздел предоставляет информацию о физической памяти компьютера. Total – общий объем физической памяти на компьютере. Available – количество доступной физической памяти, которая в данный момент не используется операционной системой. Cached – объем физической памяти, используемой для кэширования данных, что может ускорить доступ к ним.

2 Kernel Memory. Этот раздел отображает информацию о памяти ядра операционной системы. Total – общий объем занимаемой памяти ядром операционной системы. Paged – Количество памяти ядра, которое может быть выгружено на диск в случае нехватки оперативной памяти. Nonpaged – количество памяти ядра, которая не может быть выгружена на диск и всегда должна находиться в оперативной памяти.

3 I/O Operations Delta. Этот раздел отображает дельту операций ввода/вывода на диск. Reads – количество операций чтения с диска за определенный период времени. Writes – количество операций записи на диск за тот же период. Others – количество других операций ввода/вывода, таких как операции перемещения или удаления файлов.

4 System Totals. Этот раздел предоставляет общую статистику по различным системным ресурсам. Handles – общее количество дескрипторов ресурсов, которые используются процессами. Threads – количество потоков, выполняющихся в системе. Processes – общее число процессов, запущенных на компьютере. Services – количество служб, запущенных на компьютере. Devices – общее количество устройств, подключенных к компьютеру. Up Time – время, прошедшее с момента последней перезагрузки компьютера.

5 Commit Charge. Этот раздел отображает информацию о текущем использовании виртуальной памяти. Limit – максимальный объем

виртуальной памяти, который доступен системе. Current – текущее количество использованной виртуальной памяти. Peak – максимальное количество использованной виртуальной памяти за все время работы системы.

6 Pagefile Usage. Этот раздел предоставляет информацию об использовании файла подкачки операционной системой. Limit – максимальный размер файла подкачки, который может использоваться системой. Current – текущий размер файла подкачки, используемый системой. Peak – максимальный размер файла подкачки, использованный системой за все время работы.

7 I/O Transfer Delta. Этот раздел отображает дельту объема данных, переданных через ввод/вывод на диск. Reads – объем данных, прочитанных с диска за определенный период времени. Writes – объем данных, записанных на диск за тот же период. Others – объем других данных, переданных через ввод/вывод, например, для перемещения файлов.

Таким образом, в разделе «Performance» предоставляется обширная информация о производительности различных подсистем компьютера. Графическое представление основных показателей, таких как использование процессора, памяти, дискового пространства и сетевого трафика, позволяет пользователям быстро оценить текущую нагрузку на систему и ее производительность.

Помимо графических диаграмм, в разделе также представлена текстовая информация о различных аспектах производительности компьютера. Это включает данные о физической, операциях ввода/вывода на диск, общей статистике по системным ресурсам, использовании виртуальной памяти и файла подкачки, а также объеме переданных данных через ввод/вывод на диск.

Все эти показатели позволяют пользователям более глубоко анализировать работу компьютера, выявлять возможные проблемы с производительностью и принимать соответствующие меры для их решения. В целом, раздел «Performance» обеспечивает пользователям полную информацию о состоянии производительности компьютерной системы, что позволяет им эффективно управлять ресурсами и оптимизировать работу системы.

ЗАКЛЮЧЕНИЕ

В ходе данного курсового проекта были выполнены задачи, направленные на анализ конфигурации компьютерной системы под управлением операционной системы Windows, а также на изучение производительности подсистем компьютера и управление подключенными устройствами.

Для определения наилучших практик в управлении устройствами и мониторинге производительности были рассмотрены стандартные средства операционной системы Windows. Анализ диспетчера устройств и инструментов мониторинга, предоставляемых ОС, позволил выявить основные функциональные возможности и принципы работы, а также оценить их эффективность и удобство использования.

Для разработки программного обеспечения был выбран язык программирования C# и технологии .NET Framework. Использование C# и .NET позволило создать эффективное и мощное программное решение с удобным пользовательским интерфейсом. Для разработки пользовательского интерфейса была выбрана технология WinForms, которая обеспечила быструю и простую разработку интерфейса, а также интеграцию с функциональностью операционной системы Windows.

В результате было разработано программное обеспечение в виде диспетчера устройств для операционной системы Windows и средства мониторинга производительности. Эти инструменты обеспечивают удобное управление подключенными устройствами и контроль производительности системы, что способствует оптимизации работы компьютера.

Таким образом, данный курсовой проект представляет собой успешную попытку синтезировать знания о конфигурации и производительности компьютерных систем, а также навыки программирования для создания полезных инструментов, способствующих оптимизации работы компьютера под управлением операционной системы Windows. Результаты данного проекта могут быть использованы для повышения эффективности работы компьютерных систем и обеспечения более комфортного пользовательского опыта.

СПИСОК ЛИТЕРАТУРНЫХ ИСТОЧНИКОВ

[1] Доманов, А. Т. Стандарт предприятия / А. Т. Доманов, Н. И. Сорока. – Минск : БГУИР, 2017. – 167 с.

[2] Основные компоненты компьютера и их функции [Электронный ресурс]. – Режим доступа: http://mabi.vspu.ru/portfolio/osnovnyie-komponentyi-kompyutera-i-ih-funktsii/. – Дата доступа: 27.02.2024.

[3] Архитектурные модули Windows NT [Электронный ресурс]. – Режим доступа: studref.com/549707/informatika/arhitekturnye_moduli_windows. – Дата доступа: 01.03.2024.

[4] История развития Microsoft Windows [Электронный ресурс]. – Режим доступа: https://tproger.ru/translations/microsoft-windows-history. – Дата доступа: 01.03.2023.

[5] Клименко, Р. Недокументированные и малоизвестные возможности Windows XP / Р. Клименко. – Спб : Питер, 2006. – 456 с.

[6] Руссинович, М. Внутреннее устройство Microsoft Windows: Windows Server 2003, Windows XP и Windows 2000. Мастер-класс/ 4-е изд. / М. Руссинович, Д. Соломон. – СПб : Питер, 2005. – 992 с.

[7] Чекмарев, А. Н. Microsoft Windows 7, Руководство администратора. / А. Н. Чекмарев. – СПб : БХВ-Петербург, 2010. – 896 с.

[8] Леонтьев, В.П. Новейшая энциклопедия персонального компьютера / В.П. Леонтьев. – М : ОЛМА-ПРЕСС, 1999. – 640 с.

[9] Большаков, Т.В. Операционные системы: учеб. пособие / Т.В. Большаков. – Новосибирск : НГУ, 2005. – 136 с.

[10] Гордеев, А.В. Операционные системы: учеб, для вузов по напр. «Информатика и вычисл. техн» / А.В. Гордеев. – СПБ : Питер, 2004. – 415 с.

[11] Свиридова, М.Ю. Операционная система Windows XP.: учеб. пособие для нач. проф. образования / М.Ю. Свиридова. – М : Академия, 2006. – 189 с.

[12] Симонович, С.В. Информатика. Базовый курс : учебное пособие для вузов / С.В. Симонович, Г.А. Евсеев, В.И. Мураховский. – СПб : Питер, 2004. – 640 с.

[13] Матвеев, М.Д. Самоучитель Microsoft Windows XP: Все об использовании и настройках : методический материал / М.Д. Матвеев. – СПб : Наука и техника, 2005. – 620 с.

[14] Леонтьев, В.П. Осваиваем Windows XP быстро и увлекательно: справочное издание / В.П. Леонтьев. – М. : ОЛМА-ПРЕСС, 2005. – 219 с.

[15] Таненбаум, Э. Современные операционные системы / Э. Таненбаум. – СПб : Питер, 2006. – 1038 с.

[16] Гаевский, А.Ю. Информатика : учебное пособие / А.Ю. Гаевский. – М. : Гамма Пресс, 2004. – 536 с.

[17] Рихтер, Дж. CLR via C#. Программирование на платформе Microsoft .NET Framework 4.0 на языке C#. 3-е изд. / Дж. Рихтер. – СПб : Питер, 2012. – 928 с.

[18] Петцольд, Ч. Программирование для Microsoft Windows на C#. Том 1 / Ч. Петцольд. – М. : Русская редакция, 2002. – 576 с.

[19] Харт, Д. Системное программирование в среде Win32 / Д. Харт. – М. : Издательский дом "Вильямс", 2001. – 437 с.

[20] Руссинович, М. Внутреннее устройство Microsoft Windows. Основные подсистемы ОС / М. Руссинович. – М. : Питер, 2015. – 486 с.

ПРИЛОЖЕНИЕ А
(обязательное)
Функциональная схема

ПРИЛОЖЕНИЕ Б
(обязательное)
Блок схема алгоритма

ПРИЛОЖЕНИЕ В
(обязательное)
Листинг программного кода

Листинг 1 – Класс TaskManagerSystemProps

```
[SupportedOSPlatform("windows")]
internal class TaskManagerSystemProps {
    public string WindowsName { get; private set; }
    public string WindowsEdition { get; private set; }
    public string WindowsVersion { get; private set; }
    public DateTime InstallDate { get; private set; }
    public string RegisterUser { get; private set; }
    public string RegisterCompany { get; private set; }
    public string RegisterKey { get; private set; }
    public string SystemManufacturer { get; private set; }
    public string SystemProductName { get; private set; }
    public string ProcessorVendor { get; private set; }
    public string ProcessorName { get; private set; }
    public string ProcessorFamily { get; private set; }
    public int ProcessorCount { get; private set; }
    public string ProcessorSpeed { get; private set; }
    public int TotalMemory { get; private set; }
    public bool OEMInfo { get; private set; } = false;
    public Image? OEMLogo { get; private set; }
    public string OEMManufacturer { get; private set; }
    public string OEMSupportPhone { get; private set; }
    public string OEMSupportHours { get; private set; }
    public string OEMSupportURL { get; private set; }

    public void Refresh() {
        // Get Windows Version Information
        RegistryKey? rk =
Registry.LocalMachine.OpenSubKey(@"SOFTWARE\Microsoft\Windows
NT\CurrentVersion", false);
        if (rk != null) {
            try { WindowsName = $"Microsoft {rk.GetValue("ProductName")}"; }
catch { }
            try { WindowsEdition = $"{rk.GetValue("EditionID")} Edition"; }
catch { }
            try { WindowsVersion = $"{rk.GetValue("DisplayVersion")} - Build:
{rk.GetValue("CurrentBuild")}.{rk.GetValue("UBR")}"; } catch { }
            try { InstallDate =
DateTime.UnixEpoch.AddSeconds(Convert.ToInt64(rk.GetValue("InstallDate"))).To
LocalTime(); } catch { }
            try { RegisterUser = rk.GetValue("RegisteredOwner")?.ToString()!;
} catch { }
            try { RegisterCompany =
rk.GetValue("RegisteredOrganization")?.ToString()!; } catch { }
            try { RegisterKey =
DecodeProductKey((byte[])rk.GetValue("DigitalProductId")!); } catch { }
        }
        rk?.Close();
        // Get Processor Information
        RegistryKey? rk2 =
Registry.LocalMachine.OpenSubKey(@"HARDWARE\DESCRIPTION\System\CentralProcess
or\0", false);
        if (rk2 != null) {
            try { ProcessorVendor =
rk2.GetValue("VendorIdentifier")?.ToString()!; } catch { }
```

```
            try { ProcessorName =
rk2.GetValue("ProcessorNameString")?.ToString()!; } catch { }
            try { ProcessorFamily = rk2.GetValue("Identifier")?.ToString()!;
} catch { }
            try { ProcessorSpeed = rk2.GetValue("~MHz")?.ToString()!; } catch
{ }
        }
        rk2?.Close();
        // Get System Information
        RegistryKey? rk3 =
Registry.LocalMachine.OpenSubKey(@"HARDWARE\DESCRIPTION\System\BIOS", false);
        if (rk3 != null) {
            try { SystemManufacturer =
rk3.GetValue("SystemManufacturer")?.ToString()!; } catch { }
            try { SystemProductName =
rk3.GetValue("SystemProductName")?.ToString()!; } catch { }
        }
        rk3?.Close();
        // Get OEM Information
        RegistryKey? rk4 =
Registry.LocalMachine.OpenSubKey(@"SOFTWARE\Microsoft\Windows\CurrentVersion\
OEMInformation", false);
        if (rk4 != null) {
            if (File.Exists(rk4.GetValue("Logo")?.ToString()!)) {
                try { OEMLogo =
Image.FromFile(rk4.GetValue("Logo")?.ToString()!); } catch { OEMLogo = null;
}
            }
            try { OEMManufacturer =
rk4.GetValue("Manufacturer")?.ToString()!; } catch { }
            try { OEMSupportPhone =
rk4.GetValue("SupportPhone")?.ToString()!; } catch { }
            try { OEMSupportHours =
rk4.GetValue("SupportHours")?.ToString()!; } catch { }
            try { OEMSupportURL = rk4.GetValue("SupportURL")?.ToString()!; }
catch { }
            OEMInfo = (OEMManufacturer != "" || OEMSupportPhone != "" ||
OEMSupportURL != "");
        } else { OEMInfo = false; }
        rk4?.Close();
        // Get Total Memory
        if (API.GetPhysicallyInstalledSystemMemory(out long memKb)) {
            TotalMemory = (int)(memKb / 1024 / 1024);
        } else { TotalMemory = 0; }

    }
    private static string DecodeProductKey(byte[] digitalProductId) {
        if (digitalProductId == null) return string.Empty;
        if (digitalProductId.Length < 20) return string.Empty;
        // Offset of first byte of encoded product key in
'DigitalProductIdxxx" REG_BINARY value. Offset = 34H.
        const int keyStartIndex = 52;
        // Offset of last byte of encoded product key in
'DigitalProductIdxxx" REG_BINARY value. Offset = 43H.
        const int keyEndIndex = keyStartIndex + 15;
        // Possible alpha-numeric characters in product key.
        char[] digits = new char[] { 'B', 'C', 'D', 'F', 'G', 'H', 'J', 'K',
'M', 'P', 'Q', 'R', 'T', 'V', 'W', 'X', 'Y', '2', '3', '4', '6', '7', '8',
'9', };
        // Length of decoded product key
        const int decodeLength = 29;
        // Length of decoded product key in byte-form.
```

```
        // Each byte represents 2 chars.
        const int decodeStringLength = 15;
        // Array of containing the decoded product key.
        char[] decodedChars = new char[decodeLength];
        // Extract byte 52 to 67 inclusive.
        ArrayList hexPid = new();
        for (int i = keyStartIndex; i <= keyEndIndex; i++) {
            hexPid.Add(digitalProductId[i]);
        }
        for (int i = decodeLength - 1; i >= 0; i--) {
            // Every sixth char is a separator.
            if ((i + 1) % 6 == 0) {
                decodedChars[i] = '-';
            } else {
                // Do the actual decoding.
                int digitMapIndex = 0;
                for (int j = decodeStringLength - 1; j >= 0; j--) {
                    int byteValue = (digitMapIndex << 8) | (byte)hexPid[j]!;
                    hexPid[j] = (byte)(byteValue / 24);
                    digitMapIndex = byteValue % 24;
                    decodedChars[i] = digits[digitMapIndex];
                }
            }
        }
        return new string(decodedChars);
    }

}
```

Листинг 2 – Класс TaskManagerDevice

```
[SupportedOSPlatform("windows")]
internal class TaskManagerDevice : IComparable,
IComparable<TaskManagerDevice> {
    public TaskManagerDevice(TaskManagerDeviceClass cls, string name,
SP_DEVINFO_DATA data, bool connected) {
        Class = cls;
        Name = name;
        Data = data;
        ImageKey = data.ClassGuid + "-" + data.DevInst;
        Present = connected;
    }

    public string Name { get; }
    public TaskManagerDeviceClass Class { get; }
    public SP_DEVINFO_DATA Data { get; }
    public bool Present { get; }
    public bool Disabled { get => ProblemCode == 22; }
    public string InstanceID { get; set; } = "";
    public string Service { get; set; } = "";
    public int ProblemCode { get; set; } = 0;
    public string ImageKey { get; }
    public override string ToString() => Name;

    int IComparable.CompareTo(object? obj) => CompareTo(obj as
TaskManagerDevice);
    public int CompareTo(TaskManagerDevice? other) =>
Name.CompareTo(other?.Name);

}

[SupportedOSPlatform("windows")]
```

```
internal class TaskManagerDeviceClass : IComparable,
IComparable<TaskManagerDeviceClass> {
    private static readonly DEVPROPKEY DEVPKEY_Device_DeviceDesc =
new("a45c254e-df1c-4efd-8020-67d146a850e0", 2);
    private static readonly DEVPROPKEY DEVPKEY_Device_Service =
new("a45c254e-df1c-4efd-8020-67d146a850e0", 6);
    private static readonly DEVPROPKEY DEVPKEY_Device_ClassGuid =
new("a45c254e-df1c-4efd-8020-67d146a850e0", 10);
    private static readonly DEVPROPKEY DEVPKEY_Device_FriendlyName =
new("a45c254e-df1c-4efd-8020-67d146a850e0", 14);
    private static readonly DEVPROPKEY DEVPKEY_Device_InstallState =
new("a45c254e-df1c-4efd-8020-67d146a850e0", 36);
    private static readonly DEVPROPKEY DEVPKEY_DeviceClass_IconPath =
new("259abffc-50a7-47ce-af08-68c9a7d73366", 12);
    private static readonly DEVPROPKEY DEVPKEY_Device_ProblemCode =
new("4340a6c5-93fa-4706-972c-7b648008a5a7", 3);
    private static readonly DEVPROPKEY DEVPKEY_Device_InstanceId =
new("78c34fc8-104a-4aca-9ea4-524d52996e57", 256);
    private readonly List<TaskManagerDevice> _devices = new();

    public TaskManagerDeviceClass(Guid classId, string description, string
iconPath) {
        ClassId = classId;
        Description = description;
        IconPath = iconPath;
    }

    public Guid ClassId { get; }
    public string Description { get; }
    public string IconPath { get; private set; }
    public IReadOnlyList<TaskManagerDevice> Devices => _devices;

    int IComparable.CompareTo(object? obj) => CompareTo(obj as
TaskManagerDeviceClass);
    public int CompareTo(TaskManagerDeviceClass? other) =>
Description.CompareTo(other?.Description);

    public static IReadOnlyList<TaskManagerDeviceClass>
Load(TaskManagerDeviceFilter fiter, ref ImageList il) {
        var list = new List<TaskManagerDeviceClass>();
        var hdevinfo = SetupDiGetClassDevs(IntPtr.Zero, null, IntPtr.Zero,
fiter);

        try {
            var data = new SP_DEVINFO_DATA { cbSize =
Marshal.SizeOf<SP_DEVINFO_DATA>() };
            int index = 0;
            while (SetupDiEnumDeviceInfo(hdevinfo, index, ref data)) {
                index++;
                var classId = GetGuidProperty(hdevinfo, ref data,
DEVPKEY_Device_ClassGuid);
                if (classId == Guid.Empty) continue;
                var cls = list.FirstOrDefault(c => c.ClassId == classId);
                if (cls == null) {
                    string classDescription = GetClassDescription(classId);
                    string classIconPath = GetStringProperty(classId,
DEVPKEY_DeviceClass_IconPath);
                    cls = new TaskManagerDeviceClass(classId,
classDescription, classIconPath);
                    list.Add(cls);
                }
```

```
                string name = GetStringProperty(hdevinfo, ref data,
DEVPKEY_Device_FriendlyName);
                if (string.IsNullOrWhiteSpace(name)) { name =
GetStringProperty(hdevinfo, ref data, DEVPKEY_Device_DeviceDesc); }
                int state = GetIntProperty(hdevinfo, ref data,
DEVPKEY_Device_InstallState);

                var dev = new TaskManagerDevice(cls, name, data, state == 0)
{
                    Service = GetStringProperty(hdevinfo, ref data,
DEVPKEY_Device_Service),
                    InstanceID = GetStringProperty(hdevinfo, ref data,
DEVPKEY_Device_InstanceId),
                    ProblemCode = GetIntProperty(hdevinfo, ref data,
DEVPKEY_Device_ProblemCode)
                };
                if (il.Images.ContainsKey(dev.ImageKey)) {
                    // We better overwrite the images each time
                    il.Images.RemoveByKey(dev.ImageKey);
                }
                if (!il.Images.ContainsKey(dev.ImageKey)) {
                    SetupDiLoadDeviceIcon(hdevinfo, ref data, 16, 16, 0, out
IntPtr devIcon);
                    if (devIcon != IntPtr.Zero) {
                        // We need to dim the icon if state!=0
                        if (dev.Present) {
                            il.Images.Add(dev.ImageKey,
Icon.FromHandle(devIcon));
                        } else {
                            Icon icn = Icon.FromHandle(devIcon);
                            il.Images.Add(dev.ImageKey,
SetImageOpacity(icn.ToBitmap(), 0.60F));
                            icn.Dispose();
                        }
                    } else {
                        // Draw a generic icon
                        il.Images.Add(dev.ImageKey, new Bitmap(16, 16));
                    }
                    if (devIcon != IntPtr.Zero) { DestroyIcon(devIcon); }
                    // We need to check if down, overlay icon
                    if (dev.ProblemCode > 0 && il.Images[dev.ImageKey] !=
null) {
                        var icn = il.Images[dev.ImageKey]!;
                        var g = Graphics.FromImage(icn);
                        if (dev.ProblemCode == 22) {
                            g.DrawImage(Resources.Resources.Device_Down, 0,
0);
                        } else {
                            g.DrawImage(Resources.Resources.Device_Warning,
0, 0);
                        }
                        il.Images.RemoveByKey(dev.ImageKey);
                        il.Images.Add(dev.ImageKey, icn);
                        g.Dispose(); icn.Dispose();
                    }

                }
                cls._devices.Add(dev);
            }
        } finally {
            if (hdevinfo != IntPtr.Zero) {
SetupDiDestroyDeviceInfoList(hdevinfo); }
```

```
        }

        foreach (var cls in list) {
            cls._devices.Sort();
        }
        list.Sort();
        return list;
    }
    private static string GetClassDescription(Guid classId) {
        SetupDiGetClassDescription(ref classId, IntPtr.Zero, 0, out int
size);
        if (size == 0) return string.Empty;
        var ptr = Marshal.AllocCoTaskMem(size * 2);
        try {
            if (SetupDiGetClassDescription(ref classId, ptr, size, out size))
{
                return Marshal.PtrToStringUni(ptr, size - 1);
            }
        } finally { Marshal.FreeCoTaskMem(ptr); }
        return string.Empty;
    }
    private static string GetStringProperty(IntPtr hdevinfo, ref
SP_DEVINFO_DATA data, DEVPROPKEY pk) {
        SetupDiGetDeviceProperty(hdevinfo, ref data, ref pk, out int _,
IntPtr.Zero, 0, out int size, 0);
        if (size == 0) return string.Empty;
        var ptr = Marshal.AllocCoTaskMem(size);
        try {
            if (SetupDiGetDeviceProperty(hdevinfo, ref data, ref pk, out int
propertyType, ptr, size, out size, 0)) {
                return Marshal.PtrToStringUni(ptr, (size / 2) - 1);
            }
        } finally { Marshal.FreeCoTaskMem(ptr); }
        return string.Empty;
    }
    private static int GetIntProperty(IntPtr hdevinfo, ref SP_DEVINFO_DATA
data, DEVPROPKEY pk) {
        SetupDiGetDeviceProperty(hdevinfo, ref data, ref pk, out int _,
IntPtr.Zero, 0, out int size, 0);
        if (size == 0) { return -2; } // NOT FOUND
        var ptr = Marshal.AllocCoTaskMem(size);
        try {
            if (SetupDiGetDeviceProperty(hdevinfo, ref data, ref pk, out int
propertyType, ptr, size, out size, 0)) {
                return Marshal.ReadInt32(ptr);
            }
        } finally { Marshal.FreeCoTaskMem(ptr); }
        return -1; // NOT SUPPORTED
    }
    private static Guid GetGuidProperty(IntPtr hdevinfo, ref SP_DEVINFO_DATA
data, DEVPROPKEY pk) {
        SetupDiGetDeviceProperty(hdevinfo, ref data, ref pk, out int _, out
Guid guid, 16, out int _, 0);
        return guid;
    }
    private static string GetStringProperty(Guid ClassGuid, DEVPROPKEY pk) {
        SetupDiGetClassProperty(ClassGuid, ref pk, out int _, IntPtr.Zero, 0,
out int size, 1);
        if (size == 0) { return string.Empty; }
        var ptr = Marshal.AllocCoTaskMem(size);
        try {
```

```
            if (SetupDiGetClassProperty(ClassGuid, ref pk, out int
propertyType, ptr, size, out size, 1)) {
                return Marshal.PtrToStringUni(ptr, (size / 2) - 1);
            }
        } finally { Marshal.FreeCoTaskMem(ptr); }
        return string.Empty;
    }
    private static Image SetImageOpacity(Image image, float opacity) {
        try {
            Bitmap bmp = new(image.Width, image.Height);
            using (Graphics gfx = Graphics.FromImage(bmp)) {
                ColorMatrix matrix = new() { Matrix33 = opacity };
                ImageAttributes attributes = new();
                attributes.SetColorMatrix(matrix, ColorMatrixFlag.Default,
ColorAdjustType.Bitmap);
                gfx.DrawImage(image, new Rectangle(0, 0, bmp.Width,
bmp.Height), 0, 0, image.Width, image.Height, GraphicsUnit.Pixel,
attributes);
            }
            return bmp;
        } catch { return image; }
    }

}
```

Листинг 3 – Класс TaskManagerDeviceClass

```
[SupportedOSPlatform("windows")]
internal class TaskManagerDeviceClass : IComparable,
IComparable<TaskManagerDeviceClass> {
    private static readonly DEVPROPKEY DEVPKEY_Device_DeviceDesc =
new("a45c254e-df1c-4efd-8020-67d146a850e0", 2);
    private static readonly DEVPROPKEY DEVPKEY_Device_Service =
new("a45c254e-df1c-4efd-8020-67d146a850e0", 6);
    private static readonly DEVPROPKEY DEVPKEY_Device_ClassGuid =
new("a45c254e-df1c-4efd-8020-67d146a850e0", 10);
    private static readonly DEVPROPKEY DEVPKEY_Device_FriendlyName =
new("a45c254e-df1c-4efd-8020-67d146a850e0", 14);
    private static readonly DEVPROPKEY DEVPKEY_Device_InstallState =
new("a45c254e-df1c-4efd-8020-67d146a850e0", 36);
    private static readonly DEVPROPKEY DEVPKEY_DeviceClass_IconPath =
new("259abffc-50a7-47ce-af08-68c9a7d73366", 12);
    private static readonly DEVPROPKEY DEVPKEY_Device_ProblemCode =
new("4340a6c5-93fa-4706-972c-7b648008a5a7", 3);
    private static readonly DEVPROPKEY DEVPKEY_Device_InstanceId =
new("78c34fc8-104a-4aca-9ea4-524d52996e57", 256);
    private readonly List<TaskManagerDevice> _devices = new();

    public TaskManagerDeviceClass(Guid classId, string description, string
iconPath) {
        ClassId = classId;
        Description = description;
        IconPath = iconPath;
    }

    public Guid ClassId { get; }
    public string Description { get; }
    public string IconPath { get; private set; }
    public IReadOnlyList<TaskManagerDevice> Devices => _devices;

    int IComparable.CompareTo(object? obj) => CompareTo(obj as
TaskManagerDeviceClass);
```

```
    public int CompareTo(TaskManagerDeviceClass? other) =>
Description.CompareTo(other?.Description);

    public static IReadOnlyList<TaskManagerDeviceClass>
Load(TaskManagerDeviceFilter fiter, ref ImageList il) {
        var list = new List<TaskManagerDeviceClass>();
        var hdevinfo = SetupDiGetClassDevs(IntPtr.Zero, null, IntPtr.Zero,
fiter);

        try {
            var data = new SP_DEVINFO_DATA { cbSize =
Marshal.SizeOf<SP_DEVINFO_DATA>() };
            int index = 0;
            while (SetupDiEnumDeviceInfo(hdevinfo, index, ref data)) {
                index++;
                var classId = GetGuidProperty(hdevinfo, ref data,
DEVPKEY_Device_ClassGuid);
                if (classId == Guid.Empty) continue;
                var cls = list.FirstOrDefault(c => c.ClassId == classId);
                if (cls == null) {
                    string classDescription = GetClassDescription(classId);
                    string classIconPath = GetStringProperty(classId,
DEVPKEY_DeviceClass_IconPath);
                    cls = new TaskManagerDeviceClass(classId,
classDescription, classIconPath);
                    list.Add(cls);
                }

                string name = GetStringProperty(hdevinfo, ref data,
DEVPKEY_Device_FriendlyName);
                if (string.IsNullOrWhiteSpace(name)) { name =
GetStringProperty(hdevinfo, ref data, DEVPKEY_Device_DeviceDesc); }
                int state = GetIntProperty(hdevinfo, ref data,
DEVPKEY_Device_InstallState);

                var dev = new TaskManagerDevice(cls, name, data, state == 0)
{
                    Service = GetStringProperty(hdevinfo, ref data,
DEVPKEY_Device_Service),
                    InstanceID = GetStringProperty(hdevinfo, ref data,
DEVPKEY_Device_InstanceId),
                    ProblemCode = GetIntProperty(hdevinfo, ref data,
DEVPKEY_Device_ProblemCode)
                };
                if (il.Images.ContainsKey(dev.ImageKey)) {
                    // We better overwrite the images each time
                    il.Images.RemoveByKey(dev.ImageKey);
                }
                if (!il.Images.ContainsKey(dev.ImageKey)) {
                    SetupDiLoadDeviceIcon(hdevinfo, ref data, 16, 16, 0, out
IntPtr devIcon);
                    if (devIcon != IntPtr.Zero) {
                        // We need to dim the icon if state!=0
                        if (dev.Present) {
                            il.Images.Add(dev.ImageKey,
Icon.FromHandle(devIcon));
                        } else {
                            Icon icn = Icon.FromHandle(devIcon);
                            il.Images.Add(dev.ImageKey,
SetImageOpacity(icn.ToBitmap(), 0.60F));
                            icn.Dispose();
                        }
```

```
                    } else {
                        // Draw a generic icon
                        il.Images.Add(dev.ImageKey, new Bitmap(16, 16));
                    }
                    if (devIcon != IntPtr.Zero) { DestroyIcon(devIcon); }
                    // We need to check if down, overlay icon
                    if (dev.ProblemCode > 0 && il.Images[dev.ImageKey] !=
null) {
                        var icn = il.Images[dev.ImageKey]!;
                        var g = Graphics.FromImage(icn);
                        if (dev.ProblemCode == 22) {
                            g.DrawImage(Resources.Resources.Device_Down, 0,
0);
                        } else {
                            g.DrawImage(Resources.Resources.Device_Warning,
0, 0);
                        }
                        il.Images.RemoveByKey(dev.ImageKey);
                        il.Images.Add(dev.ImageKey, icn);
                        g.Dispose(); icn.Dispose();
                    }

                }
                cls._devices.Add(dev);
            }
        } finally {
            if (hdevinfo != IntPtr.Zero) {
SetupDiDestroyDeviceInfoList(hdevinfo); }
        }

        foreach (var cls in list) {
            cls._devices.Sort();
        }
        list.Sort();
        return list;
    }
    private static string GetClassDescription(Guid classId) {
        SetupDiGetClassDescription(ref classId, IntPtr.Zero, 0, out int
size);
        if (size == 0) return string.Empty;
        var ptr = Marshal.AllocCoTaskMem(size * 2);
        try {
            if (SetupDiGetClassDescription(ref classId, ptr, size, out size))
{
                return Marshal.PtrToStringUni(ptr, size - 1);
            }
        } finally { Marshal.FreeCoTaskMem(ptr); }
        return string.Empty;
    }
    private static string GetStringProperty(IntPtr hdevinfo, ref
SP_DEVINFO_DATA data, DEVPROPKEY pk) {
        SetupDiGetDeviceProperty(hdevinfo, ref data, ref pk, out int _,
IntPtr.Zero, 0, out int size, 0);
        if (size == 0) return string.Empty;
        var ptr = Marshal.AllocCoTaskMem(size);
        try {
            if (SetupDiGetDeviceProperty(hdevinfo, ref data, ref pk, out int
propertyType, ptr, size, out size, 0)) {
                return Marshal.PtrToStringUni(ptr, (size / 2) - 1);
            }
        } finally { Marshal.FreeCoTaskMem(ptr); }
        return string.Empty;
```

```
    }
    private static int GetIntProperty(IntPtr hdevinfo, ref SP_DEVINFO_DATA
data, DEVPROPKEY pk) {
        SetupDiGetDeviceProperty(hdevinfo, ref data, ref pk, out int _,
IntPtr.Zero, 0, out int size, 0);
        if (size == 0) { return -2; } // NOT FOUND
        var ptr = Marshal.AllocCoTaskMem(size);
        try {
            if (SetupDiGetDeviceProperty(hdevinfo, ref data, ref pk, out int
propertyType, ptr, size, out size, 0)) {
                return Marshal.ReadInt32(ptr);
            }
        } finally { Marshal.FreeCoTaskMem(ptr); }
        return -1; // NOT SUPPORTED
    }
    private static Guid GetGuidProperty(IntPtr hdevinfo, ref SP_DEVINFO_DATA
data, DEVPROPKEY pk) {
        SetupDiGetDeviceProperty(hdevinfo, ref data, ref pk, out int _, out
Guid guid, 16, out int _, 0);
        return guid;
    }
    private static string GetStringProperty(Guid ClassGuid, DEVPROPKEY pk) {
        SetupDiGetClassProperty(ClassGuid, ref pk, out int _, IntPtr.Zero, 0,
out int size, 1);
        if (size == 0) { return string.Empty; }
        var ptr = Marshal.AllocCoTaskMem(size);
        try {
            if (SetupDiGetClassProperty(ClassGuid, ref pk, out int
propertyType, ptr, size, out size, 1)) {
                return Marshal.PtrToStringUni(ptr, (size / 2) - 1);
            }
        } finally { Marshal.FreeCoTaskMem(ptr); }
        return string.Empty;
    }
    private static Image SetImageOpacity(Image image, float opacity) {
        try {
            Bitmap bmp = new(image.Width, image.Height);
            using (Graphics gfx = Graphics.FromImage(bmp)) {
                ColorMatrix matrix = new() { Matrix33 = opacity };
                ImageAttributes attributes = new();
                attributes.SetColorMatrix(matrix, ColorMatrixFlag.Default,
ColorAdjustType.Bitmap);
                gfx.DrawImage(image, new Rectangle(0, 0, bmp.Width,
bmp.Height), 0, 0, image.Width, image.Height, GraphicsUnit.Pixel,
attributes);
            }
            return bmp;
        } catch { return image; }
    }

}
```

Листинг 4 – Класс TaskManagerSystem

```
[SupportedOSPlatform("windows")]
internal class TaskManagerSystem : TaskManagerValuesBase {
    private API.SYSTEM_PERFORMANCE_INFORMATION _SPI = new();
    private API.PERFORMANCE_INFORMATION _PI = new();
    private readonly uint Multiplier = 4096U;
    private ulong _sumPageFileTotal, _sumPageFileUsed, _sumPageFilePeak;
    private readonly CpuUsage _Cpu = new();
    private TimeSpan _UpTime;

    public TaskManagerSystem() {
        _PI.cb = (uint)Marshal.SizeOf(_PI);
    }

    public event EventHandler? RefreshStarting;
    public event EventHandler? RefreshCompleted;

    public void Refresh(bool cancellingEvents = false) {
        CancellingEvents = cancellingEvents;
        if (LastUpdate == 0) { LastUpdate = DateTime.Now.Ticks - 10; }
        if (!CancellingEvents) RefreshStarting?.Invoke(this, new());

        // Compute System Uptime
        if (Environment.TickCount > 0) {
            _UpTime = TimeSpan.FromTicks(Environment.TickCount * 10000L);
        } else {
            _UpTime = TimeSpan.FromTicks((int.MaxValue * 10000L) +
((Environment.TickCount & int.MaxValue) * 10000L));
        }

        // Compute Functions Invokes
        if
(API.NtQuerySystemInformation(API.SYSTEM_INFORMATION_CLASS.SystemPerformanceI
nformation, ref _SPI, Marshal.SizeOf(_SPI), out _) == 0) {
            ioReadCount.SetValue(_SPI.IoReadOperationCount);
            ioWriteCount.SetValue(_SPI.IoWriteOperationCount);
            ioOtherCount.SetValue(_SPI.IoOtherOperationCount);
            ioReadBytes.SetValue(_SPI.IoReadTransferCount);
            ioWriteBytes.SetValue(_SPI.IoWriteTransferCount);
            ioOtherBytes.SetValue(_SPI.IoOtherTransferCount);
            SystemCached.SetValue(_SPI.ResidentSystemCachePage * Multiplier);
            ioTotalCount.SetValue(ioReadCount.Value + ioWriteCount.Value +
ioOtherCount.Value);
            ioTotalBytes.SetValue(ioReadBytes.Value + ioWriteBytes.Value +
ioOtherBytes.Value);

        } else { Debug.WriteLine(Marshal.GetLastPInvokeErrorMessage()); }
        if (API.GetPerformanceInfo(ref _PI, _PI.cb)) {
            HandleCount.SetValue(_PI.HandleCount);
            ThreadCount.SetValue(_PI.ThreadCount);
            ProcessCount.SetValue(_PI.ProcessCount);
            KernelTotal.SetValue(_PI.KernelTotal * Multiplier);
            KernelPaged.SetValue(_PI.KernelPaged * Multiplier);
            KernelNonPaged.SetValue(_PI.KernelNonPaged * Multiplier);
            CommitTotal.SetValue(_PI.CommitTotal * Multiplier);
            CommitLimit.SetValue(_PI.CommitLimit * Multiplier);
            CommitPeak.SetValue(_PI.CommitPeak * Multiplier);
            PhysicalTotal.SetValue(_PI.PhysicalTotal * Multiplier);
            PhysicalAvail.SetValue(_PI.PhysicalAvailable * Multiplier);

        } else { Debug.WriteLine(Marshal.GetLastPInvokeErrorMessage()); }
```

```
        // Compute PageFile Data
        ResetPageFile();
        API.EnumPageFiles(EnumPageFileCallback, IntPtr.Zero);
        PageFileTotal.SetValue(_sumPageFileTotal);
        PageFileUsed.SetValue(_sumPageFileUsed);
        PageFilePeak.SetValue(_sumPageFilePeak);

        // Get Devices & Services Count - This adds too much overhead, either
with API or NATIVE calls, so we just get it once
        if (DevicesCount.Value == 0)
DevicesCount.SetValue(ServiceController.GetDevices().Length);
        if (ServicesCount.Value == 0)
ServicesCount.SetValue(ServiceController.GetServices().Length);

        // Get CPU Usage
        _Cpu.Refresh(LastUpdate);
        CpuUsage.SetValue(_Cpu.Usage);
        CpuUsageUser.SetValue(_Cpu.UserUsage);
        CpuUsageKernel.SetValue(_Cpu.KernelUsage);

        // Compute ETW Usages
        if (Shared.ETW.Running) {
            Shared.ETW.Flush();
            DiskRead.SetValue(Shared.ETW.Stats(0).DiskReaded);
            DiskWrite.SetValue(Shared.ETW.Stats(0).DiskWroted);
            NetSent.SetValue(Shared.ETW.Stats(0).NetSent);
            NetReceived.SetValue(Shared.ETW.Stats(0).NetReceived);
        }

        LastUpdate = DateTime.Now.Ticks;
        if (!CancellingEvents) RefreshCompleted?.Invoke(this, new());
        CancellingEvents = false;
    }

    private bool EnumPageFileCallback(IntPtr lpContext, ref
API.PAGE_FILE_INFORMATION Info, string Name) {
        _sumPageFileTotal += Info.TotalSize * 4096UL;
        _sumPageFileUsed += Info.TotalInUse * 4096UL;
        _sumPageFilePeak += Info.PeakUsage * 4096UL;
        return true;
    }
    private void ResetPageFile() {
        _sumPageFileTotal = 0;
        _sumPageFileUsed = 0;
        _sumPageFilePeak = 0;
    }

    public string UpTime => string.Format("{0}d {1,2:D2}:{2,2:D2}:{3,2:D2}",
_UpTime.Days, _UpTime.Hours, _UpTime.Minutes, _UpTime.Seconds);
    public int MemoryUsage {
        get {
            if (PhysicalTotal.Value - PhysicalAvail.Value <= 0) {
                return 100;
            } else {
                return (int)(100 - ((100 * PhysicalAvail.Value) /
PhysicalTotal.Value));
            }
        }
    }
    public int SwapUsage {
        get {
```

```
            if (PageFileUsed.Value >= PageFileTotal.Value) {
                return 100;
            } else {
                return (int)(100 * PageFileUsed.Value / PageFileTotal.Value);
            }
        }
    }
    public string MemoryUsageString {
        get {
            if (PhysicalTotal.Value - PhysicalAvail.Value == 0) { return
"Full"; }
            if (((PhysicalTotal.Value - PhysicalAvail.Value) / 1024 / 1024) <
2000) { // Less than 2Gb
                return string.Format("{0:#} Mb.",
(double)(PhysicalTotal.Value - PhysicalAvail.Value) / 1024 / 1024);
            } else {
                return string.Format("{0:#.00} Gb.",
(double)(PhysicalTotal.Value - PhysicalAvail.Value) / 1024 / 1024 / 1024);
            }
        }
    }
    public Int128 ioDataUsage => ioReadBytes.Delta + ioWriteBytes.Delta +
ioOtherBytes.Delta;
    public string ioDataUsageString {
        get {
            if (ioDataUsage == 0) {
                return "Idle";
            } else if (ioDataUsage < 2048) {
                return string.Format("{0:#,0} b.", ioDataUsage);
            } else if (ioDataUsage < (1024 * 1024)) {
                return string.Format("{0:#,0} Kb.", (double)(ioDataUsage /
1024));
            } else {
                return string.Format("{0:#.0} Mb.", (double)ioDataUsage /
1024 / 1024);
            }
        }
    }

    public Int128 DiskUsage => DiskRead.Delta + DiskWrite.Delta;
    public string DiskUsageString {
        get {
            if (DiskUsage == 0) {
                return "Idle";
            } else if (DiskUsage < 2048) {
                return string.Format("{0:#,0} b.", DiskUsage);
            } else if (DiskUsage < (1024 * 1024)) {
                return string.Format("{0:#,0} Kb.", (double)DiskUsage /
1024);
            } else {
                return string.Format("{0:#.0} Mb.", (double)DiskUsage / 1024
/ 1024);
            }
        }
    }
    public Int128 NetworkUsage => NetSent.Delta + NetReceived.Delta;
    public string NetworkUsageString {
        get {
            if (NetworkUsage == 0) {
                return "Idle";
            } else if (NetworkUsage < 2048) {
                return string.Format("{0:#,0} b.", NetworkUsage);
```

```
            } else if (NetworkUsage < (1024 * 1024)) {
                return string.Format("{0:#,0} Kb.", (double)NetworkUsage /
1024);
            } else {
                return string.Format("{0:#.0} Mb.", (double)NetworkUsage /
1024 / 1024);
            }
        }
    }

    // Disk Stats
    public Metric DiskRead { get; private set; } = new("DiskRead",
MetricFormats.Kb);
    public Metric DiskWrite { get; private set; } = new("DiskWrite",
MetricFormats.Kb);
    // Net Stats
    public Metric NetSent { get; private set; } = new("NetSent",
MetricFormats.Kb);
    public Metric NetReceived { get; private set; } = new("NetReceived",
MetricFormats.Kb);
    // io Stats
    public Metric ioReadCount { get; private set; } = new("ioReadCount",
MetricFormats.Numeric);
    public Metric ioReadBytes { get; private set; } = new("ioReadBytes",
MetricFormats.Kb);
    public Metric ioWriteCount { get; private set; } = new("ioWriteCount",
MetricFormats.Numeric);
    public Metric ioWriteBytes { get; private set; } = new("ioWriteBytes",
MetricFormats.Kb);
    public Metric ioOtherCount { get; private set; } = new("ioOtherCount",
MetricFormats.Numeric);
    public Metric ioOtherBytes { get; private set; } = new("ioOtherBytes",
MetricFormats.Kb);
    // io Calculated
    public Metric ioUsageCount { get; private set; } = new("ioUsageCount",
MetricFormats.Numeric);
    public Metric ioUsageBytes { get; private set; } = new("ioUsageBytes",
MetricFormats.Kb);
    public Metric ioTotalCount { get; private set; } = new("ioTotalCount",
MetricFormats.Numeric);
    public Metric ioTotalBytes { get; private set; } = new("ioTotalBytes",
MetricFormats.Kb);

    // Commit Memory Stats
    public Metric CommitTotal { get; private set; } = new("CommitTotal",
MetricFormats.Kb);
    public Metric CommitLimit { get; private set; } = new("CommitLimit",
MetricFormats.Kb);
    public Metric CommitPeak { get; private set; } = new("CommitPeak",
MetricFormats.Kb);
    // Physical Memory Stats
    public Metric PhysicalTotal { get; private set; } = new("PhysicalTotal",
MetricFormats.Kb);
    public Metric PhysicalAvail { get; private set; } = new("PhysicalAvail",
MetricFormats.Kb);
    public Metric SystemCached { get; private set; } = new("SystemCached",
MetricFormats.Kb);
    // Kernel Memory Stats
    public Metric KernelTotal { get; private set; } = new("KernelTotal",
MetricFormats.Kb);
    public Metric KernelPaged { get; private set; } = new("KernelPaged",
MetricFormats.Kb);
```

```
    public Metric KernelNonPaged { get; private set; } =
new("KernelNonPaged", MetricFormats.Kb);
    // Page File Stats
    public Metric PageFileTotal { get; private set; } = new("PageFileTotal",
MetricFormats.Kb);
    public Metric PageFileUsed { get; private set; } = new("PageFileUsed",
MetricFormats.Kb);
    public Metric PageFilePeak { get; private set; } = new("PageFilePeak",
MetricFormats.Kb);
    // CPU Usage Stats
    public Metric CpuUsage { get; private set; } = new("CpuUsage",
MetricFormats.None);
    public Metric CpuUsageUser { get; private set; } = new("CpuUsageUser",
MetricFormats.None);
    public Metric CpuUsageKernel { get; private set; } =
new("CpuUsageKernel", MetricFormats.None);
    // System Counts Stats
    public Metric HandleCount { get; private set; } = new("HandleCount");
    public Metric ThreadCount { get; private set; } = new("ThreadCount");
    public Metric ProcessCount { get; private set; } = new("ProcessCount");
    public Metric DevicesCount { get; private set; } = new("DevicesCount");
    public Metric ServicesCount { get; private set; } = new("ServicesCount");
}
```

Листинг 5 – Класс Metric

```
internal class Metric {
    private Int128 _Value, _Delta;
    private bool _isFirst = true;
    private readonly string _Name;

    public Metric(string name) { _Name = name; }
    public Metric(string name, MetricFormats format) { _Name = name;
SetFormat(format); }
    public Metric(string name, string formatString) { _Name = name;
FormatString = formatString; }

    protected virtual void OnValueChanged(MetricChangedEventArgs e) {
AnyChanged?.Invoke(this, e); ValueChanged?.Invoke(this, e); }
    protected virtual void OnDeltaChanged(MetricChangedEventArgs e) {
AnyChanged?.Invoke(this, e); DeltaChanged?.Invoke(this, e); }
    public event EventHandler? AnyChanged, ValueChanged, DeltaChanged;
    public delegate void EventHandler(Metric sender, MetricChangedEventArgs
e);

    public Int128 Value => _Value;
    public Int128 Delta => _Delta;
    public string FormatString { get; set; } = "{0}";
    public MetricFormats Format { get; set; } = 0;
    public string ValueFmt => string.Format(FormatString, _Value / Divider);
    public string DeltaFmt => string.Format(FormatString, _Delta / Divider);
    public string FullString() => (Title == "") ? $"{Name}: {ValueFmt}" :
$"{Title}: {ValueFmt}";
    public override string ToString() => _Value.ToString();
    public int Divider { get; set; } = 1;
    public string Title { get; set; } = "";
    public string Name => _Name;

    public void SetFormat(MetricFormats format) {
        Format = format;
        switch (format) {
```

```
            case MetricFormats.None: { FormatString = "{0}"; Divider = 1;
break; }
            case MetricFormats.Numeric: { FormatString = "{0:#,0}"; Divider =
1; break; }
            case MetricFormats.Kb: { FormatString = "{0:#,0} Kb."; Divider =
1024; break; }
            case MetricFormats.Mb: { FormatString = "{0:#,0} Mb."; Divider =
1024 * 1024; break; }
            case MetricFormats.Gb: { FormatString = "{0:#,0} Gb."; Divider =
1024 * 1024 * 1024; break; }
        }
    }
    public void SetValue(Int128 newValue) {
        if (!_isFirst && !_Delta.Equals(newValue - _Value)) {
            MetricChangedEventArgs e = new(_Name, MetricValueTypes.Delta,
_Delta, newValue - _Value);
            _Delta = newValue - _Value;
            OnDeltaChanged(e);
        }
        if (!_Value.Equals(newValue)) {
            MetricChangedEventArgs e = new(_Name, MetricValueTypes.Value,
_Value, newValue);
            _Value = newValue;
            OnValueChanged(e);
        }
        _isFirst = false;
    }
    public void IncrementValue(Int128 value) {
        SetValue(_Value + value);
    }

}
```

Листинг 6 – Класс frmSystem

```
[DesignerCategory("Component"), SupportedOSPlatform("windows")]
public class frmSystem : Form {
    private tabSystem ts;
    private Button btnOk;
    private Button btnCancel;

    public frmSystem() {
        InitializeComponent();
        btnOk!.Click += (o,e) => Close();
        btnCancel!.Click += (o,e) => Close();
    }

    private readonly IContainer? components = null;
    protected override void Dispose(bool disposing) {
        if (disposing && (components != null)) { components.Dispose(); }
        base.Dispose(disposing);
    }
    private void InitializeComponent() {
        btnCancel = new Button();
        ts = new tabSystem();
        btnOk = new Button();
        SuspendLayout();
        //
        // btnCancel
        //
        btnCancel.Anchor = AnchorStyles.Bottom | AnchorStyles.Right;
        btnCancel.Location = new Point(371, 470);
```

```
        btnCancel.Name = "btnCancel";
        btnCancel.Size = new Size(75, 23);
        btnCancel.TabIndex = 2;
        btnCancel.Text = "Cancel";
        btnCancel.UseVisualStyleBackColor = true;
        //
        // ts
        //
        ts.Anchor = AnchorStyles.Top | AnchorStyles.Bottom |
AnchorStyles.Left | AnchorStyles.Right;
        ts.BorderStyle = Border3DStyle.RaisedInner;
        ts.Description = "System Properties";
        ts.Location = new Point(10, 10);
        ts.Name = "ts";
        ts.Padding = new Padding(2, 10, 10, 2);
        ts.Size = new Size(436, 452);
        ts.TabIndex = 0;
        ts.TabStop = false;
        ts.Title = "System";
        //
        // btnOk
        //
        btnOk.Anchor = AnchorStyles.Bottom | AnchorStyles.Right;
        btnOk.Location = new Point(290, 470);
        btnOk.Name = "btnOk";
        btnOk.Size = new Size(75, 23);
        btnOk.TabIndex = 1;
        btnOk.Text = "OK";
        btnOk.UseVisualStyleBackColor = true;
        //
        // frmSystem
        //
        AcceptButton = btnOk;
        CancelButton = btnCancel;
        ClientSize = new Size(454, 501);
        Controls.Add(ts);
        Controls.Add(btnOk);
        Controls.Add(btnCancel);
        FormBorderStyle = FormBorderStyle.FixedDialog;
        HelpButton = true;
        KeyPreview = true;
        MaximizeBox = false;
        MinimizeBox = false;
        Name = "frmSystem";
        SizeGripStyle = SizeGripStyle.Hide;
        StartPosition = FormStartPosition.CenterParent;
        Text = "System Properties - Feeling Nostalgic?";
        Load += OnLoad;
        ResumeLayout(false);
    }

    private void OnLoad(object? sender, EventArgs e) {
        ts.Refresher(true);
    }
}
```

Листинг 7 – Класс Shared

```
[SupportedOSPlatform("windows")]
internal static partial class Shared {
    private static string _SystemAccount = "";

    public static int bpi = 20; // Base PID Ignore
    public static List<string> skipProcess = new(new[] { "audiodg" });
    public static List<string> skipServices = new();
    public static string TotalProcessorsBin =
"".PadLeft(Environment.ProcessorCount, '1');
    public static string DebuggerCmd = "";
    public static TaskManagetETW ETW = new();

    public static bool IsNumeric(string value) => double.TryParse(value, out
_);
    public static bool IsNumeric(this object value) =>
double.TryParse(Convert.ToString(value), out _);
    public static bool IsInteger(string value) => value.All(char.IsNumber);
    public static bool IsInteger(this object value) =>
Convert.ToString(value)!.All(char.IsNumber);
    public static bool IsBetween<T>(this T value, T min, T max) where T :
IComparable<T> => (min.CompareTo(value) <= 0) && (value.CompareTo(max) <= 0);

    public static string ToTitleCase(string text) =>
CultureInfo.CurrentCulture.TextInfo.ToTitleCase(text.ToLower());
    public static string TimeSpanToElapsed(TimeSpan lpTimeSpan) {
        return string.Format("{0,3:D2}:{1,2:D2}:{2,2:D2}",
Convert.ToInt32(lpTimeSpan.Hours + (Math.Floor(lpTimeSpan.TotalDays) * 24)),
lpTimeSpan.Minutes, lpTimeSpan.Seconds);
    }
    public static string TimeDiff(long startTime, short Format = 1) {
        TimeSpan upX = new(DateTime.Now.Ticks - startTime);
        return Format switch {
            1 => string.Format("{0}d {1,2:D2}:{2,2:D2}:{3,2:D2}", upX.Days,
upX.Hours, upX.Minutes, upX.Seconds),
            2 => string.Format("{0}d {1,2:D2}h {2,2:D2}m", upX.Days,
upX.Hours, upX.Minutes),
            3 => string.Format("{0}d {1,2:D2}h {2,2:D2}m {3,2:D2}s",
upX.Days, upX.Hours, upX.Minutes, upX.Seconds),
            _ => "",
        };
    }

    public static void NotImplemented([CallerMemberName] string feature = "")
{
        MessageBox.Show("This feature is not implemented yet.", feature,
MessageBoxButtons.OK, MessageBoxIcon.Information);
    }
    public static void DebugTrap(Exception e, int Code = 0,
[CallerMemberName] string Method = "") {
        Debug.WriteLine($"*** Error at `{Method}` - Code: {Code} - Threw:
{e.Message}");
        Debug.WriteLine(e.ToString());
    }

    public static string GetSystemAccount() {
        if (_SystemAccount != "") return _SystemAccount;
        try {
            System.Security.Principal.SecurityIdentifier sid =
new(System.Security.Principal.WellKnownSidType.LocalSystemSid, null);
```

```
            _SystemAccount =
sid.Translate(typeof(System.Security.Principal.NTAccount)).ToString();
            if (_SystemAccount.IndexOf("\\") > 0) _SystemAccount =
_SystemAccount[(_SystemAccount.LastIndexOf("\\") + 1)..];
            _SystemAccount = ToTitleCase(_SystemAccount);
            Debug.WriteLine("Getting System Account... Result: " +
_SystemAccount);
        } catch (Exception ex) {
            DebugTrap(ex); _SystemAccount = "Error";
        }
        return _SystemAccount!;
    }
    public static void GetDebuggerCmd() {
        string SubKey = "Software\\Microsoft\\Windows
NT\\CurrentVersion\\AeDebug";
        string _Debugger = "";
        Microsoft.Win32.RegistryKey ParentKey =
Microsoft.Win32.Registry.LocalMachine;
        try {
            Microsoft.Win32.RegistryKey? Key = ParentKey.OpenSubKey(SubKey,
false);
            if (Key != null && Key.GetValue("Debugger") != null &&
!string.IsNullOrEmpty(Key.GetValue("Debugger")!.ToString())) {
                _Debugger = Key.GetValue("Debugger")!.ToString()!.Trim();
            }
            Key?.Close();
        } catch { } finally { ParentKey.Close(); }

        if (!string.IsNullOrEmpty(_Debugger)) {
            try {
                if (_Debugger.Contains('"')) {
                    _Debugger = _Debugger.TrimStart('\"');
                    _Debugger = _Debugger[.._Debugger.IndexOf('"')];
                    _Debugger = _Debugger.Trim('"');
                }
            } catch { _Debugger = ""; }
        }
        DebuggerCmd = _Debugger.Trim();
    }

    public static void BitClear(ref long MyByte, long MyBit) {
        long BitMask;
        BitMask = Convert.ToInt64(Math.Pow(2, MyBit - 1));
        MyByte &= ~BitMask;
    }
    public static void BitSet(ref long MyByte, long MyBit) {
        long BitMask;
        BitMask = Convert.ToInt64(Math.Pow(2, MyBit - 1));
        MyByte |= BitMask;
    }
    public static void BitToggle(ref long MyByte, long MyBit) {
        long BitMask;
        BitMask = Convert.ToInt64(Math.Pow(2, MyBit - 1));
        MyByte ^= BitMask;
    }
    public static bool BitExamine(long MyByte, long MyBit) {
        long BitMask;
        BitMask = Convert.ToInt64(Math.Pow(2, MyBit - 1));
        return (MyByte & BitMask) > 0;
    }

  }
```

ПРИЛОЖЕНИЕ Г
(обязательное)
Скриншоты работы программы

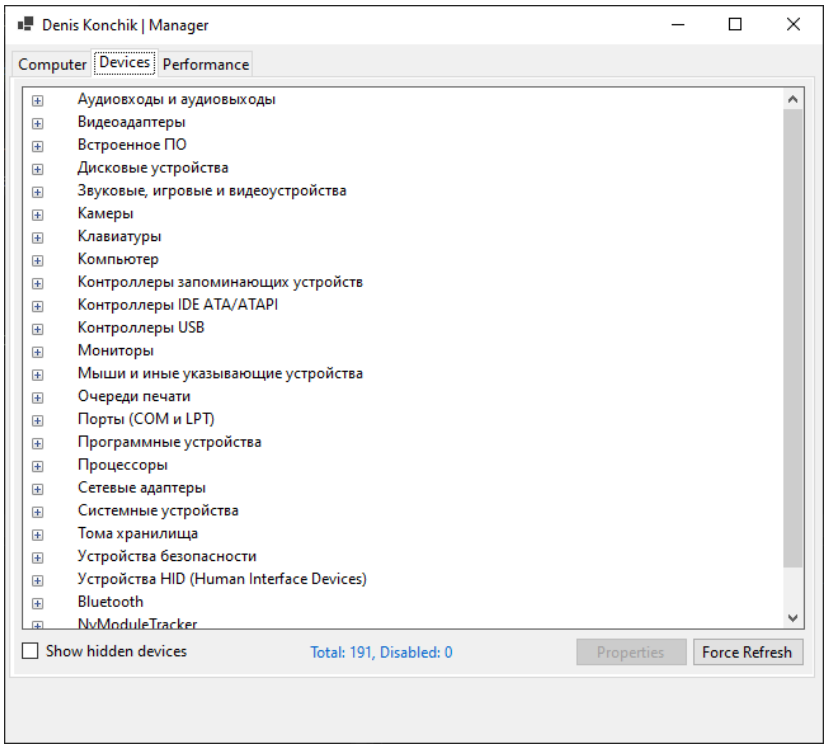

Рисунок 1 – Вкладка «Devices»

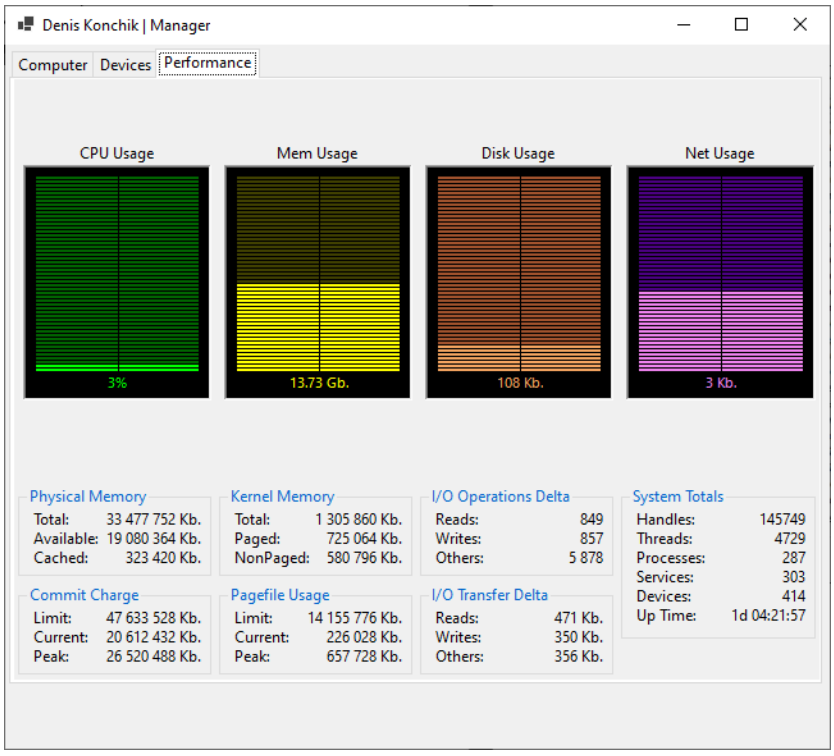

Рисунок 2 – Вкладка «Performance»

ПРИЛОЖЕНИЕ Д
(обязательное)
Ведомость документов