В.В.Штыков

FORTRAN & WIN32 API: СОЗДАНИЕ ПРОГРАММНОГО ИНТЕРФЕЙСА ДЛЯ WINDOWS СРЕДСТВАМИ СОВРЕМЕННОГО ФОРТРАНА

Пособие является практическим руководством по программированию в среде Windows на базе современного Фортрана. Основная цель книги — помочь читателю освоить приемы создания программного интерфейса. Это наиболее слабо освещенная тема в современной литературе по Фортрану. Практические навыки приобретаются в результате поэтапного создания приложения с разнообразными элементами управления. В конечном счете читатель получает в свое распоряжение каркас приложения.

Поскольку в книге фактически используется программирование на смеси языков, она будет полезна и для тех, кто программирует на языке Visial C++.

Предназначена для научно-технических работников, преподавателей, аспирантов и студентов вузов.

Содержание

Моим верным школьным товарищам –
Игорю, Виталию и Евгению –
посвящаю

Предисловие

Эта книга является прежде всего практическим руководством по программированию в среде Windows на базе современного Фортрана.

Windows – это очень сложная и большая система, и она не может быть полностью описана в одной книге. Поэтому ниже не рассматриваются теоретические аспекты этой системы, за исключением тех случаев, когда они непосредственно связаны с написанием программ. Что касается собственно программирования, то в книге обсуждаются только те его элементы, которые чаще всего используются при создании программ, либо те из них, которые являются существенными нововведениями.

Еще одно замечание. Основная цель книги – помочь читателю освоить приемы создания оконного интерфейса. Это наиболее слабо освещенная тема в современной литературе по Фортрану. Нет никакой возможности задерживаться на тонкостях программирования, кроме тех, которые непосредственно связаны с поставленной целью. Хорошим подспорьем по программированию на Фортране-90 является книга М. Меткалфа и Дж. Рида "Описание языка программирования Фортран-90" [2], а также книги О. В. Бартеньева, которые выпущены в свет издательством "Диалог - МИФИ" [4, 5, 6].

При создании программ использовалась реализация Фортрана ***Microsoft Fortran Power Station*** версии 4.0, которая основана на стандарте Фортран-90 и широко распространена в России. Это не исключает использования более современных модификаций, например ***Digital Fortran*** или ***Compag Visial Fortran 6.1a***.

Все исполнительные файлы создаются в форме проекта – приложения (***Aplication***) в среде – ***Microsoft Developer Studio*** (MDS). Однако можно воспользоваться и средствами фирмы COMPAG.

Поскольку в книге фактически используется программирование на смеси языков, изложенный ниже материал будет полезен и для тех, кто программирует на языке ***Visial C++***.

Я надеюсь, что эта книга поможет быстро освоить приемы программирования и вы скоро сможете создавать свои собственные приложения, внешний вид которых удовлетворит и вас, и ваших заказчиков.

В. Штыков

Москва, 2000 г.

УРОК

1. Краткий экскурс в Windows и современный Фортран

Прежде чем взяться за дело, нам следует хотя бы бегло ознакомиться с средой Windows и современным Фортраном.

Приступая к созданию приложений для Windows, важно понять, что это операционная система для работы на персональных компьютерах, построенная на принципах, совершенно отличных от тех, которые использовались ранее. Кроме того, Windows является 32-разрядной операционной системой. По этим причинам прикладные программы должны строиться иначе, чем в DOS.

Что касается Фортрана, то бытующее мнение о том, что Фортран устарел, ошибочно. Конечно, в связи с широким внедрением персональных ЭВМ во многие сферы человеческой деятельности удельный вес Фортрана в общем объеме программного обеспечения снизился. Однако при решении научно-технических задач, требующих большого объема вычислений, и особенно при использовании современных многопроцессорных высокопроизводительных вычислительных систем, Фортран по-прежнему занимает лидирующее положение.

В книге предлагаются практические подходы к программированию Windows-приложений. Это позволит сократить время разработки и отладки программ и упростить эти процедуры.

Первый урок не совсем выдержан в этом духе, поскольку, прежде чем приступить к написанию программы для Windows, вам необходимо хотя бы в общих чертах понять, как работает эта система, какие идеи в ней реализованы и как она управляет вашим компьютером. Важно также четко представлять, чем Windows отличается от предшествующих ей операционных систем и что в них общего.

Если вы до сих пор не писали программы для Windows, значительная часть представленного материала будет для вас новой. Однако если трудиться достаточно методично, то, дочитав книгу до конца, вы сможете создавать свои собственные приложения для Windows, используя для это-

го Фортран. Это позволит вам воспользоваться вашими предыдущими разработками и создать приложения, которые будут иметь вполне современный вид.

1.1. Обзор системы Windows

Начиная с 1995 г. система Windows (в дальнейшем также просто Система) – это многозадачная операционная система. Это означает, что она может одновременно выполнять две и более программы. Конечно, программы используют единственный процессор и отдельные части выполняются попеременно. Однако высокое быстродействие компьютера создает такую иллюзию одновременности.

Система Windows поддерживает два типа многозадачности. Первый тип – это так называемая *процессорная многозадачность*. Второй тип – *потоковая многозадачность*.

Процесс – это программа или приложение (в терминологии Windows), находящиеся в фазе выполнения. Процессная многозадачность заключается в том, что Система может выполнять одновременно более одной программы. Таким образом, Система поддерживает "традиционную" процессную многозадачность, с которой вы, вероятно, знакомы.

Поток – это отдельно выполняемая и управляемая часть программы. Название происходит от термина *поток выполнения*. Любой процесс имеет как минимум один поток. В Системе процесс может иметь несколько потоков.

Тот факт, что Система способна управлять потоками и каждый процесс может иметь несколько потоков, означает, что любой процесс может иметь две или более части, выполняющиеся одновременно. Следовательно, работая в Windows, можно одновременно выполнять как несколько программ, так и несколько частей отдельной программы.

Если вы писали программы для DOS, то знаете, что обращения к операционной системе осуществляются посредством различных программных прерываний, например через стандартное прерывание DOS #21. В то время как использование программных прерываний для доступа к функциям DOS вполне приемлемо, этот способ совсем не похож на тот, который применяется для реализации взаимодействия с такой многозадачной операционной системой, как Windows. Последняя использует интерфейс, базирующийся на вызовах функций.

Для доступа к системе интерфейс в Windows использует множество функций, определенных в ней. Это множество функций называется *про-*

граммным интерфейсом приложений (Application Program Interface – API). Интерфейс содержит несколько сотен функций, которые программа пользователя может вызывать для доступа к Системе. Функции включают все необходимые системно-зависимые действия, такие, как выделение памяти, вывод на экран, создание окон и т. п.

Поскольку API содержит несколько сотен функций, можно предположить, что каждая программа для Windows должна связываться с большим количеством библиотек и это может привести к дублированию большого объема кода. Однако это не так. Вместо обычных библиотек функции API объединены в *динамические библиотеки* (Dynamic Link Library – DLL), доступ к которым может получить любая программа во время выполнения. Функции API хранятся в перемещаемом формате в DLL. В процессе компиляции, когда программа вызывает функцию API, компоновщик не добавляет код этой функции к исполняемому модулю. Вместо него он добавляет только инструкции для загрузки функции, содержащие имя DLL, в которой находится функция, и имя самой функции. При выполнении программы все необходимые функции API также загружаются в память. Таким образом, при построении программы код функций API фактически не используется – он добавляется только тогда, когда программа загружается в память для выполнения.

Динамическое связывание имеет ряд важных преимуществ. Во-первых, поскольку практически все программы используют функции API, DLL сохраняет место на диске, не дублируя объектный код в выполняемых файлах. Во-вторых, дополнения и расширения Windows могут ограничиваться изменением программ в отдельных динамических библиотеках и существующие приложения не будут нуждаться в перекомпиляции.

Работая в Windows, вы сразу заметите новые управляющие элементы, появляющиеся на экране довольно часто: панель инструментов (toolbar), ввод с прокруткой (spin), окно просмотра деревьев (tree view) и линейку или окно состояния (status bar). Кроме того, Windows содержит так называемые *общие элементы управления* (common controls), которые можно использовать в любом приложении. Использование этих элементов придаст вашему приложению современный вид.

Система Windows предоставляет также некоторые интересные возможности, в том числе способность запуска DOS-программ как Windows-программ. Когда вы запускаете DOS-программу, автоматически создается отдельное командное окно DOS. Более того, это окно полностью входит в общий графический интерфейс Windows. Теперь можно запускать Windows-программы и прямо из командного окна DOS.

1.2. Современный Фортран

Язык Фортран представляет собой уникальное явление в области языков программирования. Появившись одним из первых и будучи вначале несовершенным (с современной точки зрения), он привлек к себе многочисленных пользователей.

Так сложилось исторически, что Фортран (FORTRAN – FORmula TRANslation) оказался основным языком программирования при решении научных, инженерных, а также многих других задач. Возникала и проходила мода на другие языки. Например, одновременно с Фортраном были разработаны языки, например Алгол, гораздо более выразительные синтаксически и обеспечивающие лучшую структурированность программ. Однако Фортран вытеснил все эти языки. Даже языки типа Pascal и C до сих пор не нашли широкого применения при решении числовых задач. Это объясняется рядом причин.

По-видимому, наиболее важной из них является то, что Фортран был разработан и в дальнейшем поддерживался огромной компьютерной фирмой International Business Machines, в результате чего он реализуется на всех наиболее популярных ЭВМ. Кроме того, для Фортрана характерен относительно примитивный синтаксис, что упрощает и одновременно повышает эффективность трансляции текста в машинный язык конкретной ЭВМ. Благодаря этому обеспечивается более высокая скорость обработки данных, что имеет все возрастающее значение для любого языка, используемого для научных расчетов. Устойчивость Фортрана обеспечивает его внедрение на всех известных больших и мини-ЭВМ, что обусловливает исключительно высокую степень переносимости языка. Большинство крупных научно-технических прикладных программ написано на Фортране именно потому, что он обладает переносимостью и устойчивостью.

Фортран постоянно развивается и совершенствуется в соответствии с развитием вычислительной техники, языков и технологии программирования. Одной из наиболее важных причин популярности и живучести Фортрана является огромный фонд прикладных программ, который накоплен за десятилетие существования языка. Этому в немалой степени способствовали удобство и эффективность выполнения программ, а также стандартизация языка.

Стандартизация языков программирования, и в частности Фортрана, обеспечивает мобильность программ при их переносе из одной вычислительной среды в другую, облегчает обмен программами, что приводит

к существенному уменьшению времени разработки и стоимости программного обеспечения

Язык Фортран дважды подвергался стандартизации в рамках Американского национального института стандартов (ANSI) и Международной организации по стандартизации (ISO) – в 1966-м и 1978 г. Вслед за публикацией второго стандарта Техническим комитетом ANSI (X3J3), ответственным за эту работу, состав комитета был сформирован заново, чтобы приступить к разработке третьего стандарта.

Намечалось подготовить новый стандарт к 1982 г., и будущему языку было присвоено рабочее название Фортран 8х. Но план оказался слишком оптимистичным – работу не удалось завершить в 80-х гг.

Дело в том, что стандартизация – это чрезвычайно формализованная и поэтому очень длительная процедура. Российскому читателю не безынтересно будет узнать, как появился новый стандарт языка Фортран.

Прежде всего в ходе процедуры стандартизации необходимо достигнуть большинства в две трети голосов в комитете X3J3, чтобы получить возможность направить предложение проекта стандарта для рассмотрения вышестоящему комитету по компьютерным системам X3. После того как комитет X3 дает согласие на рассмотрение, начинается период публичного обсуждения проекта, и от комитета X3J3 требуется, чтобы он учел поступившие замечания, принял соответствующие меры и направил доработанный проект в X3. Этот цикл может повторяться несколько раз.

К 1986 г. значительная часть пути к полному международному признанию нового стандарта была пройдена, но еще слышались довольно резкие возражения оппонентов, считавших, что в своих новациях проект зашел слишком далеко и что его надо сократить, чтобы он стал для них приемлемым.

Второе голосование состоялось в январе 1987 г. На этот раз было продемонстрировано большее единодушие, хотя истинного консенсуса по-прежнему не было, поскольку среди голосовавших "против" были крупные фирмы, утверждавшие, что они представляют интересы своих потребителей.

В сентябре 1988 г. рабочая группа Международной организации по стандартизации, состоящая из международных экспертов, определила, какие именно изменения требуется внести в исходный проект стандарта, и установила сроки подготовки второго проекта, отвечающего этим требованиям. Более того, она отважилась переименовать предлагаемый язык в Фортран-88.

Осенью 1989 г. второй проект предлагаемого стандарта стал предметом публичного обсуждения как в Америке, так и в международных кругах. Было получено 149 отзывов, причем многие из них были благожелательными. В результате обсуждения во второй проект стандарта были внесены незначительные изменения. Наконец после очередного обсуждения в США в августе 1991 г. он был официально опубликован в качестве международного стандарта. Таким образом, после многих лет обсуждений и согласований появился новый стандарт языка Фортран. Этот стандарт получил название *Фортран-90*.

Фортран-90 существенно расширяет возможности своих предшественников. В то же время практически сохраняется преемственность с предыдущими стандартами языка, что позволяет использовать ранее созданный фонд прикладного программного обеспечения. Фортран-90 позволяет создавать более мобильные и более надежные программы по сравнению с Фортраном-77 и обеспечивает современный стиль программирования.

Наиболее важные его новые черты: свободный формат текста исходной программы, производные типы данных (структуры данных), средства параметризации встроенных типов данных, операции над массивами и сечениями массивов, указатели, механизмы динамического размещения объектов в памяти, новые средства для описания и использования глобальных объектов, гибкие управляющие конструкции, определяемые пользователем операции, более развитые средства использования процедур (внутренние процедуры, рекурсии, возможность специфицировать аргументы по назначению, возможность использования необязательных и ключевых параметров при вызове процедур), а также большой набор новых стандартных процедур и т. п.

Возможность описания и использования производных типов и указателей позволяет специфицировать сложные структуры данных: списки, деревья, графы и т. п. Новый вид программных единиц (программные единицы-модули), которые предназначены для спецификации глобальных объектов, вместе с возможностью описания новых типов данных и операций являются мощным средством расширяемости языка, выражением концепции абстрактных типов данных. Эти средства позволяют создавать пакеты (модули), ориентированные на конкретные приложения.

Операции над массивами и сечениями массивов задают параллелизм действий над компонентами массивов (или массива). Такие средства, с одной стороны, позволяют пользователю лаконично и сжато описать алгоритм обработки массивов и, с другой стороны, дают возможность компилятору генерировать более эффективный код с учетом особенностей кон-

кретной ЭВМ. Очевидно, что наиболее эффективно использование этих возможностей для вычислительных систем, имеющих аппаратные средства для векторной обработки.

Параметризация встроенных типов дает возможность компиляторам поддерживать короткие целые, более двух видов точности для вещественных и комплексных данных, а также многобайтовые символьные данные (для языков с большим набором символов, таких, как китайский или японский) или использовать дополнительные наборы символов для конкретных приложений.

Введенные в язык механизмы динамического размещения массивов дают возможность пользователю организовать работу с массивами, размер которых заранее неизвестен и вычисляется в процессе выполнения программы, а также оперативно отводить память под массивы, требующиеся на том или ином этапе выполнения программы.

Представляет интерес принятая концепция эволюционного развития языка. Некоторые конструкции языка Фортран в настоящее время устарели и стали излишними после введения новых элементов; такие конструкции отнесены к категории устаревших черт. Наличие в языке устаревших черт усложняет язык и компилятор, что негативно отражается на эффективности создания программ. Кроме того, некоторые устаревшие элементы ненадежны, а другие – снижают мобильность. Многие из этих черт противоречат концепции структурного программирования и препятствуют распараллеливанию программ.

Новый стандарт предъявляет более строгие требования к компиляторам. Компилятор должен выявлять и сообщать пользователю не только конструкции, которые не отвечают требованиям языка, но также и устаревшие конструкции и расширения языка.

В настоящее время Фортран-90 реализован практически для всех современных компьютеров от персональных ЭВМ до больших параллельных вычислительных систем. Для некоторых компьютеров имеется более одного компилятора, больше всего реализации для персональных компьютеров. Современные компиляторы, как правило, снабжены хорошими сервисными возможностями.

Созданы и другие программные продукты. Для того чтобы эффективно использовать ранее написанные на Фортране-77 программы, разработаны трансляторы, переводящие программы с Фортрана-77 на Фортран-90. Такие трансляторы заменяют устаревшие конструкции на новые, введенные в Фортран-90, а также те конструкции, для которых имеются лучшие решения в Фортране-90.

Разработаны библиотеки численных методов на Фортране-90 (NAG, IMSL). Помимо официального описания стандарта языка, выпущены десятки книг на разных языках, которые содержат неформальное описание Фортрана-90. В ряде университетов читаются лекции по Фортрану-90, созданы разнообразные курсы по изучению этого языка.

Развитие Фортрана продолжается и сейчас; активно ведется разработка проекта будущего стандарта, рабочее название которого *Фортран-2000*.

Заслуживает внимания разработка и реализация подмножества Фортрана-90, получившего название *язык F*. Подмножество содержит все современные элементы Фортрана-90 и не содержит устаревших конструкций (сохранившихся в Фортране-90 для преемственности), поскольку для них имеются более мощные средства в языке. Удаление таких конструкций, а также введение дополнительных ограничений позволило сделать язык F более компактным и элегантным по сравнению с Фортраном-90 и обеспечить хороший стиль программирования. Язык F предназначен как для начинающих, так и для профессионалов. Другой вариант подмножества, основанного на аналогичном подходе, – Elf 90, разработанный и реализованный известной фирмой по разработке Фортран-компиляторов Lahey.

На базе Фортрана-90 разработан язык HPF (High Performance Fortran), который предназначен для написания параллельных программ для многопроцессорных вычислительных комплексов. Язык является расширением Фортрана-90 и существенно использует его новые возможности. HPF содержит директивы для описания способов разбиения больших массивов данных между параллельно работающими процессорами, а также операторы и директивы для явного указания параллельности.

УРОК

2. Основные принципы программирования

Настоящий урок является введением в программирование для Windows и преследует две основные цели. Во-первых, дать представление о том, как приложение должно взаимодействовать со средой Windows и какие правила должны соблюдаться каждым приложением Windows. Во-вторых, мы познакомимся со структурой приложения. Эта структура – каркас приложения – в дальнейшем будет использоваться как основа для разработки других программ для Windows, как приведенных в этой книге, так и собственных. Как вы увидите, все программы имеют некоторые общие черты, те самые, которые будут включены в каркас приложения.

Начнем с общего взгляда на программирование для Windows.

2.1. Общий взгляд на программирование для Windows

Целью Windows является предоставление любому пользователю, имеющему минимальные знания о системе, возможности сесть и запустить практически любое приложение без предварительного обучения. Эта цель достигается посредством согласованного пользовательского интерфейса. Теоретически если вы можете запустить хотя бы одно Windows-приложение и работать с ним, то сможете работать и со всеми остальными программами. В действительности же, чтобы эффективно использовать каждую программу, некоторое обучение все же необходимо, но, по крайней мере, это обучение будет касаться того, что делает программа, а не того, как к ней обращаться. Пользовательский интерфейс – это большая часть программного кода приложения.

Прежде чем продолжить, нужно заметить, что не всякая программа, созданная для Windows, будет иметь Windows-интерфейс. Можно написать Windows-программу, не использующую элементы интерфейса

Windows. Для того чтобы написать программу в стиле Windows, вы должны делать это целенаправленно, применяя приемы, описанные в данной книге. Только те программы, которые используют возможности интерфейса Windows, будут выглядеть и вести себя как Windows-программы.

Теперь рассмотрим вкратце основные элементы интерфейса Windows.

- Модель "рабочего стола"

За некоторыми исключениями главная особенность оконного пользовательского интерфейса состоит в том, что он обеспечивает на экране модель "рабочего стола". На обычном письменном столе, как правило, разбросаны различные листы бумаги, одни поверх других, содержащие разные документы или их части. Эквивалентом рабочего стола в Windows является экран, эквивалентами листов бумаги – окна на экране. На столе вы можете перемещать и перекладывать листы бумаги, и то же самое можно проделывать с окнами. Выбирая какое-либо окно, вы делаете его активным и помещаете поверх всех остальных; вы также можете изменять размеры окон и перемещать их в пределах экрана. Короче говоря, Windows позволяет управлять экраном так же, как вы "управляете" с предметами на рабочем столе.

Хотя модель "рабочего стола" является основой пользовательского интерфейса Windows, программа ею не ограничивается. Некоторые элементы интерфейса Windows, такие, как *линейка прокрутки (scroll bar), ввод с прокруткой (up-down, spin), окна просмотра деревьев (tree view) и панели инструментов (toolbar)*, предоставляют программисту большой выбор возможностей, которые вы можете использовать в своих программах.

- Мышь

Во всех версиях Windows мышь используется для большинства операций управления, выбора и рисования. Интерфейс Windows фактически создан для мыши, хотя позволяет использовать и клавиатуру! Вообще говоря, приложение может игнорировать мышь, но такое поведение программы будет нарушать основные принципы функционирования Windows.

- Компоненты окна

Окно является основополагающим элементом Windows. Абсолютно все события вашей программы будут развиваться на элементах, которые рассматриваются как окна определенного стиля и вида.

Стандартные окна имеют рамку, которая определяет их границы; рамка используется также для изменения размеров окна. В верхнем левом углу окна находится *иконка системного меню* (называемая также заголовочной иконкой). Щелчок мыши на этой иконке открывает системное меню.

Справа от иконки системного меню находится заголовок окна. В правом верхнем углу окна расположены *кнопки минимизации, полноэкранной развертки* (полноэкранного представления) и *закрытия окна. Рабочая область (client area)* – это часть окна, в которой отображается действие программы. Многие окна имеют также *вертикальные* и *горизонтальные линейки прокрутки (scroll bars)*.

- Иконки и растровые рисунки

Windows ориентирован на использование *иконок* и *растровых рисунков* (графических образов, bitmaps). Дело здесь в том, что, как известно, "лучше один раз увидеть, чем сто раз услышать".

Иконка – это небольшой символ, представляющий на экране некоторую операцию или программу. В общем случае операция или программа может быть активизирована щелчком мыши на иконке.

Растровые рисунки являются быстрым и простым средством отображения информации. Рисунки могут также использоваться и в качестве элементов меню.

- Меню, панели инструментов, панели состояния и диалоги

Кроме стандартных средств Windows предоставляет несколько специальных типов окон. Наиболее часто используемыми из них являются *меню, панель инструментов (toolbar), линейка состояния (status bar)* и *диалог*.

Меню – это специальное прямоугольное окно, которое содержит доступные операции и позволяет пользователю выбирать нужные ему элементы. В программе не нужно описывать функции управления меню, достаточно просто создать стандартное меню, используя встроенные функции управления.

Панель инструментов (toolbar) – это специальный тип меню, который отображает операции меню в виде небольших графических образов (иконок) и предоставляет пользователю возможность быстрого доступа ко многим командам и опциям. Пользователь выбирает нужный объект щелчком мыши на соответствующей иконке.

Линейка (окно) состояния (*status bar*), как правило, размещается в нижней части окна и отображает информацию о состоянии приложения.

Диалог – специальное окно, обеспечивающее более сложный по сравнению с меню и панелью инструментов интерфейс. За исключением отдельных случаев, практически весь ввод информации производится через диалоги. Система поддерживает некоторый минимальный набор стандартных диалогов. Например, через стандартный диалог ваше приложение может запросить имя файла перед его загрузкой. Однако основная часть диа-

логов – это диалоги, которые создаются для конкретного приложения программистом.

2.2. Взаимодействие Windows с программой

Во многих операционных системах взаимодействие с системой инициируется программой пользователя. Например, в DOS она обращается к операционной системе при необходимости осуществить ввод/вывод данных. Таким образом, программы, написанные в традиционном стиле, сами обращаются к операционной системе. Однако Windows работает не так. Именно Windows обращается к вашей программе.

Этот процесс выглядит следующим образом. Программа ожидает сообщения, посылаемого ей Windows. Сообщение передается в программу посредством вызова специальной функции, который также выполняется из Windows. После получения сообщения программа может выполнять некоторые действия. Эти действия могут включать вызов одной или нескольких функций API, и именно Windows инициирует их. Такое базирующееся на сообщениях взаимодействие программы и операционной системы более чем что-либо другое определяет схему построения всех программ для Windows.

В Windows существует множество различных типов сообщений. Например, каждый раз при щелчке мыши в пределах окна программы ей будет направлено сообщение о нажатии кнопки мыши. Сообщение другого типа посылается программе всякий раз, когда принадлежащее ей окно должно быть перерисовано. Сообщения иного типа приходят, если нажата клавиша на клавиатуре в то время, когда ваше окно активно или, как говорят, имеет фокус ввода (т. е. ввод с клавиатуры направляется вашему окну). Твердо запомните: когда программа начинает работать, сообщения к ней приходят неупорядоченным образом, т. е. вы никогда не знаете, какое сообщение будет следующим. Таким образом, программы для Windows напоминают программы, работающие по прерываниям.

Подобная организация общения программ с Системой предоставляет пользователю массу удобств. Однако за дополнительные услуги приходится расплачиваться скоростью исполнения программы.

2.3. Win32 API: прикладной интерфейс для Windows

Каждая программа может получить доступ к ресурсам Windows, используя множество функций API, состоящего из нескольких сотен функций, которые при необходимости могут вызываться из программы. Функции API обеспечивают доступ ко всем ресурсам Windows. Подмножество API, называемое *интерфейсом графических устройств* (Graphics Device Interface – GDI), обеспечивает поддержку независимой от оборудования графики. Именно функции GDI делают возможной работу Windows на весьма разнообразном оборудовании.

Программы для Windows-95 и более поздних версий используют Win32 API, в котором поддерживается 32-разрядная адресация. В эту версию включены также функции для реализации новых подходов к многозадачности, новых элементов интерфейса и других новых возможностей Windows.

2.4. Базовые элементы и понятия

Прежде чем начать разработку каркаса приложения для Windows, необходимо рассмотреть некоторые понятия, общие для всех программ Windows.

Приложения Windows начинают свое выполнение с вызова WinMain(), которая имеет некоторые свойства, отличающие эту функцию от остальных функций приложения.

Как уже говорилось выше, при создании приложения на Фортране используется программирование на смеси языков. По умолчанию все функции в Visual C++ используют соглашения о вызовах, принятые для C-программ. Допускается также применение функций, вызываемых иначе; в случае WinMain() используется тип соглашения о вызовах, принятый в среде языка Pascal. Значение, возвращаемое WinMain(), должно иметь тип int. Поэтому для того, чтобы система смогла общаться с вашим приложением, эту функцию в Фортране надо объявить следующим образом:

```
integer(4) function WinMain(hInstance, hPrevInstance, lpCmdLine, nShowCmd)
!MS$ATTRIBUTES STDCALL, ALIAS : '_WinMain@16' :: WinMain
```

- Оконная функция

Все программы для Windows должны содержать некоторую функцию, которая вызывается не из вашей программы, а операционной системой. Эту функцию часто называют *оконной функцией* или *оконной процедурой*.

Оконная функция вызывается операционной системой Windows, когда программе нужно послать сообщение. Используя эту функцию, Windows взаимодействует с вашей программой. В качестве параметров оконная функция получает сообщение, посылаемое Windows. По тем же причинам, что и выше, функция должна быть объявлена следующим образом:

```
integer(4) function MainWndProc(hwnd, message, wParam, lParam)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_MainWndProc@16' :: MainWndProc
```

Принимая сообщения, посылаемые Windows, оконная функция должна выполнять действия, соответствующие типу и параметрам этих сообщений. Как правило, тело оконной функции представляет собой оператор **select case**, определяющий тип получаемых сообщений и выполняющий соответствующие действия для каждого типа. Ваша программа не обязана выполнять обработку всех типов сообщений, посылаемых Windows. Сообщения, не обрабатываемые в вашей программе, должны обрабатываться Windows по умолчанию. Поскольку в Windows существуют сотни типов сообщений, большая их часть, как правило, обрабатывается по умолчанию.

Все сообщения являются 32-битовыми величинами. Кроме того, все они несут дополнительную информацию, свойственную каждому типу сообщений.

- Классы (стили) окон

Когда ваша программа, написанная для Windows, начинает выполняться впервые, ей необходимо определить и зарегистрировать *класс окна*. Здесь слово *класс* употребляется не в смысле определения из C++. Скорее оно должно означать стиль или тип. Регистрируя класс окна, вы сообщаете Windows об основных атрибутах и функциях окна. Однако регистрация класса не означает создания окна. Для этого требуются дополнительные действия.

- Цикл обработки сообщений

Как отмечалось ранее, Windows взаимодействует с вашей программой, посылая ей сообщения. Все приложения Windows должны запускать *цикл обработки сообщений* в функции **WinMain()**. Этот цикл выбирает сообщения из очереди сообщений приложения и направляет их обратно к Windows, которая затем вызывает оконную функцию вашей программы, передавая ей указанные сообщения в качестве параметров. Такой способ взаимодействия может показаться излишне сложным, однако программы для Windows функционируют именно так.

- Типы данных в Windows

Программы для Windows-95 редко используют стандартные типы данных. При программировании на языке C++ вместо стандартных типов используются типы данных, определенные при помощи оператора **typedef** в файле **windows.h** и/или сопутствующих файлах. Файл **windows.h** поставляется Microsoft (или любой другой компанией, выпускающей компиляторы Visual C++ для Windows) и должен включаться во все программы для Windows. Вот некоторые из этих типов: HANDLE, HWND, BYTE, WORD, DWORD, UINT, LONG, BOOL, LPSTR, LPCSTR. HANDLE представляет собой 32-разрядное целое число, используемое как дескриптор. Существует множество типов дескрипторов, но все они имеют ту же размерность, что и HANDLE. *Дескриптор* – это просто число, идентифицирующее некоторый ресурс. Все типы дескрипторов начинаются с буквы *H*. Например, HWND – это 32-разрядное целое, используемое как дескриптор окна. BYTE – 8-разрядное значение без знака; WORD – 16-разрядное короткое целое без знака; DWORD – длинное целое без знака; UINT – 32-разрядное беззнаковое целое; LONG – просто другое название для long. BOOL – это целое число, принимающее значения "истина" и "ложь"; LPSTR – указатель на строку, а LPCSTR – константа, указывающая на строку.

Для всех этих типов данных в Фортране существуют аналоги типа **integer(), logical(), byte**. Некоторую осторожность следует проявлять при установлении эквивалентности данных без знака. Конечно, можно было бы определить производные типы именами, которые согласованы с типами данных Visual C++, однако значительные затраты времени вряд ли принесут ощутимый результат.

Кроме собственных базовых типов данных Windows определяет несколько структур. Для разработки каркаса приложения нам понадобятся структуры MSG и WNDCLASS. Структура MSG описывает сообщение, а WNDCLASS – класс окна. Содержимое этих структур рассмотрим позже. При необходимости аналоги этих структур могут быть созданы в Фортране с помощью определения производных типов операторами.

УРОК

3. Создаем первое приложение

Теперь, когда мы обсудили основные понятия, можно начать разработку простейшего приложения для Windows.

Как уже отмечалось выше, все программы для Windows имеют некоторые общие черты. Это позволяет разработать некоторый шаблон или каркас, пригодный для создания других приложений. Вообще технология написания программ для Windows предполагает использование таких шаблонов. Это объясняется тем, что пять строк простейшей программы для DOS превращаются в 50 в Windows.

Именно по указанной причине созданы специальные средства для автоматической генерации кода приложений для Windows. Такие средства, например, включены в состав компиляторов Visial C++. В первую очередь к ним относится ***MFC AppWizard*** [3]. Аналогичные средства для Fortran-90 мне неизвестны.

Однако надо иметь в виду, что никакие автоматизированные средства разработки не смогут создать что-либо без вашего участия. Причина очевидна – проектирование, разработка, конструирование – это творчество. Результат труда человека несет на себе печать творца.

В дальнейшем мы предполагаем использовать среду ***Microsoft Developer Studio*** (MDS) для создания приложений на Фортране. В состав MDS включен редактор ресурсов, который значительно упростит процедуру создания программ. Исполняемый модуль будем создавать в форме проекта ***Application***.

3.1. Создание проекта в среде Microsoft Developer Studio

Для создания исполняемого файла нашего первого приложения используем средства MDS. Для этого выполним следующую цепочку действий: ***File – New – Project Workspace – Application – зададим имя проекта и его***

размещение – ***Create***. Таким образом будет создан новый проект. Если проект уже существует, то его следует открыть, выполняя цепочку действий ***File – Open Workspace – выбрать файл проекта – Open***. Минимально необходимые сведения о работе с проектом можно найти, например, в [6, 3].

Следующий шаг – это включение в проект файлов с текстами отдельных программных единиц. Для включения уже существующего текстового файла необходимо выполнить следующие действия: ***Insert – File Into Project – выбрать файл или ввести имя нового – Add***.

Для нашего первого приложения включим в проект файл **MyPr_1.f90**, который будет содержать головную программу, и файл – **MyPr_1b.f90** с текстом функции, которая будет обрабатывать сообщения. Учитывая особенности программирования на современном Фортране, добавим файл **MyPr_1inc.f90**, который будет содержать модуль с глобальными данными. Этот файл будет использоваться другими программными единицами через оператор **use**.

Теперь можно открыть окно ***Project Workspace*** и просмотреть (***File View***) список ***имен*** файлов, включенных в проект. Двойной щелчок мыши открывает файл. Если файла с выбранным именем нет, то появляется соответствующий запрос и после утвердительного ответа такой файл создается на диске, а на экране появляется чистое окно.

3.2. Каркас приложения

Простейшая программа, которая способна всего лишь открыть окно, использует две функции – **WinMain()** и оконную функцию, которую мы назовем **MainWndProc()**.

Функция **WinMain()** выполняет следующие действия:

1. Определяет класс окна.
2. Регистрирует класс.
3. Создает окно.
4. Отображает окно.
5. Запускает цикл обработки сообщений.

Оконная функция **MainWndProc()** должна обрабатывать все сообщения, относящиеся к данному окну.

Включим функцию **WinMain()** в файл **MyPr_1.f90**. Для того чтобы придать тексту программы более прозрачный вид, выделим две внутренние процедуры: **InitApp()** и **InitInst()**. Первая выполняет операции 1 и 2, а вторая – 3 и 4. В результате получим следующий текст:

```
!*****************************************************************************
!*                    WinMain
!*  Цель: вызвать функции инициализации и
!*        организовать цикл обработки сообщений
!*****************************************************************************
integer(4) function WinMain(hInstance, hPrevInstance, lpCmdLine,  nShowCmd)
!MS$ATTRIBUTES STDCALL, ALIAS : '_WinMain@16' :: WinMain
!_____________________________________________________________________________
use MyPr_1inc

interface
integer(4) function InitApp()
end function InitApp
end interface

interface
integer(4) function InitInst(nShowCmd)
 integer(4)  :: nShowCmd
end function InitInst
end interface

type (T_MSG)  :: mesg

integer(4)        :: hInstance, hPrevInstance, lpCmdLine, nShowCmd

hInst             = hInstance
lpCmdLine         = lpCmdLine
hPrevInstance     = hPrevInstance
!-------------- Инициализируем приложение ------------
if (InitApp() == 0) then
 WinMain = 0
 return
end if

!----- Инициализируем данный экземпляр приложения ------
if (InitInst(nShowCmd) == 0) then
 WinMain = 0
  return
end if

!---------  Обработка сообщений -----------------------
do while (GetMessage(mesg, NULL, 0, 0))
   ir = TranslateMessage(mesg)
   ir = DispatchMessage(mesg)
end do
```

```
!-----------------------------------------------
WinMain = mesg%wParam
end
!**************************************************************************
```

Обращаю ваше внимание на вторую строку:

```
!MS$ATTRIBUTES STDCALL, ALIAS : '_WinMain@16' :: WinMain.
```

В Фортране любая строка, содержащая символ $ или начинающаяся с символов !MS$, интерпретируется как метакоманда. Метакоманда – это инструкция для компилятора. Инструкция ATTRIBUTES объявляет атрибуты процедур и переменных. Ее появление связано с тем, что по сути мы с вами используем два языка программирования. Атрибут STDCALL устанавливает правила передачи данных, а атрибут ALIAS задает внешнее имя подпрограммы. В приложениях, написанных на Фортране, все подпрограммы, которые общаются с Системой, должны содержать инструкцию ATTRIBUTES.

Выполнение программы начинается с **WinMain()**, которой передаются четыре параметра: **hInstance** – дескриптор текущего экземпляра приложения, **hPrevInstance** – то же для предыдущего экземпляра, **lpCmdLine** – указатель на командную строку с аргументами, **nShowCmd** – способ отображения окна. Второй параметр получен в наследство от Windows 3.1. Поскольку начиная с Windows-95 система стала многозадачной, этот параметр всегда будет равен нулю.

Класс окна определяется при выполнении функции **InitApp()**. Для этого заполняются поля структуры типа T_WNDCLASS. Эта структура определена в файле **MsfWinty.f90** и имеет следующие поля:

```
integer   style          – тип окна,
integer   lpfnWndProc    – адрес оконной функции,
integer   cbClsExtra     – дополнительные данные для класса,
integer   cbWndExtra     – дополнительные данные для окна,
integer   hInstance      – дескриптор данного экземпляра приложения,
integer   hIcon          – дескриптор иконки для данного окна,
integer   hCursor        – дескриптор курсора для данного окна,
integer   hbrBackground  – цвет заполнения окна,
integer   lpszMenuName   – имя главного меню,
integer   lpszClassName  – имя класса окна.
```

Как видно из приведенного ниже текста программы, поле **hInstance** принимает значение параметра **hInst**, имя класса окна **lpszClassName** получает адрес строки SZWINNAME, а адрес оконной функции присваивает-

ся переменной **lpfhWndProc**. Тип окна по умолчанию равен нулю, и дополнительные данные для класса и окна не требуются.

Все приложения должны задавать форму курсора и иконку. Проще всего использовать для этого стандартные предопределенные объекты. Некоторые из них приведены в табл. 3.1. Таблица содержит также численные значения идентификаторов. Это иногда помогает при отладке приложения.

Таблица 3.1. Предопределенные типы иконок

Идентификатор	*Значение*	*Тип иконки*
IDI_APPLICATION	#00007F00	Стандартная иконка для приложения
IDI_ASTERISK	#00007F04	Иконка "информация"
IDI_EXCLAMATION	#00007F03	Иконка "восклицательный знак"
IDI_HAND	#00007F01	Иконка "стоп"
IDI_QUESTION	#00007F02	Иконка "вопросительный знак"
IDI_WINLOGO	#00007F05	Логотип Windows

Последний тип не определен в файле **msfwinty.f90**. Поэтому его значение надо каким-либо образом ввести в текст нашей программы, если вы намереваетесь использовать этот логотип.

Иконку определяем при помощи функции API **LoadIcon (hIns, lpIconName)**. Эта функция возвращает дескриптор иконки. Смысл параметров функции очевиден. Для того чтобы использовать стандартные иконки, первый из них должен быть равен NULL.

С курсором поступаем аналогичным образом. Используем функцию API **LoadCursor (hIns, lpCursorName).** Наиболее употребительные типы встроенных курсоров приведены в табл. 3.2.

Таблица 3.2. Предопределенные типы курсоров

Идентификатор	*Значение*	*Тип курсора*
IDC_APPSTARTING	#00007F8A	Стандартная стрелка и песочные часы
IDC_ARROW	#00007F00	Стандартная стрелка
IDC_CROSS	#00007F03	Перекрестие
IDC_IBEAM	#00007F01	Вертикальная черта("каретка")
IDC_NO	#00007F88	Перечеркнутая окружность
IDC_UPARROW	#00007F04	Вертикальная стрелка
IDC_WAIT	#00007F02	Песочные часы

В поле **hbrBackground** задается цвет заполнения окна, создаваемого в нашей программе. Для этого существует два способа. Можно использовать функцию API **GetStockObject()** для того, чтобы получить дескриптор определенной кисти – ресурса, задать цвет и способ заполнения объектов на экране. Если надобность в объекте отпала, то функцию API **DeleteObject()** можно не вызывать, однако ее присутствие в тексте программы будет лишним.

Несколько типов системных кистей приведено в табл. 3.3.

Таблица 3.3. Системные кисти

Идентификатор	*Значение*	*Тип кисти*
BLACK_BRUSH	4	Черная кисть
DKGRAY_BRUSH	3	Темно-серая кисть
GRAY_BRUSH	2	Серая кисть
LTGRAY_BRUSH	1	Светло-серая кисть
NULL_BRUSH	5	"Нулевая" кисть
WHITE_BRUSH	0	Белая кисть

Вторая возможность – присвоить цветовое значение параметру **hbrBackground**. Цветовое значение должно быть равно одному из стандартных цветов, **увеличенному на 1**. Общее число цветов велико. Вот лишь некоторые из них: COLOR_ MENU = 4, COLOR_WINDOW = 5, COLOR_WINDOWFRAME = 6, COLOR_WINDOWTEXT = 8, COLOR_CAPTIONTEXT = 9, COLOR_ ACTIVEBORDER = 10, COLOR_APPWORKSPACE = 12, COLOR_ GRAYTEXT = 17. Как видите, каждое цветовое значение связано с каким-либо элементом окна, а не с конкретным цветом. Цвета элементов устанавливаются при настройке системы.

Вы можете самостоятельно поэкспериментировать с цветовыми значениями. Окончательный вариант текста процедуры приведен ниже.

```
!************************************************************
!                    InitApp
!  Цель: Инициализация данных и регистрация класса окна
!************************************************************
integer(4) function InitApp()
!____________________________________________________________
use MyPr_1inc

interface
```

```
integer*4 function MainWndProc  (hwnd, message, wParam, lParam)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_MainWndProc@16' :: MainWndProc
 integer(4) :: hwnd, message, wParam, lParam
end function MainWndProc
end interface

type (T_WNDCLASS)  :: wc

wc%style                = 0                                  ! стиль по умолчанию
wc%lpfnWndProc          = LOC(MainWndProc)                   ! функция окна
wc%cbClsExtra           = 0                                  ! без дополнительной
wc%cbWndExtra           = 0                                  ! информации
wc%hInstance            = hInst              ! дескриптор данного приложения
wc%hIcon        = LoadIcon(NULL, IDI_WINLOGO)                ! логотип WIN95
wc%hCursor      = LoadCursor(NULL, IDC_ARROW)                ! стандартный курсор
wc%hbrBackground        = COLOR_APPWORKSPACE+1 ! стандартный цвет
wc%lpszMenuName         = NULL                               ! без меню
wc%lpszClassName        = LOC(SZWINNAME)                     ! имя класса окна

InitApp = RegisterClass(wc)
end
!*****************************************************************************
```

3.3. Создание окна

Следующим нашим шагом является создание окна. Для этого в процедуре InitInst() используется функция **CreateWindow (lpszClassName, lpszWindowName, dwStyle, x, y, nWidth, nHeight, hWndParent, hMenu, hInstance, lpParam)**.

Первый параметр, **lpszClassName** – это так называемая С-строка с именем класса, следующий, **lpszWindowName** – С-строка с именем окна*. Далее следуют:

dwStyle	– стиль окна,
x, y	– координаты верхнего левого угла,
nWidth. nHeight	– размеры окна,
hWndParent	– дескриптор родительского окна,
hMenu	– дескриптор главного меню,
hInstance	– дескриптор приложения,
lpParam	– указатель на структуру с дополнительной информацией.

---

* *Строковые переменные в языках С и С++ всегда заканчиваются нулевым символом. В Фортране это не так. Поскольку мы используем оконный интерфейс Visial C++, надо всегда это иметь в виду при программировании на смеси языков.*

Все эти параметры имеют тип **integer(4)**.

Перечень стилей окна приведен в табл. 3.4. Их обилие повергает в уныние, тем более что пока большинство из этих "заклинаний" ни о чем не говорит. Расстраиваться все же нет оснований, т. к., во-первых, часть стилей дублирует друг друга, а, во-вторых, значение многих стилей станет ясным в дальнейшем.

Таблица 3.4. Стили окна

Стиль	*Значение*	*Описание*
WS_OVERLAPPED	#00000000	Окно имеет заголовок и обрамляющую рамку
WS_POPUP	#80000000	Создается всплывающее окно
WS_CHILD	#40000000	Создается дочернее окно, имеющее по умолчанию только рабочую область
WS_MINIMIZE	#20000000	Создается свернутое окно
WS_VISIBLE	#10000000	Окно отображается
WS_DISABLED	#08000000	Окно недоступно для пользователя
WS_CLIPSIBLINGS	#04000000	Создается дочернее окно, у которого сохраняется область, перекрытая другим дочерним окном
WS_CLIPCHILDREN	#02000000	При прорисовке родительского окна область, занимаемая дочерним окном, сохраняется
WS_MAXIMIZE	#01000000	Создается распахнутое окно
WS_CAPTION	#00C00000	Окно с заголовком и стилем WS_BORDER
WS_BORDER	#00800000	У окна есть тонкая ограничивающая рамка
WS_DLGFRAME	#00400000	Окно имеет стиль диалогового и не может иметь заголовока
WS_VSCROLL	#00200000	Окно с вертикальной линейкой прокрутки
WS_HSCROLL	#00100000	Окно с горизонтальной линейкой прокрутки
WS_SYSMENU	#00080000	Окно с системным меню
WS_THICKFRAME	#00040000	Окно без заголовка с широкой рамкой, позволяющей изменять его размеры
WS_GROUP	#00020000	Окно, объединяющее элементы в группу
WS_TABSTOP	#00010000	Окно активизируется при нажатии клавиши табуляции

Стиль	*Значение*	*Описание*
WS_MINIMIZEBOX	#00020000	У окна есть кнопка, сворачивающая изображение
WS_MAXIMIZEBOX	#00010000	У окна есть кнопка, распахивающая изображение
WS_TILED	#00000000	WS_OVERLAPPED
WS_ICONIC	#20000000	WS_MINIMIZE
WS_SIZEBOX	#00040000	WS_THICKFRAME
WS_CHILDWINDOW	#40000000	WS_CHILD
WS_OVERLAPPEDWINDOW	#80880000	WS_OVERLAPPED
WS_POPUPWINDOW	#00010000	Комбинация WS_BORDER, WS_POPUP, WS_SYSMENU
WS_TILEDWINDOW	#00000000	Комбинация WS_OVERLAPPED, WS_CAPTION

Окно помещаем в левый верхний угол (**x = 0, y= 0**). Размеры окна определяем, дважды вызывая функцию **GetSystemMetrics()**. Дескриптор приложения нам уже известен. Остальные параметры равны нулю по вполне понятным причинам.

Функция **CreateWindow()** создает окно как некоторую структуру в памяти и возвращает дескриптор окна. Это вовсе не означает, что окно появиться на экране. Для отображения окна используем функцию **ShowWindow(ghwndMain, nShowCmd)**. Первый параметр – это дескриптор окна **ghwndMain**, а второй – **nShowCmd** – устанавливает способ отображения. В нашем случае мы используем значение, полученное при вызове функции **WinMain()**.

Функция **UpDateWindow(ghwndMain)** посылает оконной функции сообщение о необходимости перерисовки рабочей области окна. Система перерисовывает только внешние атрибуты окна. Ответственность за эту процедуру полностью лежит на приложении.

Текст функции, которая создает окно, приведен ниже.

```
!*****************************************************************************
!                    InitInst
!  Цель: сохранить дескриптор и создать главное окно
!*****************************************************************************
integer(4) function InitInst (nShowCmd)
!_____________________________________________________________________________
```

```
use MyPr_1inc

integer(4)      :: hInstance, nShowCmd
integer(4)      :: iW, iH
integer(4)      :: ir

iW = GetSystemMetrics(SM_CXSCREEN)! Размеры
iH = GetSystemMetrics(SM_CYSCREEN) ! экрана

ghwndMain =   CreateWindow(SZWINNAME, SZTITLE, &
              WS_OVERLAPPEDWINDOW,             &
                  0, 0, iW, iH,                &
                  NULL, NULL, hInst, NULL)

if (ghwndMain == NULL) then
  InitInst = 0
  return
end if

ir = ShowWindow(ghwndMain, nShowCmd)
ir = UpDateWindow(ghwndMain)

InitInst = 1
end
!*****************************************************************************
```

3.4. Цикл обработки сообщений

Программа **WinMain()** завершается циклом обработки сообщений*

```
!--------- Обработка сообщений ----------
do while (GetMessage(mesg, NULL, 0, 0))
  ir = TranslateMessage(mesg)
  ir = DispatchMessage(mesg)
end do.
!-----------------------------------------------
```

Такой цикл содержится во всех программах для Windows. Его целью является получение и обработка сообщений, передаваемых операционной системой. Эти сообщения ставятся в очередь сообщений вашего приложения, откуда они затем (по мере готовности программы) выбираются функцией **GetMessage(lpMSG, hWnd, wMSGFilterMin, wMSGFilterMax)**.

---

* *Вместо оператора do while() лучше использовать цикл do без указания пределов и условный оператор if совместно с оператором exit в теле цикла.*

Первый параметр – это структура типа T_MSG. Она содержит следующие поля:

```
integer      hwnd        – дескриптор окна, которому адресовано сообщение,
integer      message     – собственно сообщение,
integer      wParam      – дополнительная информация,
integer      lParam      – дополнительная информация,
integer      time        – время посылки сообщения
type(T_POINT) pt         – положение курсора мыши (pt%x и pt%y).
```

Следующий параметр определяет окно, которому направляется сообщение. Если в приложении несколько окон, то может потребоваться выделить сообщения, адресованные одному из них. Однако, как правило, вы хотите получать все сообщения, адресованные вашему приложению, и тогда параметр **hWnd** должен быть равен NULL.

Последние два параметра определяют границы получаемых сообщений. Как правило, вам нужны все сообщения и поэтому оба параметра должны быть равны нулю.

Функция **GetMessage()** возвращает нуль, если пользователь завершает программу, и ненулевое значение в остальных случаях. Прикладная программа обычно использует значение возврата, чтобы закончить основной цикл обработки сообщений и закрыть программу.

Внутри цикла обработки сообщений вызываются две функции. Вначале вызывается функция **TranslateMessage(lpMSG)**, которая транслирует коды клавиш в клавиатурные сообщения. Затем – функция **DispatchMessage (lpMSG)**, которая возвращает сообщение после обработки Системе. Система хранит его до тех пор, пока оно не будет послано оконной функции **MainWndProc()**.

3.5. Оконная функция

Оконная функция **MainWndProc()**, как мы уже говорили выше, должна обрабатывать сообщения. В качестве параметров этой функции передаются первые четыре поля структуры T_MSG – **hwnd**, **message**, **wParam**, **lParam**. В нашем первом приложении пока используется собственно сообщение **message**. Однако скоро мы познакомимся и с тем, как используются остальные параметры.

На данном этапе оконная функция обрабатывает одно-единственное сообщение WM_DESTROY. Это сообщение посылается окну, когда пользователь завершает программу. В ответ оконная процедура вызывает функцию **PostQuitMessage(0)**. Параметр этой функции используется как

код возврата. Вызов этой функции приводит к посылке сообщения WM_QUIT, получив которое, функция **GetMessage()** возвращает нулевое значение, завершает цикл обработки сообщений и закрывает вашу программу.

Все остальные сообщения, получаемые оконной функцией, пересылаются системе с помощью вызова функции **DefWindowProc()** для дальнейшей обработки. Это необходимо, поскольку все сообщения должны быть обработаны тем или иным способом.

Текст оконной функции нашего приложения приведен ниже.

```
!***************************************************************
!          Оконная функция:     MainWndProc
!***************************************************************
integer(4) function MainWndProc(hwnd, message, wParam, lParam)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_MainWndProc@16' :: MainWndProc
!_____________________________________________________________
use MyPr_1inc

integer(4)     :: hwnd, message, wParam, lParam

!-------------- начало процедуры обработки сообщений ---------
select case (message)
 case (WM_DESTROY)
!------- разрушить все ----------------------------
    call PostQuitMessage(0)
 case DEFAULT
!---------- по умолчанию --------------------------------
  MainWndProc = DefWindowProc(hWnd, message, wParam, lParam)
  return
end select
MainWndProc = 0
end
!*******************************************************************************
```

3.6. Модуль MyPr_1inc

Как уже говорилось выше, файл ***MyPr_1inc.f90*** содержит модуль с глобальными данными. Текст его пока еще очень короток и вполне понятен для тех, кто знаком с современным Фортраном, однако следует обратить внимание на вторую строку – **use msfwina**:

```
!*****************************************************************************
module MyPr_1inc

use msfwina
!_____________________________________________________________________________
! constants

character, parameter          :: NullChar = 0
character*60, parameter, public :: SZTITLE  = &
                    "Мое первое приложение WIN32"C
character*60, parameter, public :: SZWINNAME ="Moe_OKHO"C

integer                       :: hInst, ghwndMain

!*****************************************************
end module MyPr_1inc
```

Оператор **use** в модуле **MyPr_1inc** играет ключевую роль в нашем приложении. Он обеспечивает доступ к модулю **msfwina**, который находится в файле **MsfWin.f90**. Через этот модуль в свою очередь обеспечивается доступ к модулю **msfwin**, который находится в том же файле. Такой запутанный путь необходим для устранения конфликтов между WIN32 API и **QuickWin**. Файл **MsfWin.f90** находится в директории **Msdev\Iclude** и содержит интерфейсы большинства оконных функций WIN32 API.

В модуле **msfwin** есть ссылка на модуль **msfwinty**, который содержит константы и описания производных типов данных. По своей сути файлы **MsfWin.f90** и **MsfWinty.f90** выполняют совместно те же функции, что и файл **Winuser.h** в программах Visual C++.

Поскольку программирование приложения в Фортране ведется с использованием функций Visual C++, то необходимо соблюдать соглашения о вызовах(см., например, [4]). Вот как, например, выглядит интерфейс функции отображения окна **ShowWindow()**:

```
interface
logical(4) function ShowWindow (hWnd, nCmdShow)
!MS$ATTRIBUTES STDCALL, ALIAS : '_ShowWindow@8' :: ShowWindow
integer hWnd
integer nCmdShow
end function ShowWindow
end interface
```

В дальнейшем нам с вами придется добавлять в приложение функции WIN32 API, которые отсутствуют в файле **Msfwin.f90**. Интерфей-

сы всех вновь добавленных функций должны иметь метакоманду **!MS$ATTRIBUTES**.

3.7. Создание исполняемого файла

Для создания исполняемого файла **MyPr_1.exe** необходимо использовать пункт меню ***Build***. В результате получим отладочную версию приложения. Результат действия программы показан на рис. 3.1.

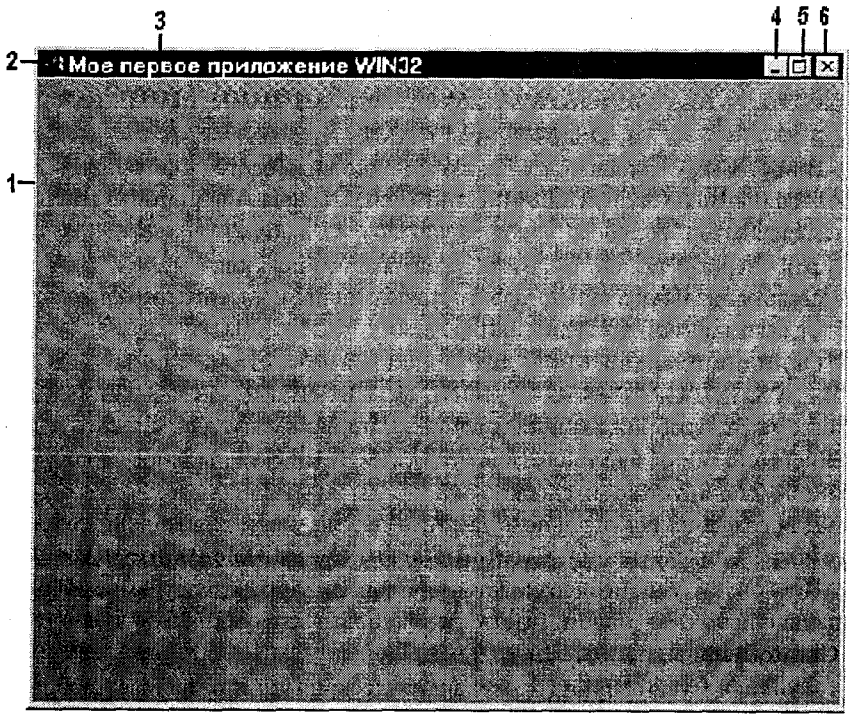

Рис. 3.1. Главное окно приложения:
1 – рамка; 2 – иконка системного меню; 3 – заголовок окна;
4 – кнопка минимизации; 5 – кнопка полноэкранного вида;
6 – кнопка закрытия экрана

Для того чтобы создать версию приложения без отладочных средств, необходимо на панели инструментов в окне ***Select Default Project Configuration*** установить ***MyPr_1-WIN32 Release*** и использовать пункт меню ***Build***.

УРОК

4. Меню и обработка сообщений

Меню является наиболее распространенным элементом управления. Главное окно практически любого приложения имеет меню. Поскольку меню так распространены и так важны, Windows обеспечивает для них достаточно мощные средства поддержки.

Структура меню представляет собой совокупность отдельных взаимосвязанных пунктов или элементов. Элементы могут быть разбиты на уровни.

Меню верхнего уровня располагается в верхней части окна, под его заголовком. Ниже линейки меню отображаются подменю, которые появляются как всплывающие окна.

Кроме того, имеется возможность создать подвижное, всплывающее меню в любой точке рабочей области окна.

Для включения меню в программу и его использования для управления приложением необходимо сделать следующие шаги:

1. Определить форму меню в файле ресурсов.
2. Загрузить меню при создании главного окна.
3. Организовать обработку сообщений меню.

Прежде чем перейти к созданию приложения с меню, следует рассмотреть, что же такое ресурсы, о которых шла речь выше.

4.1. Что такое ресурсы

Некоторые типы объектов в системе Windows определяются как ресурсы. ***Ресурс*** – это двоичные данные, которые используются в программе, но не определяются в ней. Данные в ресурсе описывают иконку, курсор, меню, диалоговое окно, растровое изображение (файлы ***.bmp**), шрифт, таблицы ускорителей, таблицы сообщений, таблицу строк.

Все ресурсы, заранее определенные в Win32 API, называются *стандартными*. Для работы с ними существуют специальные функции. Но именно эта стандартность и ограничивает возможности программиста.

Для того чтобы можно было преодолеть эти ограничения, был создан особый тип ресурсов – ресурсы, определяемые пользователем. Применяя

именно этот тип, мы можем предоставить программе практически любые данные. В этом случае платой за универсальность является усложнение программы, т. к. забота о манипулировании данными из ресурсов лежит уже не на Системе, а на программе, использующей эти ресурсы. Программа способна только получать идентификаторы ресурсов, загруженные в память средствами Windows. Вся дальнейшая работа с ними ложится исключительно на плечи программы!

Помимо того что ресурсы определяются до начала работы программы и добавляются в исполнительный файл, у них есть еще одна характерная черта: при загрузке файла в память ресурсы в память не загружаются. Только в случае, если тот или иной ресурс требуется для работы программы, она сама загружает его в память.

Ресурсы создаются отдельно от текстов программы и добавляются в исполнительный файл при компоновке. Они существуют в виде файлов, имеющих расширение **.rc**. Имя файла ресурсов обычно совпадает с именем исполнительного файла. Ниже мы создадим файл ресурсов с помощью средств MDS с именем **MyPr_1.rc**. Этот файл будет включен в проект. В файлах ресурсов используется специальный язык ресурсов, поэтому по виду файл скорее похож на текстовой файл. Его следовало бы называть *файлом описания ресурса*. В литературе встречаются термины *сценарий* и *шаблон ресурса*. Вот как выглядит фрагмент текста:

```
// Menu
ГЛАВНОЕ_МЕНЮ MENU DISCARDABLE
BEGIN
  POPUP "&Файл"
  BEGIN
    MENUITEM "Созд&ать\tCtl+N",          IDM_CREATENEW
    MENUITEM "&Открыть\tCtr+O",          IDM_OPENFILE
    MENUITEM "&Закрыть",                 IDM_CLOSEFILE
    MENUITEM SEPARATOR
    MENUITEM "&Сохранить\tCtr+S",        IDM_SAVEFILE
    MENUITEM "Сохранить &как\tF12",      IDM_SAVEFILEAS
    MENUITEM SEPARATOR
    MENUITEM "&Выход\tAlt+X",            IDM_EXIT
  END
  POPUP "&Диалоги"
  BEGIN
    MENUITEM "&Окно сообщений",          IDM_DIALOG1
    MENUITEM "&Модальный диалог",        IDM_DIALOG2
    MENUITEM "&Немодальный диалог",      IDM_DIALOG3
```

```
    MENUITEM SEPARATOR
    MENUITEM "&Панель настройки",          IDM_DIALOG4
    MENUITEM "&Элементы управления",       IDM_DIALOG5
    MENUITEM "&Таблица",                   IDM_DIALOG6
    MENUITEM "&Список изображений",        IDM_DIALOG7
  END
  POPUP "&Помощь", HELP
  BEGIN
    MENUITEM "&О программе", IDM_ABOUT
    MENUITEM "&Содержание",  IDM_HELP
    MENUITEM "&Инструкция",  IDM_HELPHLP
  END
END
```

Этот фрагмент можно создать в любом текстовом редакторе. К счастью, MDS полностью избавляет вас от этой нудной работы и сама создает файл описания ресурса. И все же иногда можно использовать "смешанный" способ редактирования ресурсов. Например, при визуальном редактировании диалоговых окон достаточно трудно точно разместить элементы диалогового окна именно так, как хочется, но это можно сделать, редактируя файл ***.rc** как обычный текстовой файл, точно указывая при этом позиции и все размеры. Кроме того, можно оперативно изменить ресурс непосредственно в приложении.

Некоторые ресурсы, такие, как иконки, курсоры, диалоговые окна, изображения, могут быть сохранены в отдельных файлах с расширениями **.ico**, **.cur**, **.dig**, **.bmp** соответственно. В этом случае в файлах-описаниях делаются ссылки на упомянутые файлы.

В этом уроке мы будем рассматривать только те ресурсы, которые обеспечивают непосредственный диалог пользователя с программой. К их числу, прежде всего, относятся меню. Во-первых, именно с меню начинается знакомство с программой. Во-вторых, оценить функциональные возможности программы можно, просто взглянув на меню. То есть именно меню в большинстве случаев является визитной карточкой программы.

Кроме меню, наиболее часто для взаимодействия с пользователем применяются диалоговые окна. Они, как правило, применяются для ввода данных и информирования программы о принятых решениях. При их рассмотрении нам придется изучить элементы управления (***controls***) и общие элементы управления (***common controls***), научиться взаимодействовать с ними, устанавливать и считывать их состояние. Обращаю внимание читателя на следующее. Понимание работы меню и диалоговых окон очень

важно, т. к. оно делает возможным написание программ для Windows, несущих какую-то полезную нагрузку.

Ясно, что ваша программа не может использовать файл-описание непосредственно. Этот файл должен быть преобразован в файл двоичных данных. Эту процедуру выполняет компилятор ресурсов (обычно имеет имя **RC.exe**). В результате появляется файл с расширением **.res**, который можно включить в исполнительный модуль. Если вы используете MDS, то эти операции будут выполняться автоматически.

4.2. Создание меню

Прежде чем использовать меню в программе, нужно создать его описание. Удобнее всего использовать для этого средства MDS.

Выполним следующую цепочку действий: ***Insert – Resource – Menu***. В результате появится окно описания ресурса ***Script1***. Щелкнем по рабочей области окна правой кнопкой и откроем окно ***Свойства***. Зададим для удобства вместо целочисленного идентификатора (IDR_MENU1), предлагаемого MDS, имя меню, например ГЛАВНОЕ_МЕНЮ.

> Внимание: при использовании кириллицы строка должна содержать только прописные буквы.

Следует иметь в виду, что Система более приспособлена для работы с целочисленными идентификаторами.

Используя средства MDS, опишем один за одним все пункты меню. Для этого двойным щелчком левой кнопки мыши откроем окно ***Menu Item Properties***.

Начнем с самого распространенного пункта: *Файл*. Поскольку этот пункт предполагает наличие подпунктов, устанавливаем тип ***Pop-up***. Вводим имя пункта и переходим к формированию подменю. Для каждого подпункта заполняем поля имени (**Caption**) и идентификатора (ID). Целочисленное значение ID_ автоматически генерируется MDS. Однако можно ввести собственное значение в поле ID после имени через знак равенства (=). Существуют имена, принятые в MDS по умолчанию: ID_FILE_NEW, ID_FILE_OPEN, ID_FILE_CLOSE, ID_FILE_SAVE, ID_FILE_SAVE_AS. К сожалению численные значения этих параметров не включены в файл **msfwinty.f90**, а сами имена даже не заносятся в файл **resource.fd**. Поэтому установим следующие значения идентификаторов: IDM_CREATENEW = = 100. IDM_OPENFILE = 101. IDM_SAVEFILE = 102. IDM_ SAVEFILE-AS = 103. IDM_EXIT = 104, IDM_CLOSEFILE =105.

Внимание: идентификатор должен быть введен обязательно.

Включим в меню пункты: ***Создать***, ***Открыть***, ***Закрыть***. Далее будут следовать ***Сохранить***, ***Сохранить как.*** Эти пункты отделим от первых трех разделителем. Для этого, прежде чем ввести их имена, выберем тип ***Separator***. Наконец, после еще одного разделителя вводим пункт ***Выход***.

Следующий пункт главного меню, который практически всегда присутствует в приложениях, – это ***Помощь***. Если вы желаете, чтобы этот пункт располагался справа на экране, отметьте поле ***Help***.

После того как все пункты меню будут заполнены, следует сохранить результаты в файле ресурсов. Назовем этот файл **MyPr_1.rc**.

Меню можно создать и непосредственно в приложении. Для этого предусмотрены две функции: **CreateMenu()** и **CreatePopupMenu()**. Они не имеют параметров и создают пустое меню. Структура меню и отдельные пункты организуются с помощью более чем трех десятков специальных функций. С некоторыми из них мы познакомимся ниже.

4.3. Подключение меню

Для подключения меню необходимо добавить в модуль **MyPr_1inc** две строки:

```
include "resource.fd"
character(16), parameter, public :: SZMENUNAME = "ГЛАВНОЕ_МЕНЮ"C
```

Использование файла **resource.fd** избавляет от необходимости ручного ввода численных значений идентификаторов. Однако следует иметь в виду, что оператор **include** в новом стандарте Фортрана относится к числу нерекомендуемых к употреблению[2]. Если строго следовать этой рекомендации, то необходимо в окончательном варианте приложения изменить текст файла **resource.fd** так, чтобы он стал модулем, а затем заменить оператор **include** на оператор **use**.

В функции **InitApp** необходимо указать имя ресурса меню

```
wc%lpszMenuName = LOC(SZMENUNAME)
```

Вид окна с меню показан на рис. 4.1.

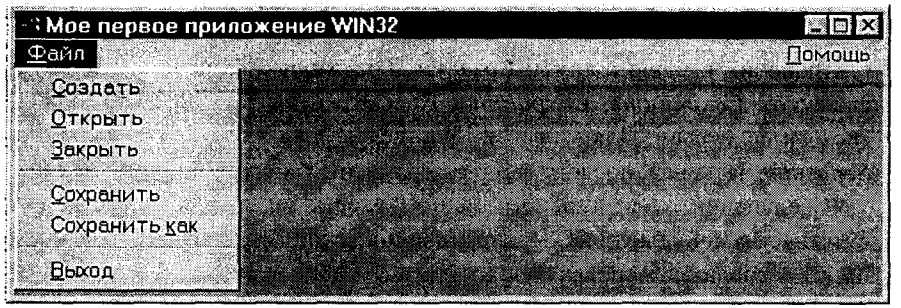

Рис. 4.1. Окно приложения с меню

4.4. Обработка сообщений

Система Windows передает управление оконной функции в форме сообщений. Сообщения генерируются Системой, окнами и приложениями. Окно генерирует сообщение при каждом изменении его состояния, например когда пользователь перемещает мышь или изменяет размеры одного из окон. Приложение может генерировать сообщения, чтобы управлять собственными окнами или чтобы выполнить некоторые действия над окнами других прикладных программ.

Сообщение представляет собой набор из четырех параметров: дескриптора окна, которому предназначено сообщение, идентификатора сообщения и двух 32-разрядных чисел (**wParam** и **lParam**), которые называют *параметрами сообщения*. Идентификатор сообщения – это именованная константа, которая идентифицирует цель сообщения. Каждое Сообщение системы имеет уникальный идентификатор и соответствующую символическую константу.

Символические константы определяют категорию, к которой принадлежит сообщение. Сообщения, общие для всех типов окон, имеют префикс WM_. Другие объекты приложений могут иметь другие приставки в именах. С некоторыми из них мы познакомимся в других уроках.

Общие сообщения окна покрывают широкий диапазон информации и запросов, включая сообщения мыши, клавиатуры, меню, диалогового окна, команды создания окна и управления им, обмена данными и т. д.

Когда оконная функция получает сообщение, она использует идентификатор для того, чтобы определить, как именно на него следует реагиро-

вать. Например, идентификатор сообщения WM_PAINT указывает, что рабочая область окна изменилась и должна быть перерисована.

Параметры сообщения определяют данные или расположение данных, используемых оконной функцией при обработке сообщения. Параметр сообщения может содержать целочисленное значение, флаги, указатели на структуры, содержащие дополнительные данные, и т. д.

В нашей простейшей программе наибольший интерес представляет сообщение WM_COMMAND. Численное значение этого сообщения равно #0111. Младшее слово параметра **wParam** содержит идентификатор пункта меню.

Кроме системных сообщений в каждой прикладной программе могут создаваться собственные сообщения для управления своими окнами и связи с окнами других программ. Если прикладная программа создает собственные сообщения, то именно оконная функция должна обеспечивать соответствующую обработку.

В Windows значения идентификатора в диапазоне от #0000 до #03FF и от #8000 до #BFFF резервируются для системных сообщений. Прикладные программы не могут использовать эти значения для собственных целей.

Значения в диапазоне от #0400 (значение WM_USER) до #7FFF доступны для идентификаторов сообщений прикладной программы. Значения в диапазоне от #C000 до #FFFF выделены для идентификаторов сообщений, определенных прикладной программой и используемых для связи с окнами других прикладных программ.

Система Windows возвращает уникальный идентификатор в диапазоне #C000 до #FFFF, если в прикладной программе использована функция **RegisterWindowMessage()** для регистрации сообщения. Идентификатор сообщения, возвращенный этой функцией, гарантирует ваше приложение от возникновения конфликтов с другими прикладными программами, в которых может использоваться тот же идентификатор.

Функция **SendMessage(hWnd, MSG, wParam, lParam)** используется для того, чтобы послать сообщение окну непосредственно. Она вызывает процедуру окна и ожидает момента, когда сообщение будет обработано, после чего возвращает результат. Первый параметр, **hWnd** – это дескриптор окна, второй, MSG – собственно сообщение, а два последних, **wParam** и **lParam** – дополнительные параметры, значение которых зависит от вида сообщения. Сообщение может быть послано любому окну в Системе. Все, что требуется, – так это программа обработки.

В многопотоковых приложениях ваша программа приостанавливает любые действия и ожидает результата, если было послано сообщение другому потоку. В результате ситуация может стать тупиковой. Поэтому перед обработкой сообщения, которое может быть послано другим потоком, процедура окна должна сначала вызвать функцию **InSendMessage (lResult)**. Если эта функция возвращает TRUE, то перед тем, как передать управление, должна быть вызвана функция **ReplyMessage(lResult)**.

Некоторые наиболее часто встречающиеся сообщения для управления окнами приведены в табл. 4.1.

Таблица 4.1. Наиболее важные сообщения для создания и управления окнами

Сообщение	*Значение*	*Описание*
WM_CREATE	#0001	Посылается функции окна после его создания
WM_DESTROY	#0002	Окно будет уничтожено
WM_MOVE	#0003	Окно было перемещено, координаты в lParam
WM_SIZE	#0005	Размеры окна были изменены, lParam – новые размеры
WM_ACTIVATE	#0006	Изменение состояние активности, lParam-дескриптор окна
WM_SETFOCUS	#0007	Устанавливает фокус клавиатуры
WM_KILLFOCUS	#0008	Уничтожает фокус клавиатуры
WM_ENABLE	#000A	Окно меняет статус доступности
WM_PAINT	#000F	Окно должно быть перерисовано
WM_CLOSE	#0010	Окно будет закрыто
WM_QUIT	#0012	Завершение программы
WM_SHOWWINDOW	#0018	Окно будет показано или спрятано
WM_COMMAND	#0111	Сообщение меню, wParam – идентификатор пункта меню
WM_SYSCOMMAND	#0112	Сообщение системного меню
WM_TIMER	#0113	Сообщение таймера
WM_HSCROLL	#0114	Сообщение линейки горизонтальной прокрутки
WM_VSCROLL	#0115	Сообщение линейки вертикальной прокрутки

При программировании на Фортране-90 надо проявлять определенную осторожность при преобразовании типа данных. Если для создания ресурсов используются средства MDS, то генерируемый системой файл **resource.fd** содержит идентификаторы в виде целых чисел типа **integer(4)**. Результатом выполнения макроса **LOWORD(wParam)** в C++ является целое число **int2** без знака в диапазоне от 0 до 65 536, а в Фортране-90 – целое типа **integer(2)**, которое изменяется от -32 768 до +32767.

> Внимание: по этой причине сравнение полученного числа с идентификатором может привести к ошибке.

Этого можно избежать, если использовать макрос **MakeLong (LOWORD(wParam), 0)*** Фортран-90 не содержит встроенных функций, аналогичных **LOWORD(dWrd)**, **HIWORD(dWrd)**, **MakeLong(lWrd, hWrd)**. Однако они определены в файле **MsfWin.f90** и поэтому оказываются доступными для пользователя, создающего приложения. Файл **MsfWin.f90** содержит интерфейсы целого ряда функций языка C++, область использования которых не ограничивается рамками создания оконного интерфейса. Можно сказать, что благодаря файлу **MsfWin.f90** мы имеем дело с расширением языка Фортран-90.

```
!****************************************************************
!          Оконная функция:    MainWndProc
!****************************************************************
integer(4) function MainWndProc(hwnd, message, wParam, lParam)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_MainWndProc@16' :: MainWndProc
!-----------------------------------------------------------------
use MyPr_1inc

integer(4)     :: hwnd, message, wParam, lParam

character(32)  :: szText = "Выполнен  пункт  "C
!------------ начало процедуры обработки сообщений --------
select case (message)
 case (WM_COMMAND)

select case (MakeLong(LOWORD(wParam), 0))
   szText(18:24) = "Создать"
```

** В примерах программирования рекомендуется использовать операцию INT4(). Однако это не дает нужного результата. Тип данных преобразуется, но значение (отрицательное) не изменяется!*

```
  case (IDM_OPENFILE)
   szText(18:24) = "Открыть"
  case (IDM_CLOSEFILE)
   szText(18:24) = "Закрыть"
  case (IDM_SAVEFILE)
   szText(18:24) = "Сохран."
  case (IDM_SAVEFILEAS)
   szText(18:24) = "Сох.как"
  case (IDM_EXIT)
!------- закроем окно, послав ему сообщение WM_DESTROY -----------
   ir=SendMessage(ghWndMain, WM_DESTROY, 0, 0)
  case(IDM_ABOUT)
   szText(18:24) = "О прогр"
  case (IDM_HELPHLP)
 end select
 ir = MessageBox(ghwndMain, szText, "Сообщение"C, MB_OK)

case (WM_DESTROY)
!------- разрушить все ----------------------------
    call PostQuitMessage(0)
case DEFAULT
!---------- по умолчанию --------------------------------
 MainWndProc = DefWindowProc(hWnd, message, wParam, lParam)
 return
end select
MainWndProc = 0
end
!*******************************************************
```

4.5. Включение акселераторов меню

Прежде чем закончить разговор о меню, следует упомянуть о паре приемов, которые ускоряют общение с ним.

Работу с меню можно сделать более удобной, если при создании меню отметить знаком & какую-либо букву в имени пункта, например ***Созд&ать***. Это приведет к тому, что буква, следующая за &, будет подчеркнута, и теперь, если пункт меню виден на экране, соответствующая клавиша будет заменять действие мыши. Эта дополнительная услуга предоставляется автоматически и не требует каких-либо изменений в текстах приложения.

Совсем иным образом действуют так называемые акселераторы – клавиатурные комбинации, которые вы можете определить самостоятельно и которые, будучи нажаты, автоматически осуществляют выбор пункта ме-

ню даже в том случае, когда меню неактивно и не отображается на экране. Включение акселераторов в приложение потребует от вас несколько больших усилий, чем это было выше.

Во-первых, для удобства пользователя рядом с именем пункта меню следует указать комбинацию клавиш, которая ему соответствует. Это очень простая операция. Достаточно открыть окно ресурсов, а затем окно свойств пункта меню. В поле заголовка следует через комбинацию символов \t набрать комбинацию клавиш, например Ctr+N. Как вы понимаете, ничего нового в действиях приложения после этой процедуры не появится.

Для того чтобы добиться результата, необходимо добавить в ваше приложение таблицу ускорителей. Среда MDS позволяет достаточно легко это сделать. Построим таблицу ускорителей, добавляя в проект новый ресурс – ***Accelerator***. Прежде всего загрузим заготовку таблицы: ***Insert – Resource – Accelerator***. В результате появится окно с именем файла ресурсов **MyPr_1.rc** и идентификатором ресурса. Так же как и выше, заменим целочисленный идентификатор на имя, например: МОИ_УСКОРИТЕЛИ.

Заполним таблицу, используя окно "***Accel Properties***". Это окно открывается двойным щелчком левой клавиши мыши. В поле ID вводим идентификатор пункта меню. Это делается легко с помощью перечня идентификаторов проекта. Далее устанавливаем комбинацию клавиш, закрываем окно и переходим к заполнению следующей строки таблицы. Ресурс создан и добавлен в проект.

Теперь надо внести изменения в текст **WinMain**. Хотя акселераторы уже внесены в файл ресурсов, они должны быть загружены в приложение при помощи функции **LoadAccelerators(hInst, Name)**, где **hInst** – дескриптор приложения, **Name** – адрес строки с именем таблицы. После выполнения функция возвращает дескриптор таблицы или NULL, если таблица не может быть загружена. Программа не будет обрабатывать акселераторы до тех пор, пока вы не добавите в цикл обработки сообщений функцию **TranslateAccelerator(hWnd, hAccel, mes)**. Здесь **hWnd** – дескриптор окна, **hAccel** – дескриптор таблицы ускорителей и, наконец, **mes** – указатель на сообщение. Таким образом, концовка текста функции **WinMain** выглядит теперь следующим образом:

```
hAccel = LoadAccelerators(hInst, LOC("МОИ_УСКОРИТЕЛИ"C))
!-------- Обработка сообщений ----------------------
do while (GetMessage(mesg, NULL, 0, 0))
 if(.NOT.TranslateAccelerator(ghwndMain, hAccel, mesg)) then
!-------- если это не ускорители ---------------------
  ir = TranslateMessage(mesg)
```

```
  ir = DispatchMessage(mesg)
 end if
end do
!--------------------------------------------------
WinMain = mesg%wParam
end
```

Постройте исполняемый файл и испытайте ваше приложение.

4.6. Взаимодействие приложения с меню

Существует обширный арсенал API функций для создания и модернизации меню, а также управления его пунктами. Вот те, которые могут потребоваться вам при создании вашего приложения:

AppendMenu	– добавляет новый пункт в конце указанного меню,
CheckMenuItem	– проверяет, помечен ли указанный пункт меню маркером ✓,
DrawMenuBar()	– перерисовывает меню после изменений,
LoadMenu()	– загружает новое меню,
EnableMenuItem())	– делает пункт меню доступным или недоступным и рисует его черным или серым цветом,
InsertMenu()	– вставляет новый пункт в меню,
ModifyMenu()	– заменяет выбранный пункт в меню
DeleteMenu()	– уничтожает указанный пункт меню,
DestroyMenu()	– уничтожает указанное меню,
TrackPopupMenu()	– отображает всплывающее меню в заданной точке рабочей области родительского окна.

Взаимодействие этих функций с меню осуществляется через целочисленные значения флагов. Функциональное действие флажков описано в справочной системе MDS, а численные значения определены в файле **MsfWinty.f90**. Для того чтобы воспользоваться функциями, необходимо: во-первых, обратиться к справочной системе, а во-вторых, **обязательно** ознакомиться с интерфейсом в файле **MsfWin.f90**. По воле тех, кто создавал файл **MsfWin.f90**, типы данных могут отличаться от приведенных в справочной системе и содержаться в файле **winuser.h**. Более того, интерфейсы некоторых функций могут и вовсе отсутствовать. В таком случае их придется ввести в текст приложения самостоятельно. Более подробно мы обсудим эту проблему позже.

Все флаги имеют префикс MF_. Значения флагов, которые включены в файл **MsfWin.f90** приведены в табл. 4.2. Флаги разбиты на группы по ха-

рактеру действия. Большинство флагов используется несколькими функциями, а некоторые из них предназначены для определенной функции.

Таблица 4.2. Битовые флаги элементов меню

Флаг	*Значение*	*Описание*
MF_BYCOMMAND	#00000000	Выбор элемента по идентификатору
MF_BYPOSITION	#00000400	Выбор элемента по порядковому номеру
MF_SEPARATOR	#00000800	Элемент представляет собой разделительную линию
MF_ENABLED	#00000000	Элемент становится доступным
MF_GRAYED	#00000001	Элемент становиться "серым"
MF_DISABLED	#00000002	Элемент становится недоступным
MF_UNCHECKED	#00000000	Элемент не помечен
MF_CHECKED	#00000008	Элемент помечен
MF_USECHECKBITMAPS	#00000200	
MF_STRING	#00000000	Элемент содержит текст
MF_BITMAP	#00000004	Элемент содержит рисунок
MF_OWNERDRAW	#00000100	Элемент прорисовывается пользователем
MF_POPUP	#00000010	С элементом связано меню более низкого уровня
MF_MENUBREAK	#00000040	Помещает элемент на новую строку или в новую колонку
MF_MENUBARBREAK	#00000020	То же, но отделяет колонку вертикальной линией
MF_UNHILITE	#00000000	
MF_HILITE	#00000080	
MF_SYSMENU	#00002000	
MF_HELP	#00004000	
MF_MOUSESELECT	#00008000	

Функции позволяют оперативно видоизменять меню непосредственно из приложения. В качестве примера добавим в приложение меню, всплывающее в рабочей области главного окна при нажатии правой клавиши мыши. Такое меню называют иногда *контекстным*.

4.7. Создание контекстного меню

Подобное меню появляется на экране рядом с курсором в результате использования функции **TrackPopupMenu()**, интерфейс которой задан в файле **MsfWin.f90** в следующем виде:

```
interface
logical(4) function  TrackPopupMenu (hMenu, uFlags, x, y, nReserved, hWnd,
prcRect)
!MS$ATTRIBUTES STDCALL, ALIAS : '_TrackPopupMenu@28' ::
TrackPopupMenu
!MS$ATTRIBUTES REFERENCE :: prcRect
use msfwinty
integer      hMenu
integer      uFlags
integer      x
integer      y
integer      nReserved
integer      hWnd
type(T_RECT)   prcRect
end function TrackPopupMenu
end interface
```

Функция **TrackPopupMenu()** только отображает меню в любой точке родительского окна и обеспечивает выбор элемента. Для создания меню следует использовать другие средства.

Первый параметр, **hMenu** – это дескриптор меню. Значение дескриптора определяется при создании всплывающего меню непосредственно в приложении с помощью функции **CreatePopupMenu()**. Если соответствующее всплывающее меню уже существует, например как подменю главного, то дескриптор получаем, вызывая **GetSubMenu()**.

В поле **uFlags** устанавливаются битовые флаги, значения которых приведены в табл. 4.3.

Таблица 4.3. Флаги функции TrackPopupMenu()

Флаг	*Значение*	*Описание*
TPM_LEFTBUTTON	#0000	Меню возникает при нажатии левой клавиши мыши
TPM_RIGHTBUTTON	#0002	Меню возникает при нажатии правой клавиши мыши
TPM_LEFTALIGN	#0000	Окно меню размещается слева от координаты *x*

Флаг	*Значение*	*Описание*
TPM_CENTERALIGN	#0004	Окно центрируется относительно указанных координат
TPM_RIGHTALIGN	#0008	Окно меню размещается справа от координаты *x*

В поля **x** и **y** заносятся координаты верхнего левого угла окна меню, а в поле **HWnd** – дескриптор родительского окна.

Последнее поле, **PrcRect** задает границы области, за пределами которой фиксация курсора приводит к исчезновению окна меню. Если **PrcRect** = = NULL, то границы этой области совпадают с границами окна меню.

В процедуру **MainWndProc()** добавим обработку сообщения правой кнопки мыши:

```
case (WM_RBUTTONDOWN)
 iX2 = LOWORD(lParam)
 iY2 = HIWORD(lParam)
 ir  = TrackPopupMenu(hMenuWnd, TPM_RIGHTBUTTON, &
                iX2, iY2, 0, ghwndMain, null_rect)
```

Всплывающее меню располагается справа от текущей координаты курсора. Главное окно приложения объявлено родительским по отношению к окну меню.

К моменту вызова функции **TrackPopupMenu()** должен быть определен дескриптор меню. Для этого в модуль **MyPr_1inc** добавлена процедура инициализации контекстного меню правой кнопки:

```
!*****************************************************************************
integer(4) function InitRMenu()
!_____________________________________________________________________________
integer :: i, ir, hMenu
character(16), parameter, dimension(7) &
 :: SZMENU= (/ 'Востановить'C, &
    &'Выделить'C, &
    &'Размеры'C, &
    &'Подвинуть'C, &
    &'Свернуть'C, &
    &'Развернуть'C, &
    &'Закрыть'C/)
 hMenu = CreatePopupMenu()
do i=1.7
 ir = AppendMenu(hMenu, MF_STRING, i, LOC(SZMENU(i)))
```

```
end do
InitRMenu = hMenu
end function InitRMenu
!*****************************************************************************
```

Меню создается с помощью функции **CreatePopupMenu()**, которая возвращает дескриптор. У созданного меню пока еще нет элементов. Они появляются после использования функции **AppendMenu (hMenu, uFlags, uIDNewIte, lpNewItem)**. Первый параметр задает дескриптор меню, **uIDNewItem** – идентификатор нового пункта меню. Второй параметр, **uFlags** – это флаг, который задает вид элемента и его поведение.

Заголовки пунктов меню объявлены в форме массива. Дополнение пунктов меню производится в цикле **do i=1, 7**, и счетчик цикла **i** используется в качестве значения параметра **uIDNewItem**. Таким образом, при выборе пункта меню генерируется сообщение WM_COMMAND, **wParam** которого будет содержать порядковый номер строки текстового массива. Это не соответствует принятым правилам. В демонстрационном варианте с этим можно мириться, но в общем случае следует объявить идентификаторы контекстного меню, например в модуле **MyPr_1inc**.

Процедуру инициализации контекстного меню можно вызвать при инициализации приложения следующим образом:

```
hMenuWnd = InitRMenu().
```

Дескриптор меню должен быть доступен оконной функции. Для этого переменную **hMenuWnd** надо объявить в модуле **MyPr_1inc**.

Построив файл **MyPr_1.exe** и запустив приложение, после щелчка по правой клавише мыши получим на экране окно меню. Его вид показан на рис. 4.2.

Рис. 4.2. Контекстное меню

Переместив курсор за пределы окна меню, вы можете его закрыть и открыть снова в новой точке рабочей области главного окна.

Попробуйте добавить в ваше приложение обработку сообщений контекстного меню. Для этого объявите идентификаторы пунктов меню в модуле **MyPr_1inc**. Это можно сделать, например, так

```
integer, parameter, public :: IDM_CONT1 = 1
```

В оконную функцию включите строку*

```
case (IDM_DIALOG1)
    ir = MessageBox(ghwndMain,"Это окно сообщений"C,
                    "Сообщение"C,MB_OK)
```

* *Функция MessageBox() описана в уроке 5.*

УРОК

5. Диалоги

В Microsoft Windows диалоговое окно это – временное окно, которое создается для удобства общения пользователя с приложением. В прикладных программах диалоги чаще всего используются для того, чтобы запросить дополнительную информацию, которая необходима для дальнейших действий. Диалоговое окно обычно содержит один или большее количество элементов управления, с помощью которых пользователь вводит текст или посылает команды.

Хорошо организованная структура диалогов делает общение с приложением удобным, эффективным и эффектным.

5.1. Использование в приложении диалогов

В Windows существует категория стандартных, или общих, диалоговых окон, которые поддерживают такие процедуры, как открытие файла, печать файла, установка шрифтов и цветов. Так, например, перед выполнением оператора Фортрана-90 **Open()** прикладная программа может использовать диалоговое окно для того, чтобы запросить имя файла. Соответствующий стандартный диалог предоставляет возможность поиска нужного файла, передачу имени и пути прикладной программе.

Часто в прикладных программах диалоговые окна используются также для того, чтобы отобразить некоторую информацию в то время, когда пользователь работает в другом окне. Например, прикладные программы подготовки текстов часто используют диалоговое окно для поиска текстового фрагмента. Пока идет поиск текста, диалоговое окно остается на экране. Пользователь может вновь и вновь обращаться к диалоговому окну и искать то же самое слово или заменить его новым. Обычно подобные диалоговые окна отображаются на экране, пока приложение или пользователь не закроет его.

Система Windows поддерживает два типа диалогов: модальный и немодальный. В большинстве случаев используются модальное диалоги. Это означает, что программа ждет завершения диалога и только после этого продолжается ее выполнение. Немодальный диалог не задерживает вы-

полнения программы, т. е. для ее продолжения не требуется завершения диалога. Модальные диалоги проще, чем немодальные, т. к. одна-единственная функция их создает, обслуживает и закрывает.

Чтобы создать модальное и немодальное диалоговое окно, прикладная программа должна определить шаблон диалогового окна, т. е. описать вид диалогового окна и его содержание. Прикладная программа должна также обеспечить функционирование диалога в соответствии с поставленной задачей.

Шаблон диалогового окна – это двоичное описание самого диалогового окна и средств управления, которые оно содержит. Разработчик может создавать этот шаблон как ресурс, который будет загружен либо в процессе выполнения прикладной программы, либо непосредственно при выполнении программы. Процедура диалогового окна это функция, которую система вызывает, когда появляется сообщение, адресованное диалоговому окну.

Прикладная программа обычно создает диалоговое окно, используя или функцию **DialogBox()**, или функцию **CreateDialog()**. **DialogBox()** создает модальное диалоговое окно, а **CreateDialog()** – немодальное. Эти две функции загружают шаблон диалогового окна и создают окно, которое соответствует этому шаблону. Имеются другие функции, обслуживающие диалоги. Одни из них создают диалоговое окно, используя шаблоны, хранящиеся в памяти, другие передают дополнительную информацию к функции диалогового окна, когда оно уже создано.

В этом уроке мы познакомимся со стандартными диалогами.

5.2. Окно сообщений

Окно сообщений (Message Box) – это стандартное диалоговое окно, которое поддерживает очень простой способ общения с приложением. Окно обычно содержит текстовое сообщение и одну или большее количество кнопок. Его можно, например, использовать для вывода на экран сообщения или подсказки. Пользователь передает свое сообщение прикладной программе, выбирая нужную кнопку.

Прикладная программа создает блок сообщения, используя функции **MessageBox()** или **MessageBoxEx()**, определяя тексты и типы кнопок. Функция **MessageBoxEx()**, кроме того, позволяет определить язык всех текстов.

Хотя окно сообщений – это диалоговое окно, но Windows полностью берет на себя контроль над его созданием и управлением. Это означает,

что прикладная программа не определяет вид диалогового окна и процедуру его функционирования. Система самостоятельно создает собственный шаблон, основанный только на оговоренных разработчиком текстах и кнопках, и организует собственную процедуру обслуживания диалогового окна.

Окно сообщений – это модальное диалоговое окно, и Windows создает его, используя API-функцию **DialogBox()**. Поэтому оно обладает всеми свойствами модального диалога, которые описаны ниже в разд. 5.4. Так, например, при вызове **MessageBox()** или **MessageBoxEx()** Система делает родительское окно неактивным.

Функция **MessageBox(hWnd, lpText, lpCaption, uType)** создает, отображает и обслуживает окно сообщений. Параметр **hWnd** – это дескриптор родительского окна. Если этот параметр **NULL**, то окно сообщений не имеет владельца. Параметры **lpText** и **lpCaption** – это строки, оканчивающиеся нулем (С-строки), с текстом сообщения и заголовком окна. Последний параметр определяет свойства окна. Он представляет собой комбинацию значений, некоторые из которых приведены в табл. 5.1.

Таблица 5.1. Некоторые типы окон сообщений

Тип окна	*Значение*	*Описание*
MB_OK	#00000000	Отображается кнопка OK
MB_OKCANCEL	#00000001	Отображаются кнопки OK и CANCEL
MB_ABORTRETRYIGNORE	#00000002	Отображаются кнопки ABORT, RETRY и IGNORE
MB_YESNOCANCEL	#00000003	Отображаются кнопки YES, NO и CANCEL
MB_YESNO	#00000004	Отображаются кнопки YES и NO
MB_RETRYCANCEL	#00000005	Отображаются кнопки RETRY и CANCEL
MB_ICONHAND	#00000010	Отображается иконка "стоп"
MB_ICONQUESTION	#00000020	Отображается иконка "знак вопроса"
MB_ICONEXCLAMATION	#00000030	Отображается иконка "знак восклицания"
MB_ICONINFORMATION	#00000040	Отображается иконка "информация"

Тип окна	Значение	Описание
MB_DEFBUTTON1	#00000000	1-я кнопка по умолчанию
MB_DEFBUTTON2	#00000100	2-я кнопка по умолчанию
MB_HELP	#00004000	Добавляется кнопка HELP

Функция **MessageBox** возвращает нуль, если не удается создать окно. В остальных случаях возвращается число, значение которого соответствует действию пользователя. В табл. 5.2 приведены возможные сообщения функции **MessageBox**.

Таблица 5.2. Значения, возвращаемые окном сообщений

Идентификатор	Значение	Описание
IDOK	#00000001	Нажата кнопка OK
IDCANCEL	#00000002	Нажата кнопка CANCEL
IDABORT	#00000003	Нажата кнопка ABORT
IDRETRY	#00000004	Нажата кнопка RETRY
IDIGNORE	#00000005	Нажата кнопка IGNORE
IDYES	#00000006	Нажата кнопка YES
IDNO	#00000007	Нажата кнопка NO
IDCLOSE	#00000008	Нажата кнопка CLOSE
IDHELP	#00000009	Нажата кнопка HELP

Если окно сообщений имеет кнопку CANCEL, функция возвращает значение **IDCANCEL**, когда нажата клавиша ESC или выбрана кнопка CANCEL. Если блок сообщения не имеет кнопки CANCEL, то клавиша ESC не оказывает никакого действия на окно.

В оконную функцию **MainWndProc()** следует добавить следующую строку:

```
case (IDM_DIALOG1)
  ir = MessageBox(ghwndMain, "Это окно сообщений"C, "Сообщение"C,
                  MB_OK)
```

> Внимание: для того, чтобы была возможность убрать окно с экрана и продолжить выполнение приложения, вы должны предусмотреть наличие хотя бы одной-единственной кнопки OK.

После построения файла **MyPr_1.exe** и его запуска на фоне главного окна появится изображение окна сообщений (рис. 5.1).

Рис. 5.1. Окно сообщений

Рассмотрим теперь стандартные диалоги.

5.3. Стандартные диалоги

К стандартным диалогам можно отнести диалоги, предоставляемые пользователю Системой. Одним из примеров являются диалоги, обеспечивающие загрузку и сохранение файлов с заданным именем.

Две функции WIN32 API **GetOpenFileName()** или **GetSaveFileName()** создают близкие по форме диалоги.

Функция **GetOpenFileName()** создает стандартный системный диалог, который позволяет выбрать имя файла для последующего его использования. В файле **Msfwin.f90** ее интерфейс представлен в следующем виде:

```
interface
logical(4) function GetOpenFileName (dummy)
!MS$ ATTRIBUTES STDCALL, ALIAS:'_GetOpenFileNameA@4' ::
GetOpenFileName
!MS$ ATTRIBUTES REFERENCE :: dummy
use msfwinty
type(T_OPENFILENAME) dummy
end function GetOpenFileName
end interface
```

Здесь ***dummy*** – структура T_OPENFILENAME, которая содержит информацию для инициализации диалога. После выполнения функция **GetOpenFileName** передает значения **TRUE** или **FALSE** и возвращает структуру T_OPENFILENAME с выбранными пользователем данными.

Структура T_OPENFILENAME определена в файле **Msfwinty.f90.** Она столь обширна, что за подробными сведениями следует обратиться к справочной системе MDS. Остановимся только на наиболее важных элементах:

lStructSize – задает размер структуры в байтах; **hwndOwner** – дескриптор родительского окна (может быть нулем); **hInstance** – значение элемента игнорируется в большинстве случаев.

Элемент **lpstrFilter** заслуживает особого внимания. Он задает шаблон и фильтр для поиска файлов. Шаблон задается в виде нескольких парных строк, каждая из которых заканчивается нулем. Первая строка пары содержит просто текст шаблона (например, "Файлы данных..."), в то время как вторая играет роль собственно фильтра(например, ***.dat**). В сложных шаблонах отдельные образцы файлов разделяются знаком ";" (например, ***.dat;*.rtm;*.dvh**). Последняя строка заканчивается двумя нулевыми символами.

Элемент ***lpstrFile*** содержит имя файла, которое будет использовано при инициализации диалога. Если это не требуется, то первый символ строки должен быть нулем. Функции **GetOpenFileName()** или **GetSaveFileName()** возвращают путь и имя выбранного файла. Элемент **lpstrFileTitle** содержит только имя файла. Этот элемент – наследие Windows 3.1. Если это число равно нулю, имя файла не возвращается; **nMaxFile** указывает размер в байтах текстового буфера, на который указывает **lpstrFile**. Для **lpstrFileTitle** тот же смысл имеет элемент **nMaxFileTitle**; **lpstrInitialDir** указывает на текстовой буфер, который содержит имя директории, используемое при инициализации диалога. Если **lpstrInitialDir** равно NULL, то система использует для этой цели текущую директорию. Элемент **lpstrTitle** – это указатель на строку с заголовком диалога. Если **lpstrTitle** равно NULL, то Система использует имена по умолчанию (Save As or Open).

Вид диалога устанавливается значением элемента **Flags**, которое может быть комбинацией нескольких стилей (общим числом более 20), наиболее важные из которых приведены в табл. 5.3.

Таблица 5.3. Флаги диалогов GetOpenFileName и GetSaveFileName

Флаг	*Значение*	*Описание*
OFN_READONLY	#00000001	Контрольный переключатель разрешает только чтение файла
OFN_OVERWRITEPROMPT	#00000002	Для **Save As** выдается запрос на продолжение, если файл с выбранным именем уже существует
OFN_HIDEREADONLY	#00000004	Контрольный переключатель не создается

Флаг	*Значение*	*Описание*
OFN_SHOWHELP	#00000010	Создается диалог с кнопкой ПОМОЩЬ; значение **hwndOwner** не должно быть равно NULL
OFN_CREATEPROMPT	#00002000	Если файл с выбранным именем не существует, то выдается запрос на продолжение

Текст модуля **MyPr_1inc** приведен ниже.

```
module MyPr_1inc
use msfwina
include "resource.fd"
!_____________________________________________________________
! constants

!--------- символьные ----------------------------------------
character, parameter                 :: NullChar   = 0
character(32), parameter, public     :: SZTITLE    = &
                                        "Мое первое приложение WIN32"C
character(16), parameter, public     :: SZWINNAME  = "Мое_ОКНО"C
character(16), parameter, public     :: SZMENUNAME = "ГЛАВНОЕ_МЕНЮ"C
character(256)                        :: szFile, szFileTitle, szFilter

!--------- целые ---------------------------------------------
integer                               :: hInst, ghwndMain, hAccel
!_____________________________________________________________
! типы
type (T_OPENFILENAME)          OpenFN
!_____________________________________________________________
contains
!*****************************************************
integer(4) function NameFileOpen(hWnd)

integer     hWnd
integer     ir

character*100     szTitle

szFilter = 'Файлы данных, (*.dat;*.rtm;*.dyh)'C// &
       &'*.dat;*.rtm;*.dyh'C// &
       &'Все файлы(*.*)'C//'*.*'C//NullChar
szFile = " "C
szFileTitle = " "C
```

```
szTitle= "Открыть Файл"C
!------- неизменная часть структуры ------------------
    OpenFN%lStructSize          = 76
    OpenFN%lpstrCustomFilter    = NULL
    OpenFN%nMaxCustFilter       = NULL
    OpenFN%nFilterIndex         = 1
    OpenFN%nMaxFile             = 256
    OpenFN%nMaxFileTitle        = 256
    OpenFN%nFileOffset          = NULL
    OpenFN%nFileExtension       = NULL
    OpenFN%lCustData            = NULL
    OpenFN%lpfnHook             = NULL
    OpenFN%lpTemplateName       = NULL
    OpenFN%lpstrInitialDir      = NULL
    OpenFN%lpstrDefExt          = NULL

!------- переменная часть структуры ------------------
  OpenFN%hwndOwner      = hWnd
  OpenFN%hInstance      = hInst
  OpenFN%lpstrFilter    = LOC(szFilter)
  OpenFN%lpstrFile      = LOC(szFile)
  OpenFN%lpstrFileTitle = LOC(szFileTitle)
  OpenFN%lpstrTitle     = LOC(szTitle)
  OpenFN%Flags          = IOR(OFN_CREATEPROMPT, &
                              OFN_HIDEREADONLY)

  ir=GetOpenFileName(OpenFN)
  if (ir.ne.0) then
  else
   NameFileOpen = 0
   return
  end if
  NameFileOpen = 1
end function NameFileOpen
!*****************************************************************************
end module MyPr_1inc
```

В оконную функцию **MainWndProc()** следует добавить несколько строк для обработки команды IDM_OPENFILE:

```
case (IDM_OPENFILE)
   ir = NameFileOpen(ghWndMain)
   szText(18:24) = "Открыть"
   ir = MessageBox(ghwndMain, szText, "Сообщение"C, MB_OK)
```

Вид окна диалога показан на рис. 5.2.

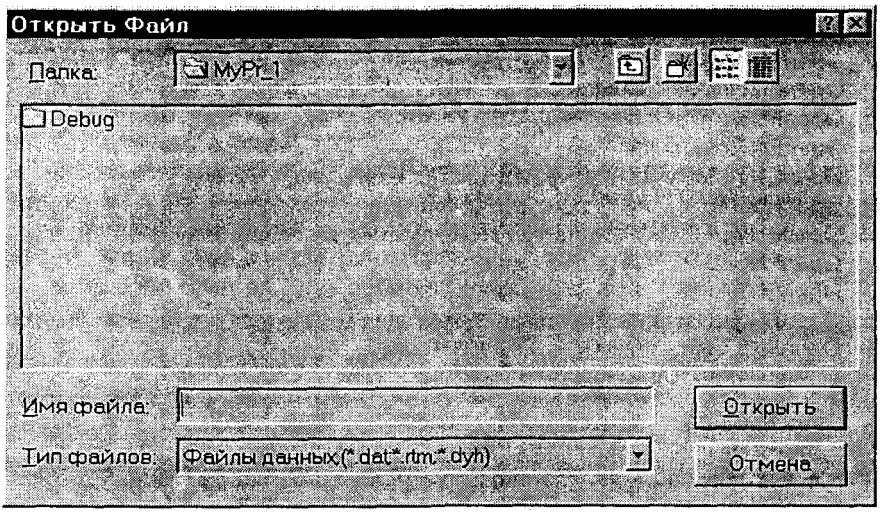

Рис. 5.2. Стандартное окно выбора файлов

Функция **GetSaveFileName()** мало чем отличается от **GetOpenFileName()**. Интерфейс отличается только именем функции. Предлагаю попытаться самостоятельно использовать функцию **GetSaveFileName()** в вашем приложении.

Функция **ChooseColor (dummy)** создает системное диалоговое окно для выбора цвета. В файле **Msfwin.f90** интерфейс этой функции задан следующим образом:

```
interface
logical(4) function ChooseColor (dummy)
!MS$ ATTRIBUTES STDCALL, ALIAS:'_ChooseColorA@4' :: ChooseColor
!MS$ ATTRIBUTES REFERENCE :: dummy
use msfwinty
type(T_CHOOSECOLOR) dummy
end function ChooseColor
end interface
```

Единственный параметр этой функции – это структура типа T_CHOOSECOLOR, которая содержит информацию для инициализации диалогового окна. После того как пользователь закрывает диалоговое окно, система возвращает в этой структуре вновь установленные данные.

Описание типа содержится в файле **Msfwinty.f90**. Структура содержит следующие элементы:

integer lStructSize – размер структуры,
integer hwndOwner – дескриптор родительского окна,
integer hInstance – дескриптор приложения,
integer rgbResult – цвет,
integer lpCustColors – набор из 16 дополнительных цветов,
integer Flags – флаги, определяющие стиль диалога,
integer lCustData – данные, которые передаются процедуре обработки,
integer lpfnHook – адрес процедуры обработки диалога при перехвате управления,
integer lpTemplateName – заголовок диалога.

Если добавить вызов функции **ChooseColor()** в приложение, то в ответ на какое-либо подходящее сообщение можно на экране получить изображение, показанное на рис. 5.3.

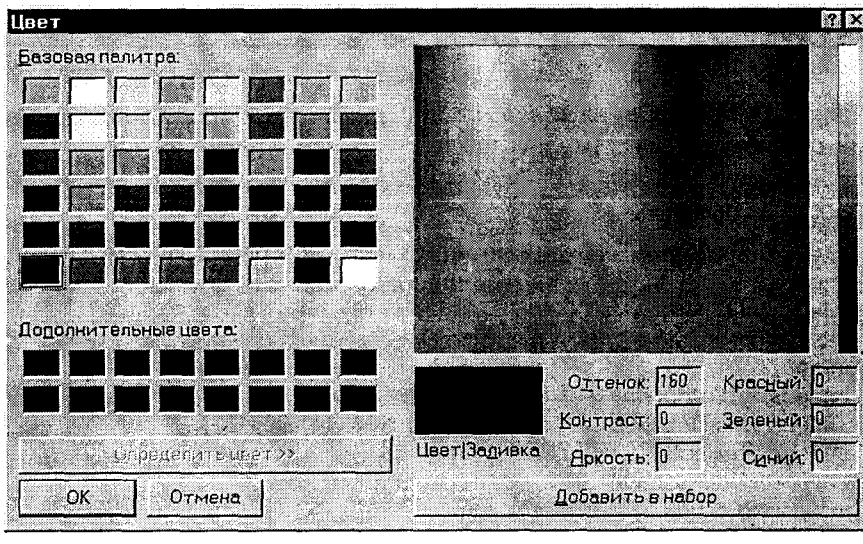

Рис. 5.3. Диалог выбора цвета

Предварительно надо определить массив для хранения дополнительных цветов, которые выбираются пользователем или устанавливаются предварительно в приложении, и структуру типа T_CHOOSECOLOR.

```
integer(4)     :: iColor(16)
type (T_CHOOSECOLOR)   :: TCC
```

```
case (IDM_CREATENEW)
  TCC%lStructSize =36
  TCC%hwndOwner     =ghwndMain
  TCC%lpCustColors  = LOC(iColor)
  TCC%Flags         = CC_FULLOPEN
  ir                = ChooseColor(TCC)
```

Функция **ChooseFont (dummy)** создает системное диалоговое окно для выбора шрифта. В файле **Msfwin.f90** интерфейс этой функции задан следующим образом:

```
interface
logical(4) function  ChooseFont (dummy)
!MS$ ATTRIBUTES STDCALL, ALIAS:'_ChooseFontA@4' :: ChooseFont
!MS$ ATTRIBUTES REFERENCE :: dummy
use msfwinty
type(T_CHOOSEFONT) dummy
end function ChooseFont
end interface
```

Параметр **dummy** – это структура типа T_CHOOSEFONT, которая содержит информацию для инициализации диалогового окна. Описание типа содержится в файле **Msfwinty.f90**. Структура содержит следующие элементы:

```
integer lStructSize       – размер структуры,
  integer hwndOwner       – дескриптор родительского окна или NULL
  integer hDC             – контекст устройств печати или NULL,
  integer lpLogFont       – указатель на структуру T_LOGFONT,
  integer iPointSize      – десятые доли размера шрифта,
  integer Flags           – флаги типа диалога,
  integer rgbColors       – цвет шрифта,
  integer lCustData       – данные для функции перехвата,
  integer lpfnHook        – указатель функции перехвата,
  integer lpTemplateName  – заголовок диалога,
  integer hInstance       – дескриптор приложения,
  integer lpszStyle       – указатель буфера стиля,
  integer(2)  nFontType   – тип шрифта,
  integer(2)  MISSING_ALIGNMENT,
  integer nSizeMin        – минимальный размер,
  integer nSizeMax        – максимальный размер,
```

Эта структура содержит информацию для инициализации диалогового окна Системой. Когда пользователь закрывает диалоговое окно, Система возвращает информацию о выбранном типе шрифта.

Структура T_CHOOSEFONT должна быть определена в нашем приложении. Кроме того, надо определить структуру, которая будет содержать параметры выбранного шрифта. Это можно сделать, например, в файле **MyPr_1.f90** следующим образом:

```
type (T_CHOOSEFONT)     :: TCF
type (T_LOGFONT)        :: TLF
```

Три поля структуры TCF должны быть заполнены до вызова функции **ChooseFont()**. Фрагмент текста для вызова стандартного диалога выбора шрифта имеет вид:

```
TCF%lStructSize     = 60
TCF%hwndOwner       = ghwndMain
TCF%lpLogFont       = LOC(TLF)
 TCF%Flags          = IOR(CF_SCREENFONTS, CF_APPLY)
ir                  = ChooseFont(TCF)
```

Вызов функции **ChooseFont()** в приложение выводит окно диалога, внешний вид которого показан на рис. 5.4.

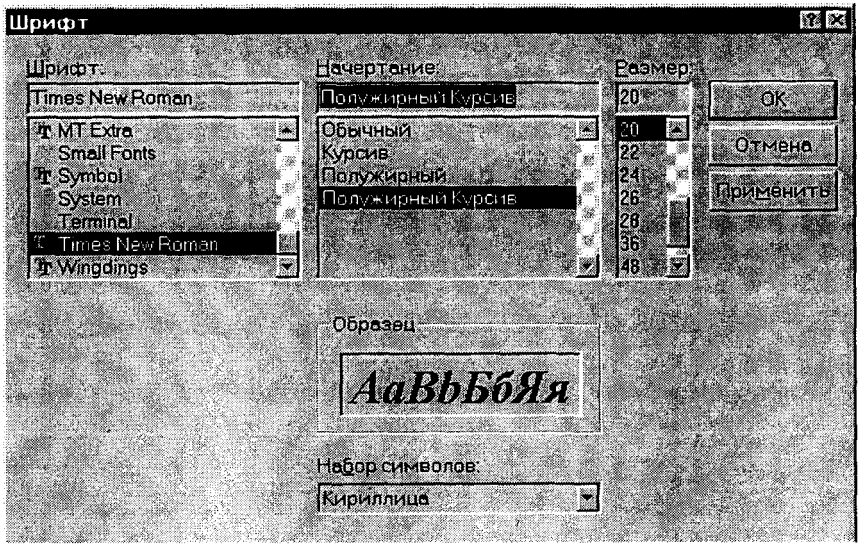

Рис. 5.4. Окно стандартного диалога для выбора шрифта

После того как вы выбрали тип шрифта, вся необходимая информация будет содержаться в структуре TLF. Попытайтесь самостоятельно вывести какой-либо текст с помощью функции **TextOut(hDC, iX, iY, iStr, iLn)**. Предварительно создайте шрифт, используя данные, полученные из диалога. Для этой цели в приложение необходимо добавить следующий фрагмент:

```
hFont = CreateFontIndirect(TLF)
hOldFont = SelectObject(hDCScl, hFont)
```

Необходимые сведения о функции можно найти в справочной системе.

УРОК

6. Пользовательские диалоги

Стандартные диалоги решают лишь ограниченный круг задач. Подавляющая часть диалогов создается разработчиком приложения. Эти диалоги могут быть модальными и немодальными.

Для модального диалогового окна объявляются стили WS_POPUP, WS_SYSMENU, WS_CAPTION и DS_MODALFRAME.

В программу модальное диалоговое окно включается с помощью функций **DialogBox()** или **DialogBoxIndirect()**. Первая функция использует идентификатор ресурса, который содержит шаблон диалогового окна. Вторая оперирует блоком памяти, содержащим шаблон.

Окно модального диалога остается активным до того момента, когда будет вызвана функция **EndDialog()**.

Окно немодального диалога должно иметь стиль в виде комбинации WS_POPUP, WS_CAPTION, WS_BORDER и WS_SYSMENU. Система не отображает диалоговое окно автоматически, если только в шаблоне не объявлен стиль WS_VISIBLE. Немодальное диалоговое окно не блокирует родительское окно.

Для создания немодального диалога, так же как и для модального, предусмотрены две функции: **CreateDialog()** и **CreateDialogIndirect()**.

Поскольку удобнее создавать шаблон диалога средствами MDS, то в дальнейшем мы будем иметь дело только с функциями **DialogBox()** и **CreateDialog()**.

Функция **CreateDialog()** возвращает дескриптор и разрешает использовать процедуру диалогового окна для управления. Например, если стиль WS_VISIBLE не был определен в шаблоне диалогового окна, то прикладная программа может отобразить диалоговое окно с помощью функции **ShowWindow()**.

При создании диалоговое окно активно, но пользователь или прикладная программа может изменить его состояние в любое время.

Немодальный диалог закрывается с помощью функции **DestroyWindow()**. Процедура диалогового окна не должна вызвать для этого функцию **EndDialog()**.

6.1. Построение модального диалога

Добавим новый ресурс в проект, используя средства MDS. Для этого выполним следующую цепочку действий: ***Insert – Resource – Dilog*** – *выбрать любой тип диалога, например* ***FormView***, – ***OK***. На экране появится окно диалога, которое можно редактировать. Однако прежде нужно вернуться в окно проекта (***Project Workspace***) и, щелкнув правой кнопкой мыши на имени вновь созданного диалога, открыть окно ***Dialog Properties***. Для удобства заменим целочисленный идентификатор на имя ресурса, например: МОЙ_ДИАЛОГ1. Кроме того, заменим английский язык на русский.

Далее отредактируем уже имеющееся у нас поле со статическим текстом. Щелкнем левой кнопкой мыши на поле надписи **TODO: lauout formview**. В результате появятся границы этого поля. Теперь можно изменять его размеры и положение. Если же после этого щелкнуть мышкой дважды, то откроется окно ***Text Properties***. Заменим идентификатор статического текста на IDC_STATIC1. Через знак равенства можно ввести его численное значение, а можно довериться MDS. Все зависит от ваших дальнейших планов.

Для каждого шаблона диалогового окна необходимо задать некоторую комбинацию значений стиля, которая определяет его вид и особенности функционирования. Кроме общих для всех окон стилей для диалогов определены специализированные. В табл. 6.1 приведены значения таких стилей.

Таблица 6.1. Стили диалоговых окон

Стиль	*Значение*	*Описание*
DS_ABSALIGN	#0001	Координаты окна диалога – координаты экрана
DS_SYSMODAL	#0002	Разрешает доступ к другим окнам
DS_LOCALEDIT	#0020	Окно не взаимодействует с WIN32
DS_SETFONT	#0040	Установка шрифта
DS_MODALFRAME	#0080	Окно модального диалога
DS_NOIDLEMSG	#0100	Запрещает посылку сообщения WM_ENTERIDLE
DS_SETFOREGROUND	#0200	Активизирует окно диалога

Для того чтобы объявить стиль, следует перейти в окно ***Styles*** редактора ресурсов. Окно модального диалога должно иметь стиль POPUP. У него должны быть системное меню, заголовок и широкая рамка, т. е. стиль диа-

логового окна должен быть определен в виде комбинации WS_POPUP, WS_SYSMENU, WS_CAPTION и DS_MODALFRAME. Система всегда отображают модальное диалоговое окно независимо от того, установлен или нет в шаблоне стиль WS_VISIBLE. Нельзя использовать стиль WS_CHILD. Модальное диалоговое окно с этим стилем становится недоступным для прикладной программы. Для того чтобы достичь нужного результата, в поле ***Style*** следует установить именно стиль POPUP, а в поле ***Border*** установить DIALOG FRAME. Остальные параметры более или менее очевидны. Изменяя их, вы увидите результат на экране сразу или, проведя тестирование из меню ***Dialog***.

В процессе редактирования для того, чтобы добавить новые поля или какие-то другие элементы, удобно использовать окно редактора ресурсов с управляющими элементами (меню ***Controls***).

Ради практики добавим кнопку CANCEL. Выберем в меню ***Controls*** элемент ***Button*** и перенесем его на рабочую область окна нашего диалога. Пока видны границы кнопки, можно менять ее размеры и положение. Затем двойным щелчком откроем окно ***Push Button Properties***. В поле ID установим идентификатор. Для стандартных кнопок удобно использовать список редактора ресурсов. Для кнопки CANCEL это IDCANCEL. Численные значения этих идентификаторов содержатся в файле **Mswinty.f90**. Как уже говорилось выше, надо быть совершенно уверенным, что численные значения идентификатора будут известны программным блокам. Дело в том, что принятые по умолчанию идентификаторы не заносятся автоматически в файл **resource.fd**. В следующем уроке процедура создания кнопок рассмотрена более подробно.

После того как редактирование ресурса закончено, необходимо произвести некоторые изменения в текстах файлов проекта.

6.2. Включение диалога в программу

Как уже говорилось выше? прикладная программа создает модальное диалоговое окно, используя функцию **DialogBox(hInst, lpszTemplate, hwndOwner, dlgprc)**. Эта функция создает модальное диалоговое окно из ресурса с шаблоном диалогового окна. Она не возвращает управление до тех пор, пока диалог не будет завершен. Первый параметр, **hInst** – это идентификатор приложения, второй, **lpszTemplate** – идентификатор ресурса диалога, следующий, **hwndOwner** – дескриптор родительского окна, и наконец, последний, **dlgprc** – адрес функции диалога.

Идентификатор ресурса является или адресом строки с нулевым символом в конце, которая определяет имя шаблона диалогового окна, или целочисленным значением, которое определяет идентификатор ресурса шаблона диалогового окна. Если параметр определяет идентификатор ресурса, старшее слово должно быть нулем, а слово младшего разряда должно содержать сам идентификатор. Вы можете использовать функцию **MakeINTResource()**, чтобы генерировать это значение.

Если диалог не удалось создать, то функция **DialogBox()** возвращает -1. В случае успешного завершения возвращается значение параметра **nResult** функции **EndDialog()**.

Функция **DialogBoxParam()** также создает модальные диалоговые окна. Отличие в том, что эта функция позволяет передать какой-либо параметр для использования его в процедуре диалогового окна. **Dialog BoxIndirect()** требует программы обработки объекта памяти, содержащего шаблон диалогового окна.

При создании модального диалогового окна Система делает это окно активным. Оно остается активным до вызова процедурой диалогового окна функции **EndDialog()**. Ни пользователь, ни прикладная программа не могут сделать активным родительское окно до тех пор, пока модальный диалог не завершен. Хотя сама функция диалогового окна и способна в любое время разрешить доступ к родительскому окну, однако это может нарушить функционирование диалога, и поэтому использовать ее не рекомендуется.

Прикладная программа уничтожает модальное диалоговое окно, используя функцию **EndDialog()**. В большинстве случаев функция диалогового окна вызывает **EndDialog()**, когда пользователь выбирает команду CLOSE из системного меню или нажимает кнопку CANCEL в диалоговом окне. Диалоговое окно через функцию **DialogBox()** (или другие функции создания) может возвращать значение, которое задается одним из параметров функции **EndDialog()**. Система возвращает его после разрушения диалогового окна. Большинство прикладных программ использует это значение для того, чтобы определить, завершился ли диалог успешно или был отменен пользователем. Система не возвращает управление, пока не будет вызвана функция **EndDialog()**.

Функция **EndDialog(hDlg, nResult)**, о которой неоднократно упоминалось выше, уничтожает диалоговое окно, после чего Система закончит обработку сообщений диалогового окна. Первый параметр этой функции

hDlg представляет собой дескриптор диалогового окна, а второй, **nResult** – значение, которое будет возвращено диалоговой функцией.

Диалоговые окна, созданные функциями **DialogBox()** и **DialogBox Param()**, должны использовать только функцию **EndDialog()**.

> Внимание: прикладная программа может вызывать **EndDialog()** только внутри процедуры диалогового окна и должна использовать ее исключительно для завершения диалога.

Функция диалогового окна может вызывать **EndDialog()** в любое время, даже во время обработки сообщения WM_INITDIALOG. В этом случае диалоговое окно уничтожается прежде, чем оно появится на экране.

Функция **EndDialog()** не уничтожает диалоговое окно немедленно. Вместо этого она устанавливает флаг и разрешает процедуре диалогового окна возвратить управление Системе, которая проверяет флаг прежде, чем извлечь следующее сообщение из очереди. Если флаг установлен, цикл обработки сообщения завершается, диалоговое окно уничтожается и параметр **nResult** используется как значение, возвращаемое функцией **DialogBox()**.

Текст оконной функции модального диалога (рис. 6.1) приведен ниже.

```
!*****************************************************************************
integer*4 function DialogFunc1(hDlg, message, wParam, lParam)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_DialogFunc1@16' :: DialogFunc1

integer hDlg, message, wParam, lParam
integer hWndFocus, lInitParam

select case(message)
 case (WM_INITDIALOG)
!---------- инициализация диалога -----------------
  hWndFocus = wParam
  lInitParam   = lParam ! на случай использования DialodBoxParam
 case (WM_COMMAND)
!---------- обработка команд ----------------------
  select case (MakeLong(LOWORD(wParam), 0))
   case (IDCANCEL)
    ir = EndDialog(hDlg, 0)
  end select
 case (WM_SYSCOMMAND)
!--------- обработка сообщений системного меню ---------------
  if (wParam == SC_CLOSE) then
   ir = EndDialog (hDlg, 0)
  end if
```

```
 case DEFAULT
  DialogFunc1 = 0
  return
end select
DialogFunc1 =1
end function DialogFunc1
!*************************************************************************
```

В этом фрагменте функция **EndDialog (hDlg, 0)** вызывается при обработке как сообщений кнопки диалогового окна, так и сообщений системного меню.

Взаимодействие приложения с окном диалога осуществляется через сообщения. В тексте диалоговой функции вы можете видеть три сообщения: WM_COMMAND, WM_INITDIALOG, WM_SYSCOMMAND. Первое из них нам уже знакомо.

Сообщение WM_INITDIALOG передается функции диалогового окна прежде, чем диалоговое окно появится на экране, и обычно используется для инициализации средств управления или определения вида окна. Значение **wParam** передает идентификатор элемента управления, который получает фокус клавиатуры по умолчанию. Система назначает заданный по умолчанию фокус, только если процедура диалогового окна возвращает **.TRUE.** Значение **lParam** определяет дополнительные данные для инициализации. Эти данные передаются как параметр **lInitParam** при обращении к функции **DialogBoxParam**.

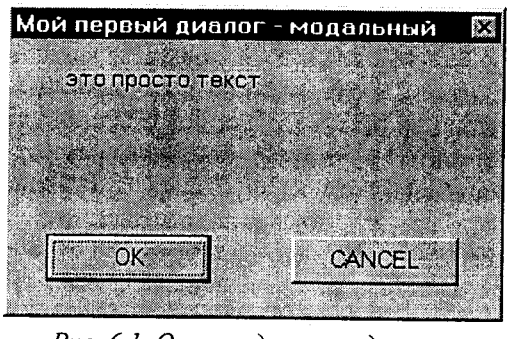

Рис. 6.1. Окно модального диалога

Окно получает сообщение WM_SYSCOMMAND, когда пользователь выбирает команду из системного меню или пользуется кнопками MAXIMIZE и MINIMIZE. Параметр **wParam** указывает тип запрошенной команды. Наиболее важные это SC_CLOSE (закрывает окно),

SC_HSCROLL (горизонтальная прокрутка), SC_VSCROLL (вертикальная прокрутка), SC_MAXIMIZE, SC_MINIMIZE.

В сообщении WM_SYSCOMMAND 4 бита младшего разряда параметра предназначены для внутреннего использования Системой. Чтобы получать правильный результат при обработке сообщения, прикладная программа должна объединить значение **0#FFF0** со значением **wParam**, используя оператор **AND**.

Посредством параметра **lParam** передаются координаты курсора, если команда была выбрана с помощью мыши. Младшее слово **lParam** указывает горизонтальную координату. Старшее – вертикальную. Если выбор произведен иначе, то горизонтальная координата не используется. Старшее слово в этом случае равно 1, если использовались системные клавиши -ускорители, или 0, если использовалась мнемоника.

Клавиши – ускорители, которые определены, чтобы выбрать пункты из меню Системы, транслируются в WM_SYSCOMMAND сообщения; все другие нажатия клавиши акселератора транслируются в WM_COMMAND сообщения.

6.3. Немодальный диалог

Ресурс немодального диалога создается совершенно также, как и модального. При использовании редактора ресурсов MDS в поле ***Style*** следует установить стиль POPUP, а в поле **Border – Thin**, отметить поля **Titlebar**, **System Menu** и **Visible**.

Для того чтобы управлять диалогом добавим в шаблон диалога пару кнопок: ВВЕСТИ и ОТМЕНИТЬ. Ради простоты, установим для этих кнопок стандартные идентификаторы: **IDOK** и **IDCANCEL**.

Если прикладная программа создает немодальный диалог на базе шаблона, который хранится в ресурсе, то она использует функции **Create Dialog()** или **CreateDialogParam()**. Функция **CreateDialog()** требует имени или идентификатора ресурса, содержащего шаблон диалогового окна.

Прикладная программа полностью отвечает за обработку и пересылку сообщений диалоговому окну. В большинстве случаев для этого достаточно использовать основной цикл обработки сообщений. Однако для того, чтобы взаимодействовать с элементами управления диалога, прикладная программа должна вызвать функцию **IsDialogMessage(hDialog, mesg)**. Первый параметр – это дескриптор диалогового окна, второй – структура типа T_MSG, о которой говорилось ранее.

Функция **IsDialogMessage()** определяет, предназначено ли сообщение для диалогового окна с дескриптором **hDialog**, и если да, то обрабатывает его. Если сообщение было обработано, то возвращается **TRUE** в противном случае возвращается **FALSE**.

Хотя функция **IsDialogMessage()** предназначена для немодальных диалогов, вы можете использовать ее с любым окном, которое содержит средства управления.

Когда **IsDialogMessage()** обрабатывает сообщение, она проверяет сообщения клавиатуры и преобразовывает их в команды выбора для соответствующего диалогового окна.

Поскольку функция **IsDialogMessage()** выполняет все необходимые действия, уже обработанное сообщение должно миновать функции **TranslateMessage()** и **DispatchMessage()**.

Обращаю ваше внимание на то, что функция **IsDialogMessage()** может посылать сообщения DM_GETDEFID и DM_SETDEFID. Эти сообщения определены в файле **Msfwinty.f90** как WM_USER и WM_USER + 1 (WM_USER = #0400). Если в приложении встречаются те же значения, то не исключено возникновение конфликтных ситуаций.

Цикл обработки сообщений в функции **WinMain()** при использовании в приложении немодального диалога выглядит следующим образом*:

```
do while (GetMessage(mesg, NULL, 0, 0))
 if(.NOT.IsDialogMessage(hDialog, mesg)) then
   ir = TranslateMessage(mesg)
   ir = DispatchMessage(mesg)
 end if
end do
```

Сообщения, адресованные диалогу, теперь пересылаются функцией **IsDialogMessage()** диалоговой процедуре, а все остальные обрабатываются обычным образом.

Немодальный диалог не может возвращать значение прикладной программе так, как это делает модальное диалоговое окно. Однако процедура диалогового окна может передавать сообщения родительскому окну, используя функцию **SendMessage()**.

Прикладная программа перед завершением должна самостоятельно уничтожить все немодальные диалоговые окна. Для этого используется

---

* *Более изящное решение получается при использовании строки if(IsDialog Message (hDialog, mesg)) cycle.*

функция **DestroyWindow()**. Обычно эта функция вызывается в ответ на команду завершения диалога. Если соответствующая команда не поступила, то прикладная программа должна сама вызвать **DestroyWindow()**.

Функция **DestroyWindow()** уничтожает дескриптор диалогового окна. Поэтому любые последующие обращения к функциям, которые используют значение дескриптора, приводят к ошибке. Чтобы избежать этого, процедура диалогового окна должна сообщить родительскому окну, что диалоговое окно было разрушено. Чаще всего это достигается установкой в нуль объявленной ранее в приложении глобальной переменной, когда функция диалога уничтожает диалоговое окно.

> Внимание: процедура немодального диалога не должна вызвать функцию **EndDialog()**.

Текст оконной функции немодального диалога приведен ниже.

```
!*****************************************************************************
integer*4 function DialogFunc2(hDlg, message, wParam, lParam)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_DialogFunc2@16' :: DialogFunc2

integer hDlg, message, wParam, lParam
integer hWndFocus, lInitParam

select case(message)
 case (WM_INITDIALOG)
!---------- инициализация диалога ----------------
  hWndFocus  = wParam
  lInitParam = lParam !на случай использования DialodBoxParam
 case (WM_COMMAND)
!---------- обработка команд ----------------------
  select case (MakeLong(LOWORD(wParam), 0))
   case (IDOK)
    ir = DestroyWindow(hDlg)
   case (IDCANCEL)
    ir = DestroyWindow(hDlg)
  end select
 case (WM_CLOSE)
  ir = DestroyWindow(hDlg)
 case DEFAULT
  DialogFunc2 = 0
  return
end select
DialogFunc2 =1
```

```
end function DialogFunc2
!*****************************************************************************
```

Внешний вид окна диалога показан на рис. 6.2.

Рис. 6.2. Окно немодального диалога

Пока в ответ на все сообщения диалоговое окно просто ликвидируется.

В оконную функцию **MainWndProc()** следует добавить в цикл обработки команд следующую строку:

```
case (IDM_DIALOG3)
  hDlg2=CreateDialog(hInst, LOC("МОЙ_ДИАЛОГ2"C), ghwndMain,
LOC(DialogFunc2))
```

Пока диалоги, которые вы можете создавать, используя материал этого раздела, чрезвычайно просты. Они содержат минимальное число элементов управления. О том, как разнообразить арсенал средств управления, мы поговорим в следующих уроках.

6.4. Оперативное редактирование окна диалога

Функции API позволяют оперативно изменять вид диалогового окна. Мы с вами уже обсуждали возможность использования функции **ShowWindow()**. Арсенал средств API позволяет изменять практически все параметры окна. Рассмотрим несколько примеров.

Начнем с изменения размеров и позиции диалогового окна. Для этого надо воспользоваться функцией **MoveWindow()**. Интерфейс этой функции имеет следующий вид:

```
interface
logical(4) function  MoveWindow (hWnd, X, Y, nWidth, nHeight, bRepaint)
!MS$ATTRIBUTES STDCALL, ALIAS : '_MoveWindow@24' :: MoveWindow
integer hWnd              – дескриптор окна,
```

```
integer x              – координаты верхнего
integer y                левого угла окна,
integer nWidth         – ширина окна,
integer nHeight        – высота окна,
logical(4) bRepaint    – флаг перерисовки.
end function MoveWindow
end interface
```

Введем эту функцию в диалоговую процедуру **DialogFunc2()**. Пусть нажатие клавиши ВВЕСТИ приводит к перемещению окна и увеличению его ширины в два раза. Здесь нам понадобится функция, которая определит первоначальные размеры окна: **GetWindowRect(hWnd, lpRect)**. Эта функция возвращает в структуре типа T_RECT экранные координаты окна.

Замените строки:

```
case (IDOK)
 ir = DestroyWindow(hDlg)
```

следующим фрагментом:

```
case (IDOK)
 ir = GetWindowRect(hDlg, TRc) ! определяем размеры окна
 ir = MoveWindow(hDlg, TRc%left/2, TRc%top/2, &
    2*(TRc%right-TRc%left), &
      (TRc%bottom-TRc%left), .TRUE.) ! рисуем новое окно
```

и испытайте приложение.

Теперь при нажатие клавиши ВВЕСТИ диалоговое окно должно перемещаться и изменять ширину.

Кнопки диалога – это тоже окна. Вы можете самостоятельно попытаться переместить и изменить размеры кнопок при изменении размеров диалогового окна. Дескриптор окна можно определить с помощью функции **GetWindow()**. Однако удобнее использовать функцию **GetDlgItem (hDlg, IDDlgItem)**, которая возвращает дескриптор элемента управления диалога с идентификатором **IDDlgItem**.

Если возникает необходимость оперативного изменения стиля окна диалога, то следует использовать пару функций **GetWindowLong (hWnd, nIndex)** и **SetWindowLong(hWnd, nIndex, dwNewLong)**. Первая передает приложению значение комбинации стилей, а вторая устанавливает новую комбинацию. Новый стиль добавляется с помощью побитовой операции **IOR()**.

УРОК

7. Элементы управления диалогом

Теперь, когда диалог создан, поговорим более подробно об элементах управления. Именно эти элементы обеспечивают взаимодействие пользователя с программой. Функции Win32 API предоставляют программисту возможность создавать различные виды таких элементов.

Элемент управления (control) представляет собой особый вид окна, предназначенный для ввода и вывода информации. Средства управления наиболее часто используются внутри диалоговых окон, однако их можно использовать и в других окнах. Элементы управления внутри диалоговых окон обеспечивают ввод текста, выбор данных из списка, завершение действия диалога и т. д.

Элементы управления обычно имеют родительское окно, например диалоговое. Система Windows поддерживает несколько типов элементов управления. Исторически сложилось разделение на *стандартные* и так называемые *общие* элементы управления. Стандартные достались нам от прежних версий системы. Начиная с версии Windows-95, появилась возможность использования некоторых новых элементов, дополняющих стандартные и расширяющих возможности ваших приложений. Такие элементы получили название *общих* (***common control***).

Начнем со стандартных элементов. К стандартным элементам управления относятся:

- кнопка (***button***);
- контрольный переключатель (***check box***);
- селекторная кнопка (***radio button***);
- список (***list box***);
- окно ввода (***edit box***);
- комбинированный список (***combo box***);
- линейка прокрутки (***scrollbar***);
- статический элемент (***static***).

Кнопка является простейшим элементом управления, который, тем не менее, входит в состав практически любого диалога.

7.1. Кнопки

Кнопка – это элемент управления, имитирующий кнопку или переключатель. Имеется несколько типов кнопок, а внутри каждого типа – еще и несколько стилей. Пользователь взаимодействует с кнопкой посредством мыши или клавиатуры. Выбор кнопки обычно изменяет ее внешний вид. Кнопка и родительское окно могут обмениваться сообщениями. Некоторые кнопки отображаются Системой, а некоторые – прикладной программой. Кнопки могут использоваться индивидуально или в составе группы.

Кнопки были разработаны для использования в диалоговых окнах, где Windows обеспечивает их стандартное поведение. В принципе их можно использовать и в других случаях. Однако если прикладная программа использует кнопки вне диалоговых окон, то увеличивается риск нестандартного поведения приложения. Windows поддерживает пять видов кнопок: *стандартную кнопку; контрольный переключатель; селекторную кнопку; группу кнопок; кнопки, создаваемые разработчиком приложения.*

Стандартная кнопка (***button***) – это прямоугольник, содержащий текст, пиктограмму или рисунок, который указывает на ее назначение. Кнопка может иметь два стиля: стандартный (BS_PUSHBUTTON) и по умолчанию (BS_DEFPUSHBUTTON)*. Стандартная кнопка обычно используется, чтобы инициировать какую-либо операцию. Кнопка активизируется, когда пользователь выбирает ее с помощью мыши или клавиатуры. С другой стороны, заданная по умолчанию кнопка обычно используется, чтобы определить заданный по умолчанию выбор. Это кнопка, которую пользователь в любой момент может привести в действие клавишей ENTER.

Когда пользователь выбирает кнопку, Windows посылает родительскому окну сообщение WM_COMMAND, содержащее код BN_CLICKED. В ответ диалоговое окно обычно закрывается или выполняется оговоренная для данной кнопки операция.

Контрольный переключатель (checkbox)состоит из квадратного окна и метки в виде текста, иконки, пиктограммы или рисунка. Появление метки сигнализирует о сделанном выборе. Прикладные программы обычно

* *Стили кнопок и сообщения, адресованные им, будут описаны позже.*

отображают переключатели в диалоге, чтобы обеспечить возможность выбора из группы связанных, но независимых опций.

Переключатель может иметь один из четырех стилей: стандартный (BS_CHECKBOX), автоматический (BS_AUTOCHECKBOX), с тремя состояниями (BS_3STATE) и автоматический с тремя состояниями или положениями (BS_AUTO3STATE). Два состояния – это состояния с меткой и без метки. Переключатель с тремя состояниями может принимать неопределенное состояние (серая метка). Состояния изменяются последовательно одно за другим.

Когда пользователь выбирает переключатель, Система посылает родительскому окну сообщение WM_COMMAND, содержащее код BN_CLICKED. Если сообщение послано автоматическим переключателем или автоматическим переключателем с тремя состояниями, то Система автоматически устанавливают состояние. Однако родительское окно в ответ на сообщение должно самостоятельно устанавливать состояние простого переключателя или переключателя с тремя положениями.

Селекторная кнопка (radiobutton) состоит из круглой кнопки и текстовой метки, иконки или рисунка. Селекторные кнопки используют в диалоге для выбора опции. Селекторная кнопка может быть двух стилей: стандартного (BS_RADIOBUTTON) или автоматического (BS_AUTORADIOBUTTON). Каждый стиль может принимать два состояния: с точкой в кнопке или без нее.

Когда пользователь выбирает любое состояние селекторной кнопки, она становится активной и Windows посылает родительскому окну сообщение WM_COMMAND, содержащее код BN_CLICKED. Автоматическая кнопка полностью обслуживается системой, а стандартная – родительским окном. Когда пользователь выбирает автоматическую селекторную кнопку, Windows автоматически сбрасывает все другие кнопки. То же самое поведение доступно и для стандартных кнопок, если использовать стиль WS_GROUP.

Группа кнопок (groupbox) – это прямоугольник, который обрамляет набор средств управления типа переключателей или селекторных кнопок с текстовой меткой в верхнем левом углу. Группа визуально объединяет средства управления, связанные общей целью, и имеет только один стиль, BS_GROUPBOX. Группу нельзя выбрать и поэтому какая-либо проверка ее состояния не предусмотрена. Прикладная программа не может посылать сообщения группе.

Объединение селекторных кнопок в группы приводит к тому, что кнопки разных групп становятся независимыми. Этот стиль не оказывает действия на другие типы элементов управления.

Кнопки, создаваемые разработчиком приложения (BS_OWNERDRAW), в отличие от селекторных кнопок прорисовывается не Системой, а прикладной программой. Когда пользователь выбирает такую кнопку, Windows посылает окну родителя кнопки сообщение WM_COMMAND, содержащее код BN_CLICKED, точно так же, как и для других кнопок, и прикладная программа должна обработать это сообщение.

7.2. Создание кнопок

Кнопки создаются как особого вида окна класса BUTTON функцией **CreateWindow()** или **CreateWindowEx()**. При создании кнопки этими функциями надо указывать ее стиль. Значения стилей кнопок и их описание приведены в табл. 7.1.

Таблица 7.1. Стили кнопок

Стиль	*Значение*	*Описание*
BS_PUSHBUTTON	#0000	Стандартная кнопка
BS_DEFPUSHBUTTON	#0001	Кнопка приводится в действие клавишей ENTER
BS_CHECKBOX	#0002	Контрольный переключатель
BS_AUTOCHECKBOX	#0003	Автоматический контрольный переключатель
BS_RADIOBUTTON	#0004	Селекторная кнопка
BS_3STATE	#0005	Кнопка с тремя состояниями
BS_AUTO3STATE	#0006	Автоматическая кнопка с тремя состояниями
BS_GROUPBOX	#0007	Объединение кнопок в группу
BS_USERBUTTON	#0008	Устаревшая форма, ориентированная на 16 бит
BS_AUTORADIOBUTTON	#0009	Автоматическая селекторная кнопка
BS_OWNERDRAW	#000B	Кнопка, создаваемая программистом
BS_LEFTTEXT	#0020	Текст слева от элемента (устаревшая форма)
BS_RIGHTBUTTON	#0020	Bs_lefttext
BS_TEXT	#0000	Кнопка с текстом
BS_ICON	#0040	Кнопка с пиктограммой
BS_BITMAP	#0080	Кнопка с рисунком
BS_LEFT	#0100	Выравнивание текста влево

Стиль	*Значение*	*Описание*
BS_RIGHT	#0200	Выравнивание текста вправо
BS_CENTER	#0300	Выравнивание текста по центру
BS_TOP	#0400	Текст у верхней границы кнопки
BS_BOTTOM	#0800	Текст у нижней границы кнопки
BS_VCENTER	#0C00	Вертикальное выравнивание по центру кнопки
BS_PUSHLIKE	#1000	Кнопка остается всегда нажатой
BS_MULTILINE	#2000	Многострочный текст кнопки
BS_NOTIFY	#4000	Разрешает посылку нотификационных сообщений
BS_FLAT	#8000	Кнопка выглядит плоской

Создавать все типы кнопок проще с помощью редактора ресурсов MDS. Добавим в диалог, который мы создавали ранее, кнопку. Для этого откроем диалог **МОЙ_ДИАЛОГ1**. В процессе редактирования шаблона диалога будем использовать окно редактора ресурсов ***Controls***.

Выбираем в меню элемент ***Button*** и перемещаем его на поле диалога. Пока видны границы, устанавливаем нужные размеры. Двойным щелчком левой клавиши мыши открываем окно ***Push Button Properties***. Заменим идентификатор, предлагаемый MDS, на стандартный **IDOK**. В поле **Caption** введем имя кнопки OK. Установим стиль окна **Visible**.

При использовании редактора ресурсов стили устанавливаются при создании шаблона диалога. Перейдем в окно ***Styles*** для того, чтобы установить стиль кнопки. Это окно содержит несколько полей, значение которых описано ниже. В каждом из полей устанавливается один из стилей, приведенных в табл. 7.1.

Поле **Default button** устанавливает стиль BS_DEFPUSHBUTTON. Заданная по умолчанию кнопка выделяется черной рамкой и может быть выбрана только клавишей ENTER, даже если она не активизирована. Система поддерживает в диалоговом окне только одну заданную по умолчанию кнопку.

Поле **Owner Draw** определяет взаимоотношения списка с родительским окном. Если установлен стиль BS_OWNERDRAW, то программист создает и перерисовывает кнопку самостоятельно в ответ на сообщение WM_DRAWITEM. Стиль BS_OWNERDRAW нельзя объединять с любыми другими стилями.

Поле **Icon** указывает, что кнопка содержит иконку.

Поле **Bitmap** указывает, что кнопка содержит рисунок.

Поле **Multiline** устанавливает многострочный режим для текстовых строк в кнопке.

Поле **Notify** разрешает передачу нотификационных сообщений родительскому окну.

Поле **Flat** устанавливает плоский (не трехмерный) вид кнопки.

Поле **Horizontal Alignment** определяет тип выравнивания текста по горизонтали: по левому краю, по центру, по правому краю или по позиции, заданной по умолчанию.

Поле **Vertical Alignment** задает смещение текста по вертикали: в верхнюю часть, нижнюю часть, в среднюю или по позиции, заданной по умолчанию.

Установите стили, которые вас устраивают.

Для демонстрации действия селекторных кнопок, объединенных в группу, создадим диалог **ПАНЕЛЬ**, который будет, например, управлять настройкой радиоприемника. Для этого проделаем процедуры, описанные в разд. 6.2. Затем добавим в диалог селекторные кнопки.

Выбираем в меню элемент ***Radio Button*** и перемещаем его на поле диалога. Пока видны границы устанавливаем нужные размеры. Двойным щелчком левой клавиши мыши открываем окно ***Radio Button Properties***. Заменим идентификатор, предлагаемый MDS, на IDC_RADIO1. В поле **Capton** введем имя кнопки, например ДЛИННЫЕ. Уставим стиль окна ***Visible*** и включим эту кнопку в группу, установив поле **Group**.

Переходим в окно ***Styles*** и устанавливаем стили кнопки так, как мы это делали для обычной кнопки. Новым для нас является следующее:

- поле **Auto** устанавливает, что при выборе конкретной селекторной кнопки любая другая кнопка этой группы сбрасывается автоматически (этот стиль обязательно используется, если кнопки объединены в группу);
- поле **Left Text** регулирует размещение текста относительно кнопки;
- поле **Pushlike** устанавливает изображение в виде простой кнопки.

Повторим эти действия еще три раза и создадим кнопки СРЕДНИЕ, КОРОТКИЕ и УКВ.

Теперь объединим четыре кнопки в группу ДИАПАЗОН. Для этого выбираем в меню редактора ресурсов элемент ***Group Box***, переносим его в окно диалога и размещаем рамку так, чтобы она охватила четыре кнопки. Открываем окно ***Group Box Properties*** и проводим необходимые установки в полях.

Для того чтобы продемонстрировать независимость групп, создадим дополнительно группу ФИКСИР. НАСТРОЙКА с парой селекторных кнопок.

Покончим с кнопками, добавляя в диалог контрольный переключатель, который будет, например, включать автоподстройку частоты радиоприемника. Выбираем в меню элемент ***Check Box*** и размещаем его в окне диалога. Открываем окно ***Check Box Properties***. Определяем идентификатор IDC_CHECK1. В поле **Capton** введем имя кнопки, например ВКЛ. АВТОПОДСТРОЙКУ. Установим стиль окна ***Visible***. Никаких незнакомых полей окно ***Styles*** не содержит.

Нужно иметь очень веские основания для создания собственных оригинальных кнопок. Поэтому не будем здесь и далее описывать процедуру создания любых элементов стиля ***Owner Draw***. Можно с уверенностью сказать, что, когда дело дойдет до элементов собственной конструкции, уровень ваших знаний и умений будет уже достаточным для выполнения любых, даже самых вычурных, задумок.

7.3. Управление кнопками

Родительское окно может посылать сообщения кнопке, используя функцию **SendMessage()**. Если кнопка находится в окне диалога, то можно использовать также функции **SendDlgItemMessage()**, **CheckDlg Button()**, **CheckRadioButton()** и **IsDlgButtonChecked()**. Для взаимодействия с кнопками Система предоставляет программисту богатый арсенал специализированных сообщений, описание которых приведено в табл. 7.2.

Таблица 7.2. Сообщения для кнопок

Сообщение	*Значение*	*Описание*
BM_GETCHECK	#00F0	Получить состояние контрольного переключателя
BM_SETCHECK	#00F1	Установить состояние контрольного переключателя
BM_GETSTATE	#00F2	Получить состояние кнопки
BM_SETSTATE	#00F3	Установить состояние кнопки
BM_SETSTYLE	#00F4	Изменить стиль кнопки
BM_CLICK	#00F5	Изменить состояние кнопки, имитация нажатия
BM_GETIMAGE	#00F6	Получить дескриптор изображения
BM_SETIMAGE	#00F7	Связать изображение с кнопкой

Прикладная программа может использовать сообщение BM_GETCHECK для того, чтобы определить состояние переключателя или селекторной кнопки. В ответ возвращается информация о состоянии селекторной кнопки или контрольного переключателя: BST_CHECKED, BST_INDETERMINATE или BST_UNCHECKED.

Предусмотрена группа сообщений, которая устанавливает состояние и стиль кнопки. Это сообщения BM_SETCHECK, BM_SETSTAT, BM_SETSTYLE. Для стилей BS_BITMAP и BS_ICON используются сообщения BM_SETIMAGE и BM_GETIMAGE. При установке состояния и стиля через параметр **wParam** передаются соответствующие константы.

В табл. 7.3 приведены значения констант, которые задают состояние кнопки.

Таблица 7.3. Константы, задающие состояние кнопки

Константа	*Значение*	*Описание*
BST_UNCHECKED	#0000	Кнопка не помечена
BST_CHECKED	#0001	Кнопка помечена
BST_INDETERMINATE	#0002	Состояние не определено, кнопка "серая"
BST_PUSHED	#0004	Кнопка нажата
BST_FOCUS	#0008	Кнопка имеет фокус клавиатуры

Для работы с кнопками в приложении вы можете также использовать сообщения DM_GETDEFID и DM_SETDEFID. Эти сообщения передают и устанавливаю идентификатор кнопки, которая приводится в действие по умолчанию клавишей ENTER.

Вызов функций **CheckDlgButton()** или **CheckRadioButton()** эквивалентен посылке сообщения BM_SETCHECK, а вызов функции **IsDlgButton Checked** – BM_GETCHECK.

Для взаимодействия с кнопками можно также использовать, так называемые *нотификационные сообщения*. Сообщения этого типа занимают особое место в процедуре взаимодействия приложения с окнами. Мы рассмотрим их более подробно позже, здесь же обсудим только наиболее важные аспекты.

Когда пользователь выбирает кнопку и изменяет ее состояние, кнопка посылает родительскому окну сообщение WM_COMMAND. Младшее слово параметра **wParam** содержит идентификатор управления, а старшее слово – код нотификационного сообщения. Например, кнопка посылает сообщение BN_CLICKED всякий раз, когда пользователь ее выбирает.

В табл. 7.4 приведены нотификационные сообщения кнопок, которые может обрабатывать прикладная программа.

Таблица 7.4. Нотификационные сообщения кнопок

Сообщение	*Значение*	*Описание*
BN_CLICKED	0	Пользователь нажал кнопку
BN_PAINT	1	Кнопка должна быть прорисована
BN_HILITE	2	Кнопка нажата
BN_PUSHED	2	BN_HILITE
BN_UNHILITE	3	Кнопка отпущена
BN_UNPUSHED	3	Bn_unhilite
BN_DISABLE	4	Кнопка заблокирована
BN_DOUBLECLICKED	5	Пользователь дважды нажал кнопку
BN_DBLCLK	5	Bn_doubleclicked
BN_SETFOCUS	6	Кнопка получила фокус клавиатуры
BN_KILLFOCUS	7	Кнопка потеряла фокус клавиатуры

Сообщения BN_SETFOCUS и BN_KILLFOCUS не определены в файле **Msfwinty.f90**.

Кнопка посылает сообщения BN_DISABLE, BN_PUSHED, BN_ KILLFOCUS, BN_PAINT, BN_SETFOCUS и BN_UNPUSHED, только если она имеет стиль BS_NOTIFY. Сообщения BN_CLICKED и BN_DBLCLK посылаются всегда.

Для автоматических кнопок Система выполняет графическую имитацию работы кнопки и прикладная программа обычно обрабатывает только сообщения BN_CLICKED и BN_DBLCLK. Если же стиль BS_AUTOCHECKBOX, BS_AUTO3STATE или BS_ AUTORADIOBUTTON не установлен, то прикладная программа обрабатывает сообщение и посылает команду на изменение состояния кнопки.

7.4. Включение кнопок в диалоговые функции

Обработка сообщений простой кнопки не представляет особой сложности. Поэтому займемся диалогом **ПАНЕЛЬ**, который содержит селекторные кнопки и контрольный переключатель.

При инициализации диалога необходимо инициализировать группы кнопок. Для этого используется функция **CheckRadioButton (hDlg, nIDFirstButton, nIDLastButton, nIDCheckButton)**. Значение первого пара-

метра вполне понятно. Второй и третий параметры, **nIDFirstButton** и **nIDLastButton** определяют идентификаторы первой и последней кнопок в группе. Последний параметр, **nIDCheckButton** – это идентификатор кнопки, которая будет включена после инициализации диалога. При обработке сообщения WM_COMMAND диалоговая процедура использует эту же функцию для переключения кнопок в группах.

Окончательные текст функции диалога **ПАНЕЛЬ** (рис. 7.1) выглядит следующим образом:

```
!*****************************************************************************
integer*4 function DialogFunc4(hDlg, message, wParam, lParam)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_DialogFunc4@16' :: DialogFunc4

integer hDlg, message, wParam, lParam
integer hWndFocus,lInitParam

hDialog = hDlg
select case(message)
 case (WM_INITDIALOG)
!---------- инициализация диалога ------------------
  hWndFocus = wParam
  lInitParam = lParam !на случай использования DialodBoxParam
  ir=CheckRadioButton(hDlg, IDC_RADIO1,IDC_RADIO4,IDC_RADIO1)
  ir=CheckRadioButton(hDlg, IDC_RADIO5,IDC_RADIO6,IDC_RADIO5)
  DialogFunc4 = 1
  return
 case (WM_COMMAND)
!---------- обработка команд ------------------------
 select case (MakeLong(LOWORD(wParam),0))
  case(IDC_RADIO1:IDC_RADIO4)
  ir=CheckRadioButton(hDlg, IDC_RADIO1,IDC_RADIO4, &
        LOWORD(wParam))
  case(IDC_RADIO5:IDC_RADIO6)
  ir=CheckRadioButton(hDlg, IDC_RADIO5,IDC_RADIO6, &
        LOWORD(wParam))
 end select
 case (WM_CLOSE)
 ir = DestroyWindow(hDlg)
end select
DialogFunc4=0
end function DialogFunc4
!*****************************************************************
```

Рис. 7.1. Диалог с группами селекторных кнопок

УРОК

8. Диалог со списком элементов

Список (List box) – это окно управления, которое содержит перечень элементов, из которых пользователь может выбрать необходимые ему. Элементы списка могут представлять собой текстовые строки, растровые изображения или и то и другое вместе. Наверное, в русскоязычном варианте можно было бы использовать термин *таблица*.

Если поле списка недостаточно для размещения всех элементов списка сразу, то окно дополняется линейкой вертикальной прокрутки.

Система Windows позволяет просматривать список, выбирать или удалять отдельные элементы. Выбор элемента списка обычно отмечается изменением цвета соответствующих строк. Когда пользователь выбирает элемент или удаляет его, Windows посылает сообщение родительскому окну (диалоговому окну) списка.

Процедура диалогового окна ответственна за инициализацию и текущий контроль за состоянием списка, связь с окном списка, пересылку ему сообщений и обработку нотификационных сообщений.

С каждым элементом списка можно связать некоторый объект в памяти. В таком случае элемент будет использовать адрес, для того чтобы обратиться к некоторой более обширной информации.

Списки могут быть двух видов: с одиночным выбором (стиль по умолчанию) и многоэлементным выбором. В окне списка с одиночным выбором пользователь одновременно может выбирать только один элемент, а в списке с многоэлементным – несколько.

8.1. Создание и инициализация списка

Для создания списка можно использовать функции **CreateWindow()** или **CreateWindowEx()**, объявляя класс окна **LISTBOX**.

Свойства списка определяются стилем окна. Наряду со стандартными стилями используются специальные, которые указывают, сортируются ли элементы списка, имеет ли список столбцы, возможен ли выбор несколь-

ких элементов одновременно и т. п. Все стили списков имеют префикс LBS_. В табл. 8.1 приведены стили окна списка.

Таблица 8.1. Стили окна списка

Стиль	*Значение*	*Описание*
LBS_NOTIFY	#0001	Разрешает посылку нотификационных сообщений
LBS_SORT	#0002	Строки сортируются по алфавиту
LBS_NOREDRAW	#0004	Список не перерисовывается
LBS_MULTIPLESEL	#0008	Разрешает множественный выбор для клавиш мыши
LBS_OWNERDRAWFIXED	#0010	Программист рисует элементы одинакового размера
LBS_OWNERDRAWVARIABLE	#0001	Программист рисует элементы различного размера
LBS_HASSTRINGS	#0040	Список имеет строки
LBS_USETABSTOPS	#0080	Список различает символ табуляции
LBS_NOINTEGRALHEIGHT	#0100	Система не может изменять размеры списка
LBS_MULTICOLUMN	#0200	Список содержит несколько колонок
LBS_WANTKEYBOARDINPUT	#0400	Разрешается обработка
LBS_EXTENDEDSEL	#0800	Множественный выбор для мыши и клавиатуры
LBS_DISABLENOSCROLL	#1000	Показывает заблокированную линейку прокрутки
LBS_NODATA	#2000	Устаревший стиль

Размеры и стили списка обычно определяются в шаблоне диалогового окна. Шаблон списка удобнее создавать средствами MDS. Добавим в диалог, который мы создавали ранее, список. Для этого откроем диалог **МОЙ_ДИАЛОГ1**. В процессе редактирования шаблона диалога будем использовать окно редактора ресурсов ***Controls***.

Выбираем в меню элемент ***List box*** и перемещаем его на поле диалога. Пока видны границы, устанавливаем нужные размеры. Двойным щелчком левой клавиши мыши открываем окно ***List box Properties***. Заменим иден-

тификатор, предлагаемый MDS, на IDC_LIST1. Установим стиль окна ***Visible***.

Переходим в окно ***Styles*** для того, чтобы установить стили списка. Это окно содержит несколько полей, значения которых описаны ниже.

Поле **Selection** определяет способ выбора элемента списка. Возможны следующие значения:

- *Single* (значение по умолчанию). Только один элемент списка может быть выбран;
- *Multiple* (стиль LBS_MULTIPLESEL). Несколько элементов списка могут быть выбраны с помощью мыши. Клавиши SHIFT и CTRL не используются. Выбор элемента происходит при щелчке или двойном щелчке клавиши мыши. Повторное действие отменяет выбор;
- *Extended* (стиль LBS_EXTENDEDSEL). Клавиши SHIFT и CTRL могут использоваться вместе с мышью для выбора и отмены выбора элементов списка или группы элементов, которая содержит несмежные поля.

Поле **Owner Draw** определяет взаимоотношения списка с родительским окном (т. е. с программистом). Можно выбрать одно из следующих значений:

- *No* (значение по умолчанию). Родительское окно не отвечает за прорисовку элементов. Поле списка содержит строки;
- *Fixed* (стиль LBS_OWNERDRAWFIXED). Родительское окно ответственно за прорисовку элементов списка. Все элементы списка имеют одинаковую высоту и могут быть не только текстовыми строками, но и рисунками. Родительское окно прорисовывает элементы при получении сообщения WM_DRAWITEM;
- *Variable* (стиль LBS_OWNERDRAWVARIABLE). Родительское окно отвечает за прорисовку элементов списка. Элементами могут быть не только текстовые строки, но и рисунки. Элементы могут иметь разную высоту. Родительское окно прорисовывает элементы при получении сообщения WM_DRAWITEM.

Поле **Has Strings** (стиль LBS_HASSTRINGS) устанавливает, что элементы списка это строки. Список выделяет память и указатели для строк, поэтому прикладная программа может использовать сообщение LB_GETTEXT. По умолчанию все поля списка имеют этот стиль.

Поле **Border** создает рамку вокруг поля списка.

Поле **Sort** (стиль LBS_SORT) устанавливает сортировку списка в алфавитном порядке.

Поле **Notify** (стиль LBS_NOTIFY) устанавливает режим передачи родительскому окну нотификационного сообщения о щелчке и двойном щелчке клавишей мыши.

Поле **Multicolumn** (стиль LBS_MULTICOLUMN) определяет список с несколькими колонками и линейкой горизонтальной прокрутки. Сообщение LB_SETCOLUMNWIDTH устанавливает ширину столбцов.

Поле **Horz Scroll** создает полосу горизонтальной прокрутки.

Поле **Vert Scroll** создает полосу вертикальной прокрутки.

Поле **No Redraw**(стиль LBS_NOREDRAW) устанавливает, что вид списка не изменяется даже тогда, когда производятся какие-то изменения.

Поле **Use Tabstops** (стиль LBS_USETABSTOPS) устанавливает возможность распознавания символов табуляции при изображении строк.

Поле **Want Key Input** (стиль LBS_WANTKEYBOARDINPUT) , получает ли родительское окно сообщение WM_VKEYTOITEM всякий раз, когда пользователь нажимает клавишу клавиатуры. Это позволяет прикладной программе выполнять специальную обработку ввода.

Поле **Disable No Scroll** (стиль LBS_DISABLENOSCROLL) показывает заблокированную линейку вертикальной прокрутки списка, если в списке недостаточное для прокрутки число элементов. Если этот стиль не установлен, то в этом случае линейка скрыта.

Поле **No Integral Height** (стиль LBS_NOINTEGRALHEIGHT) устанавливает, что размер списка точно равен размеру, указанному в прикладной программе. Система Windows не может изменить размеров окна.

Поле **No Data** (устаревший стиль LBS_NODATA). Если этот стиль установлен, то данные не сохраняются.

Поля, для которых не указаны специальные стили списка, устанавливают стандартные стили окна с префиксом WS_.

8.2. Взаимодействие диалога со списком

Процедура диалогового окна может посылать сообщения окну списка, чтобы добавлять, удалять, вставлять или изменять элементы списка. Например, процедура диалогового окна может посылать сообщение LB_ADDSTRING для того, чтобы добавить строку, или сообщение LB_GETSEL для того, чтобы определить, какая строка выбрана.

Взаимодействие родительского окна со списком осуществляется через функции **SendMessage (hWnd, MSG, wParam, lParam)** или **SendDlgItem Message (hDlg, nIDDlgItem, MSG, wParam, lParam)**. Прежде чем использовать первую из этих функций, необходимо определить дескриптор окна

списка с помощью функции **GetDlgItem (hDlg, nIDDlgItem)**. Параметр **nIDDlgItem** – это идентификатор списка.

Каждому элементу списка ставится в соответствие порядковый номер или индекс. Индекс – это целое число, которое указывает позицию строки в списке.

Внимание: порядковый номер первого элемента равен нулю.

В табл. 8.2 приведены сообщения, которые можно адресовать списку.

Таблица 8.2. Сообщения окну списка

Сообщение	*Значение*	*Описание*
LB_ADDSTRING	#0180	Добавляет строку и возвращает ее индекс
LB_INSERTSTRING	#0181	Вставляет в список после заданной строки новую строку
LB_DELETESTRING	#0182	Удаляет строку и возвращает число оставшихся строк
LB_SELITEMRANGEEX	#0183	Выделяет строки в заданном интервале индексов
LB_RESETCONTENT	#0184	Удаляет все элементы списка
LB_SETSEL	#0185	Выбирает строку списка с многострочным выбором
LB_SETCURSEL	#0186	Выбирает строку с заданным индексом
LB_GETSEL	#0187	Передает состояние элемента с заданным индексом
LB_GETCURSEL	#0188	Возвращает индекс выбранной строки
LB_GETTEXT	#0189	Загружает строку в буфер
LB_GETTEXTLEN	#018A	Возвращает длину строки для LB_GETTEXT
LB_GETCOUNT	#018B	Передает число элементов в списке
LB_SELECTSTRING	#018C	Выбирает строку, соответствующую шаблону
LB_DIR	#018D	Добавляет файл с заданными атрибутами

Сообщение	*Значение*	*Описание*
LB_GETTOPINDEX	#018E	Передает индекс первого видимого элемента
LB_FINDSTRING	#018F	Возвращает индекс строки, соответствующей шаблону
LB_GETSELCOUNT	#0190	Передает общее число выбранных элементов
LB_GETSELITEMS	#0191	Заполняет массив индексов выбранных элементов
LB_SETTABSTOPS	#0192	Устанавливает метку табуляции
LB_GETHORIZONTALEXTENT	#0193	Передает диапазон горизонтальной прокрутки
LB_SETHORIZONTALEXTENT	#0194	Устанавливает диапазон горизонтальной прокрутки
LB_SETCOLUMNWIDTH	#0195	Устанавливает ширину столбцов
LB_ADDFILE	#0196	Добавляет в список файл с заданным именем
LB_SETTOPINDEX	#0197	Помещает заданный элемент в верхнюю строку
LB_GETITEMRECT	#0198	Передает габариты текущего элемента
LB_GETITEMDATA	#0199	Передает 32-битовое число, связанное с элементом
LB_SETITEMDATA	#019A	Связывает с элементом 32-битовое число
LB_SELITEMRANGE	#019B	Отмечает последовательные элементы
LB_SETANCHORINDEX	#019C	Отмечает начальный элемент многострочного выбора
LB_GETANCHORINDEX	#019D	Передает индекс начального элемента
LB_SETCARETINDEX	#019E	Устанавливает каретку на заданный элемент
LB_GETCARETINDEX	#019F	Передает положение каретки
LB_SETITEMHEIGHT	#01A0	Устанавливает высоту элемента списка

Сообщение	*Значение*	*Описание*
LB_GETITEMHEIGHT	#01A1	Передает высоту элемента списка
LB_FINDSTRINGEXACT	#01A2	Находит строку, полностью согласующуюся с шаблоном
LB_SETLOCALE	#01A5	Устанавливает рабочий язык списка
LB_GETLOCALE	#01A6	Передает рабочий язык списка
LB_SETCOUNT	#01A7	Задает общее число элементов списка
LB_MSGMAX	#01A8	Максимальное значение сообщения
Сообщения, установленные для версий выше 4 и отсутствующие в файле Msfwinty.f90		
LB_INITSTORAGE	#01A8	Добавляет в список заданное число элементов и выделяет для каждого из них память
LB_ITEMFROMPOINT	#01A9	Передает индекс строки, ближайшей к заданной точке
LB_MSGMAX	#01B0	Максимальное значение сообщения
Сообщения, возвращаемые комбинированным списком		
LB_OKAY	0	Успешное выполнение
LB_ERR	-1	Сообщение об ошибке
LB_ERRSPACE	-2	Сообщение об ошибке

Сообщение LB_ADDSTRING добавляет в список строку, адрес которой указан в параметре **lParam**. Чтобы удалить строку, надо использовать сообщение LB_DELETESTRING, параметр **wParam** которого содержит ее номер.

Поиск строки производится с помощью сообщения LB_FINDSTRING, начиная с номера, указанного в параметре **wParam**. Адрес строки, которая содержит префикс, указывается в параметре **lParam**.

Строка списка с номером, равным значению параметра **wParam**, копируется в буфер сообщением LB_GETTEXT. Размер буфера, адрес которого задается параметром **lParam**, должен быть достаточным для размещения

строки. Чтобы это гарантировать, прикладная программа может сначала использовать сообщение LB_GETTEXTLEN.

На первых порах вам будет достаточно этих сообщений. Если понадобятся дополнительные сведения, то их можно найти в справочной системе MDS.

Кроме специализированных сообщений диалоговая процедура может обрабатывать стандартные сообщения. Наиболее важные из них:

- WM_LBUTTONDOWN – нажата левая клавиша мыши в рабочей области окна списка;
- WM_LBUTTONUP – левая клавиша мыши отпущена;
- WM_LBUTTONDBLCLK – двойной щелчок левой клавиши мыши.

Если после создания окна с помощью редактора ресурсов построить исполнительный модуль, то на экране вы увидите пустой список. Для того чтобы получилось то, что изображено на рис. 8.1, список надо инициализировать. Заполнение списка производится в ответ на сообщение WM_INITDIALOG.

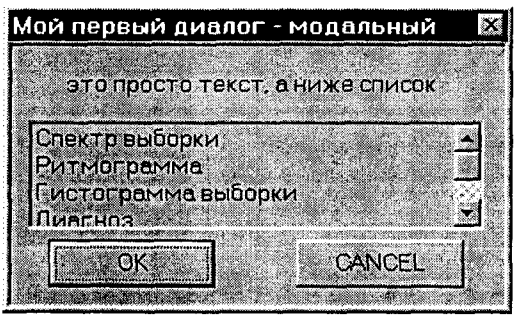

Рис. 8.1. Диалог со списком

Добавляем строки в список, посылая сообщение LB_ADDSTRING окну списка с идентификатором IDC_LIST1 с помощью функции **SendDlgItem Message()**. Дополнения, которые надо включить в текст функции **DialogFunc1()**, могут выглядеть следующим образом:

```
character(32), dimension(0:4), parameter :: szListItem = &
(/"Спектр выборки          "C, &
"Ритмограмма               "C, &
"Гистограмма выборки       "C, &
"Диагноз                   "C, &
"Медицинская карта         "C/)
```

```
case (WM_INITDIALOG)
!---------- инициализация диалога ----------------
  hWndFocus  = wParam
  lInitParam = lParam !на случай использования DialodBoxParam
!---------- заполним список ----------------------
  do i=0, 4
   ir= SendDlgItemMessage(hDlg, IDC_LIST1, &
         LB_ADDSTRING, 0, LOC(szListItem(i)))
  end do
  DialogFunc1 =1
!---------- список заполнен  -- ------------------------
```

В качестве демонстрации управления списком выберем нотификационное сообщение LBN_DBLCLK в роли инициатора процесса взаимодействия и LB_GETCURSEL в роли реакции на команду.

Нотификационные сообщения занимают особое место в процедуре взаимодействия приложения с окнами. Мы рассмотрим этот тип сообщений более подробно позже. Однако уже здесь будет уместно дать таблицу нотификационных сообщений окна списка (табл. 8.3):

Таблица 8.3. Нотификационные сообщения окна списка

Сообщение	*Значение*	*Описание*
LBN_ERRSPACE	-2	Не хватает памяти
LBN_SELCHANGE	1	Выделенным стал другой элемент
LBN_DBLCLK	2	Пользователь сделал двойной щелчок
LBN_SELCANCEL	3	Пользователь снял выделение элемента
LBN_SETFOCUS	4	Список получил фокус клавиатуры
LBN_KILLFOCUS	5	Список потерял фокус клавиатуры

Для того чтобы список мог посылать родительскому окну нотификационные сообщения, надо установить в поле **Notify** стиль LBS_NOTIFY. В этом случае старшее слово **wParam** сообщения WM_COMMAND будет содержать нотификационные сообщения.

Добавим в диалоговую процедуру функцию **SendDlgItemMessage (hDlg, IDC_LIST1, LB_GETCURSEL, 0, 0)**. В результате получим индекс выбранного элемента списка. Фрагмент текста приведен ниже.

```
!---------- выбор элемента списка ----------------------
case  (IDC_LIST1)
```

```
  if(HIWORD(wParam)==LBN_DBLCLK) then
   iStr= SendDlgItemMessage(hDlg, IDC_LIST1, &
                            LB_GETCURSEL, 0, 0)
   ir= MessageBox(hDlg, szListItem(iStr), "Выбор"C, MB_OK)
   end if
  DialogFunc1 =1
!---------- выбор элемента списка произведен ----------
```

8.3. Стандартный список

Функции Win32 API позволяют создать стандартный список файлов с заданными параметрами, содержащихся в текущей директории. Именно для этой цели предусмотрены сообщения LB_DIR и LB_ADDFILE.

В качестве параметра **wParam** в первом сообщении используются константы, задающие атрибуты файлов, численные значения и описание которых приведено в табл. 8.4. Параметр **lParam** – это указатель на строку, которая определяет, какие именно файлы должны быть добавлены в список. Строка формируется по правилам, которые достались нам в наследство от DOS. Так, например, для того чтобы отобрать все файлы в директории **Msdev** на диске **c:**, необходимо записать: **c:\ Msdev*.*C**.

Таблица 8.4. Атрибуты файлов, добавляемых в список

Константа	*Значение*	*Описание*
DDL_READWRITE	#0000	Включать только файлы, доступные для чтения и записи
DDL_READONLY	#0001	Включать только файлы, доступные для чтения
DDL_HIDDEN	#0002	Включать скрытые файлы
DDL_SYSTEM	#0004	Включать системные файлы
DDL_DIRECTORY	#0010	Включать поддиректории
DDL_ARCHIVE	#0020	Включать архивные файлы
DDL_DRIVES	#4000	Включать в список имена дисководов
DDL_POSTMSGS	#2000	Поместить сообщения в очередь
DDL_EXCLUSIVE	#8000	Включать файлы только с указанными атрибутами

Сообщение LB_ADDFILE добавляет файл в список. **lParam** – это параметр, указывающий на строку, которая определяет, какой именно файл должен быть добавлен. Список предварительно заполняется функцией **DlgDirList()**. Интерфейс этой функции задан в файле **Mswinty.f90** следующим образом:

```
interface
integer(4) function  DlgDirList (hDlg, lpPathSpec, nIDListBox, nIDStaticPath,
uFileType)
!MS$ATTRIBUTES STDCALL, ALIAS : '_DlgDirListA@20' :: DlgDirList
!MS$ATTRIBUTES REFERENCE :: lpPathSpec
integer     hDlg              – дескриптор диалога,
character*(*)   lpPathSpec    – строка с текстом, задающим файл,
integer   nIDListBox          – идентификатор списка,
integer   nIDStaticPath       – идентификатор статического элемента
                                диалога,
integer   uFileType           – атрибуты файлов,
end function DlgDirList
end interface.
```

Идентификатор статического элемента используется, если вы ходите вывести в окно диалога информацию о директории. Атрибуты файлов приведены в табл. 8.4. Назначение остальных параметров функции очевидны.

Для того чтобы заполнить список файлов, к диалоговой процедуре следует добавить следующий текст:

```
case (WM_INITDIALOG)
!---------- инициализация диалога ----------------
  hWndFocus  = wParam
  lInitParam = lParam !на случай использования DialodBoxParam
  buffer = 'C:\users\Shtykov\MyPr_1\*.*'C
  ir =    DlgDirList(hDlg, buffer, IDC_LIST1, 0, DDL_READWRITE)
  DialogFunc2 =1
```

Результат выполнения программы показан на рис. 8.2.

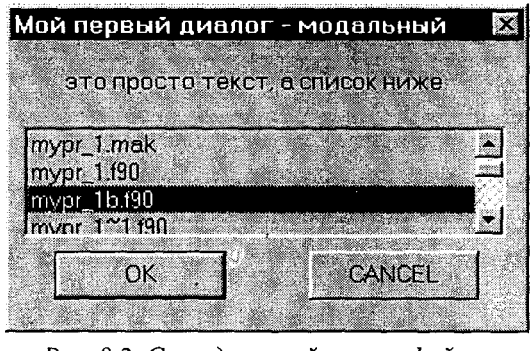

Рис. 8.2. Стандартный список файлов

УРОК

9. Диалог с окном редактирования

Окно редактирования (Edit Box), или ***ввода*** – это прямоугольное окно, обычно применяемое в диалоговом окне, в котором пользователь может вводить и редактировать текст с помощью клавиатуры. В поле окна выводятся текст (если он есть) и изображение мерцающей каретки. Пользователь может вводить текст, перемещать каретку или выделять нужный фрагмент текста для последующего его перемещения, копирования в буфер или удаления.

Окна ввода могут быть однострочными и многострочными. Однострочные используются для ввода данных. Многострочное окно фактически представляет собой простейший текстовой редактор.

9.1. Создание окна редактирования

Окно редактирования можно создать как дочернее окно с помощью функций **CreateWindow()** или **CreateWindowEx()** или как ресурс. В первом случае необходимо объявить класс **EDIT** и определить явным образом стиль окна ввода.

Система Windows поддерживает несколько стилей окон редактирования. Стиль конкретного окна может быть комбинацией нескольких стилей. Стили задают внешний вид и особенности взаимодействия с окном. Перечень стилей окна ввода содержится в табл. 9.1.

Таблица 9.1. Стили окна ввода

Стиль	*Значение*	*Описание*
ES_LEFT	#0000	Выравнивание по левому краю
ES_CENTER	#0001	Выравнивание по центру
ES_RIGHT	#0002	Выравнивание по правому краю
ES_MULTILINE	#0004	Многострочное окно ввода
ES_UPPERCASE	#0008	Вводится текст прописными буквами

Стиль	Значение	Описание
ES_LOWERCASE	#0010	Вводится текст строчными буквами
ES_PASSWORD	#0020	Все вводимые символы отображаются в виде звездочек (*)
ES_AUTOVSCROLL	#0040	Автоматическая прокрутка текста по вертикали
ES_AUTOHSCROLL	#0080	То же по горизонтали
ES_NOHIDESEL	#0100	Выделение текста сохраняется
ES_OEMCONVERT	#0400	Символы преобразуются из одного набора в другой
ES_READONLY	#0800	Редактирование текста запрещено
ES_WANTRETURN	#1000	При нажатии Enter добавляется возврат каретки
ES_NUMBER	#2000	Разрешается ввод только цифр

> Внимание: стиль ES_NUMBER отсутствует в файле **Msfwinty.f90_**. Поэтому его надо объявить в вашем приложении, если вы не собираетесь использовать редактор ресурсов.

Для каждого окна редактирования устанавливается комбинация значений, приведенных в табл. 9.1.

Имеются два стиля, которые задают тип текста в окне. По умолчанию окно имеет одну строку, но прикладная программа может создать окно редактирования с множеством строк, используя стиль ES_ MULTILINE.

Предусмотрены три стиля, которые задают способ выравнивания текста на странице: ES_LEFT, ES_CENTER и ES_RIGHT.

Задавая стили ES_LOWERCASE и ES_UPPERCASE, можно изменять регистр символов. В некоторых случаях требуется преобразовывать текст (например, имя файла) в специфический машинно зависимый набор символов. Стиль ES_OEMCONVERT гарантирует соответствующее преобразование.

Когда размер текста превышает размер окна, прикладная программа может использовать два стиля: ES_AUTOHSCROLL – для горизонтальной прокрутки – и ES_AUTOVSCROLL – для вертикальной.

Другие стили определяют различные аспекты управления окном редактирования. При необходимости вы можете их использовать в ваших приложениях.

Большинство разработчиков предпочитает использовать инструментальные средства MDS. Это во всех отношениях более удобный способ, в частности и потому, что не надо явно задавать стиль окна.

Если вы внимательно прочитали предыдущие уроки и самостоятельно создали окно списка, то вам не составит большого труда создать шаблон диалога с окном ввода.

Добавим в диалог **МОЙ_ДИАЛОГ2**, который мы создавали ранее, окно ввода. В процессе редактирования шаблона диалога будем использовать окно редактора ресурсов ***Controls***.

Выбираем в меню элемент ***Edit box*** и перемещаем его на поле диалога. Пока видны границы, устанавливаем нужные размеры. Двойным щелчком левой клавиши мыши открываем окно ***Edit box Properties***. Многие поля этого окна вам уже знакомы.

Заменим идентификатор, предлагаемый MDS, на IDC_EDIT1. Установим стиль окна ***Visible***.

Переходим в окно ***Styles*** для того, чтобы установить стили окна ввода. Окно ***Styles*** содержит несколько полей. Связь полей с табл. 9.1 вполне очевидна и не требует каких-то дополнительных пояснений. В нашем примере построим однострочное окно ввода. Для того чтобы выделить окно ввода, установим в поле **Border** стиль WS_BORDER. Окно ввода будем использовать для ввода численного значения некоторого параметра, например номинала резистора. Поэтому в поле **Number** установим стиль ES_NUMBER

Теперь придадим диалогу более законченный вид, добавляя статические элементы с текстами, например *резистор:*, и список с размерностями резистора, например *Ом, кОм, МОм* и т. д.

Шаблон ресурса готов, и теперь можно заняться диалоговой функцией **DialogFunc3**.

9.2. Взаимодействие окна ввода с пользователем

Управление окном ввода осуществляется посредством посылки сообщений. Оно может посылать сообщения родительскому окну в форме WM_COMMAND, а родительское окно может использовать функцию **SendDlgItemMessage()**.

В нашем примере окно ввода имеет всего одну строку и используется только для ввода номинала резистора. Тем не менее хотя бы кратко рассмотрим взаимодействие пользователя с многострочным окном ввода – окном редактирования.

Наибольший интерес представляют сообщения, которые обслуживают взаимодействие с буфером обмена. Следующие четыре сообщения не имеют параметров и предназначены для перемещения текста из окна редактирования в буфер обмена и обратно:

- сообщение WM_COPY копирует текущий фрагмент из окна редактирования в буфер обмена;
- сообщение WM_CUT удаляет текущий фрагмент и помещает его в буфер обмена;
- сообщение WM_CLEAR удаляет фрагмент;
- сообщение WM_PASTE копирует текст из буфера обмена в окно редактирования в режиме вставки.

Начиная с версии 4.0 в окне редактирования предусмотрено встроенное контекстное меню. Это меню появляется, когда пользователь применяет правую кнопку мыши. Меню включает пять пунктов: ***Отменить***, ***Вырезать***, ***Копировать***, ***Вставить***, ***Удалить*** и ***Выделить все***.

Для отмены последней операции в окне ввода необходимо использовать сообщение WM_UNDO.

Система Windows предоставляет пользователю обширный набор специальных сообщений для работы с окном редактирования. Все они имеют префикс EM_. Значения сообщений приведены в табл. 9.2. Некоторые из этих сообщений описаны ниже более подробно.

Таблица 9.2. Сообщения окна редактирования

Сообщение	*Значение*	*Описание*
EM_GETSEL	#00B0	Передает позиции выделенного текста
EM_SETSEL	#00B1	Устанавливает позиции выделенного текста
EM_GETRECT	#00B2	Передает размеры текстового поля
EM_SETRECT	#00B3	Устанавливает размеры текстового поля
EM_SETRECTNP	#00B4	Устанавливает размеры текста
EM_SCROLL	#00B5	Прокручивает текст по вертикали
EM_LINESCROLL	#00B6	Прокручивает текст по вертикали или горизонтали

Сообщение	*Значение*	*Описание*
EM_SCROLLCARET	#00B7	Продвижение каретки
EM_GETMODIFY	#00B8	Определяет, должно ли модифицироваться содержимое окна редактирования
EM_SETMODIFY	#00B9	Устанавливает, должно ли модифицироваться содержимое окна редактирования
EM_GETLINECOUNT	#00BA	Передает число строк
EM_LINEINDEX	#00BB	Передает число символа до заданной строки
EM_SETHANDLE	#00BC	Устанавливает дескриптор текстового буфера
EM_GETHANDLE	#00BD	Передает дескриптор текстового буфера
EM_LINELENGTH	#00C1	Передает длину заданной строки
EM_REPLACESEL	#00C2	Заменить выделенный текст
EM_GETLINE	#00C4	Копирует строки текста и помещает их в буфер
EM_LIMITTEXT	#00C5	Передает предельное число символов в окне
EM_GETLIMITTEXT	#00C5	EM_LIMITTEXT
EM_CANUNDO	#00C6	Разрешает обработку сообщения EM_UNDO
EM_UNDO	#00C7	Возврат
EM_FMTLINES	#00C8	Устанавливает формат строки с автоматическим разбиением текста на строки
EM_LINEFROMCHAR	#00C9	Передает номер строки, содержащей символ заданного номера
EM_SETTABSTOPS	#00CB	Устанавливает знаки табуляции
EM_SETPASSWORDCHAR	#00CC	Устанавливает пароль
EM_EMPTYUNDOBUFFER	#00CD	Обнуляет флаг буфера EM_UNDO
EM_GETFIRSTVISIBLELINE	#00CE	Передает номер самой верхней строки в окне
EM_SETREADONLY	#00CF	Запрещает изменение текста

Сообщение	*Значение*	*Описание*
EM_SETWORDBREAKPROC	#00D0	Устанавливает новую функцию разбиения текста на строки
EM_GETWORDBREAKPROC	#00D1	Передает адрес функции разбиения строк
EM_GETPASSWORDCHAR	#00D2	Передает пароль, введенный пользователем
EM_SETMARGINS	#00D3	Устанавливает границы текста
EM_GETMARGINS	#00D4	Передает границы строки текста
EM_SETLIMITTEXT	#00D5	Устанавливает предельное число символов
EM_POSFROMCHAR	#00D6	Возвращает координату символа заданного номера
EM_CHARFROMPOS	#00D7	Передает номер строки и номер символа, ближайших к заданной точке x, y

Внимание: последние пять сообщений не определены в файле **Msfwin.f90**.

Для того чтобы получить границы выделенного текста, необходимо с помощью функций **SendMessage (hWnd, MSG, wParam, lParam)** или **SendDlgItemMessage (hDlg, nIDDlgItem, MSG, wParam, lParam)** передать окну редактирования сообщение EM_GETSEL. Младшее слово возвращаемого значения содержит начальную позицию фрагмента, а старшее – увеличенную на единицу конечную. Сообщение EM_SETSEL выделяет фрагмент текста. Параметр **lParam** указывает начальную и конечную позиции.

Сообщение EM_REPLACESEL заменяет старый фрагмент текста на новый. Параметр **lParam** является адресом буфера, который содержит новый текст.

Копирование строки в собственный буфер можно осуществить, посылая сообщение EM_GETLINE. В этом сообщении параметр **wParam** определяет номер строки, а параметр **lParam** – это адрес буфера.

Полный перечень сообщений можно найти в справочном тексте.

Для обработки действия с клавиатурой и мышью можно использовать нотификационные сообщения. Мы уже рассматривали подобные сообщения. Их перечень приведен в табл. 9.3.

Таблица 9.3. Нотификационные сообщения окна ввода

Сообщение	*Значение*	*Описание*
EN_SETFOCUS	#0100	Окно ввода получило фокус
EN_KILLFOCUS	#0200	Окно ввода потеряло фокус
EN_CHANGE	#0300	Пользователь изменил текст
EN_UPDATE	#0400	Система будет обновлять текст
EN_ERRSPACE	#0500	Не хватает памяти
EN_MAXTEXT	#0501	Превышен предел и вставка будет урезана
EN_HSCROLL	#0601	Нажата горизонтальная прокрутка
EN_VSCROLL	#0602	Нажата вертикальная прокрутка

Теперь модифицируем диалоговую функцию. При инициализации заполним список размерностей резистора. В окно ввода для определенности занесем значение 001.0. Окончательный вид диалоговой функции приведен ниже.

```
integer*4 function DialogFunc3(hDlg, message, wParam, lParam)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_DialogFunc3@16' :: DialogFunc3

integer hDlg, message, wParam, lParam
integer hWndFocus, lInitParam, iStr
integer iEnd

character(5), dimension(0:4), parameter :: szListItem = &
          (/"мкОм"C,  &
            "мОм "C,  &
            "Ом  "C,  &
            "КОм "C,  &
            "МОм "C/)
 character(6)                  :: Resistor
 character(16)                 :: Text

hDialog = hDlg

select case(message)
 case (WM_INITDIALOG)
!---------- инициализация диалога ----------
  hWndFocus  = wParam
```

```
  lInitParam = lParam !на случай использования DialodBoxParam
  Resistor="001.0"C
  ir=SetDlgItemText(hDlg, IDC_EDIT1, Resistor)
  do i=0, 4
   ir= SendDlgItemMessage(hDlg, IDC_LIST2, &
         LB_ADDSTRING, 0, LOC(szListItem(i)))
  end do
  ir= SendDlgItemMessage(hDlg, IDC_LIST2, &
                    LB_SETCURSEL, 2, 0)
  DialogFunc3 =1
 case (WM_COMMAND)
!--------- обработка команд ----------------------
  select case (MakeLong(LOWORD(wParam), 0))
   case (IDOK)
    iStr= SendDlgItemMessage(hDlg, IDC_LIST2, &
                    LB_GETCURSEL, 0, 0)
    ir=GetDlgItemText(hDlg, IDC_EDIT1, Resistor, 6)
     iEnd= INDEX(Resistor, NullChar)-1
     Text=Resistor(1:iEnd)//" "//szListItem(iStr)
     ir= MessageBox(hDlg, Text, "Выбор"C, MB_OK)
!    ir = DestroyWindow(hDlg)
   case (IDCANCEL)
    ir = DestroyWindow(hDlg)
  end select
 case (WM_CLOSE)
  ir = DestroyWindow(hDlg)
 case DEFAULT
  DialogFunc3 = 0
  return
end select
DialogFunc3 =1
end function DialogFunc3
!*****************************************************************************
```

При обработке команд в ответ на **IDOK** определяем номер строки списка размерностей и переносим текст из окна ввода в буфер. Результат этих действий выводим на экран с помощью функции **MessageBox()**. Ясно, что средствами Фортрана можно распорядиться полученными данными с большей пользой. Можно, например, преобразовать текст номинала и номер строки списка размерности в численное значение сопротивления резистора. Если бы использовались только целые числа в окне ввода, то удобно было бы применить функцию **GetDlgItemInt()**, которая возвращает значение целого числа в окне ввода.

После создания исполняемого модуля, его запуска и выбора соответствующего пункта главного меню на экране получилось окно ввода, изображенное на рис. 9.1. Теперь в окно можно ввести номинал резистора, а с помощью линейки прокрутки выбрать размерность.

Рис. 9.1. Диалог с окном ввода

В качестве упражнения попробуйте создать небольшой текстовой редактор.

УРОК

10. Диалог с комбинированным списком

Комбинированный список (Combo box) – это уникальный элемент управления, который объединяет функциональные возможности списка и окна редактирования. Он состоит из собственно списка и поля выбора, содержание которого можно редактировать.

Список – это часть, содержащая элементы, которые пользователь может выбирать. Обычно прикладная программа заполняет список при инициализации диалога.

В поле выбора автоматически отображается элемент – единица списка, выбранный пользователем. Кроме того, имеется возможность ввести новый текст в поле выбора. С текущим выбором связан номер выбранного элемента списка.

Программный интерфейс Win32 API поддерживает три типа комбинированных списков: простые с раскрытым списком, редактируемые (стиль CBS_SIMPLE); с раскрывающимся списком и с редактированием (стиль CBS_DROPDOWN); с раскрывающимся списком, но без редактирования (стиль CBS_DROPDOWNLIST). Табл. 10.1 дает представление о возможностях каждого из типов.

Таблица 10.1. Функции комбинированных списков

Тип – стиль	*Список*	*Редактирование*
CBS_SIMPLE	Постоянно раскрыт	Да
CBS_DROPDOWN	Раскрывается	Да
CBS_DROPDOWNLIST	Раскрывается	Нет

10.1. Создание комбинированного списка

Комбинированный список – это окно класса **COMBOBOX**. Для создания такого окна используются функции **CreateWindow()** и **CreateWindowEx()**. Как всякое окно, комбинированный список требует указания стиля. Все специфические стили имеют префикс CBS_. Значения

стилей приведены в табл. 10.2. Более полные толкования некоторых стилей приведены после таблицы.

Таблица 10.2. Стили комбинированного списка

Стиль	*Значение*	*Описание*
CBS_SIMPLE	#0001	Простой список с редактированием
CBS_DROPDOWN	#0002	То же, но с раскрывающимся списком
CBS_DROPDOWNLIST	#0003	То же, но без редактирования
CBS_OWNERDRAWFIXED	#0010	Программист рисует элементы равной высоты
CBS_OWNERDRAWVARIABLE	#0020	То же, но различной высоты
CBS_AUTOHSCROLL	#0040	Автоматически продвигает текст в окне редактирования
CBS_OEMCONVERT	#0080	Преобразует символы
CBS_SORT	#0100	Автоматически сортирует список
CBS_HASSTRINGS	#0200	Список содержит строки
CBS_NOINTEGRALHEIGHT	#0400	Размер окна определяется приложением
CBS_DISABLENOSCROLL	#0800	Показывает заблокированную линейку прокрутки
CBS_UPPERCASE	#2000	Преобразует текст в верхний регистр
CBS_LOWERCASE	#4000	Преобразует текст в нижний регистр

Внимание: стили CBS_UPPERCASE и CBS_LOWERCASE отсутствуют в файле **Msfwinty.f90**. Поэтому, если вы не собираетесь использовать редактор ресурсов, их надо объявить в вашем приложении.

О стилях, которые определяют тип комбинированного списка, мы уже говорили выше.

Имеется ряд стилей, которые определяют специфические свойства комбинированного списка. Например, два стиля дают возможность программисту создавать свой собственный комбинированный список. Стиль

CBS_OWNERDRAWFIXED определяет, что владелец окна списка ответственен за прорисовку элементов и что все элементы имеют равную высоту. Родительское окно получает сообщение WM_MEASUREITEM, когда список уже создан, и сообщение WM_DRAWITEM, если внешний вид окна изменился. Стиль CBS_OWNERDRAWVARIABLE отличается только тем, что элементы списка могут иметь различную высоту.

Стиль CBS_DISABLENOSCROLL устанавливает, что окно списка всегда имеет линейку вертикальной прокрутки в окне списка, которая блокируется, если число элементов недостаточно для прокрутки. Если этот стиль не объявлен, то линейка прокрутки появляется только в случае необходимости.

Стиль CBS_NOINTEGRALHEIGHT устанавливает, что размер окна определяется прикладной программой при инициализации диалога. Обычно размеры устанавливаются так, чтобы отображались только полные строки.

Стиль CBS_OEMCONVERT необходим для преобразования набора символов системы Windows в набор машинно зависимых символов (OEM) и обратно.

Поскольку для создания шаблона ресурса мы используем средства MDS, то нет смысла приводить полный перечень стилей. При необходимости со стилями комбинированных списков можно ознакомиться в справочной системе MDS.

Для демонстрации комбинированного списка добавим этот элемент в диалог **ПАНЕЛЬ**, используя меню редактора ресурсов.

В окне ***General*** определим идентификатор элемента управления IDC_COMBO1 и стиль окна ***Visible***. В этом окне есть поле **Enter listbox items**. В него можно ввести строки списка. Однако это дополнительное удобство доступно, только если поддерживается библиотека фундаментальных классов Microsoft (Microsoft Foundation Class Library – MFC). Поэтому список придется заполнить при инициализации диалога.

Переходим в окно ***Styles***, для того чтобы указать стили списка. Это окно содержит несколько полей. Новым для нас является поле **Type**. Для нашего диалога выберем стиль CBS_DROPDOWNLIST и установим в поле **Type** тип ***Drop List***. Установка остальных полей не должна вызвать у вас затруднений.

10.2. Управление комбинированным списком

Прикладная программа управляет комбинированным списком посредством сообщений, которые имеют префикс CB_. Сообщения, управляю-

щие окном списка, имеют аналоги среди сообщений, о которых шла речь в уроке 8. В табл. 10.3 приведены сообщения, адресованные комбинированному списку.

Таблица. 10.3. Сообщения комбинированного списка

Сообщение	*Значение*	*Описание*
CB_GETEDITSEL	#0140	Передает начальную и конечную позиции выделения
CB_LIMITTEXT	#0141	Устанавливает максимальную длину строки
CB_SETEDITSEL	#0142	Выделяет текст между заданными позициями
CB_ADDSTRING	#0143	Добавляет строку в список
CB_DELETESTRING	#0144	Удаляет строку с заданным номером
CB_DIR	#0145	Добавляет имя файла в список
CB_GETCOUNT	#0146	Передает число элементов в списке
CB_GETCURSEL	#0147	Передает номер выбранного элемента
CB_GETLBTEXT	#0148	Передает строку текста заданного элемента списка
CB_GETLBTEXTLEN	#0149	Передает длину строки выбранного элемента
CB_INSERTSTRING	#014A	Вставляет строку в список
CB_RESETCONTENT	#014B	Очищает список и окно редактирования
CB_FINDSTRING	#014C	Находит строку, содержащую заданные символы
CB_SELECTSTRING	#014D	Выделяет строку, содержащую заданные символы
CB_SETCURSEL	#014E	Выделяет строку с заданным номером
CB_SHOWDROPDOWN	#014F	Раскрывает и сворачивает список
CB_GETITEMDATA	#0150	Передает 32-битовое число, связанное с элементом

Сообщение	*Значение*	*Описание*
CB_SETITEMDATA	#0151	Связывает 32-битовое число с элементом списка
CB_GETDROPPEDCONTROLRECT	#0152	Передает координаты окна списка
CB_SETITEMHEIGHT	#0153	Устанавливает высоту элемента списка
CB_GETITEMHEIGHT	#0154	Передает высоту элемента списка
CB_SETEXTENDEDUI	#0155	Устанавливает расширенный интерфейс
CB_GETEXTENDEDUI	#0156	Проверяет тип интерфейса
CB_GETDROPPEDSTATE	#0157	Определяет состояние окна списка
CB_FINDSTRINGEXACT	#0158	Находит строку, точно совпадающую с шаблоном
CB_SETLOCALE	#0159	Устанавливает рабочий язык списка
CB_GETLOCALE	#015A	Передает рабочий язык списка
CB_MSGMAX	#015B	См. табл. 8.2
Сообщения установленные для версий выше 4 и отсутствующие в файле Msfwinty.f90		
CB_GETTOPINDEX	#015B	Передает индекс первого видимого элемента
CB_SETTOPINDEX	#015C	Помещает заданный элемент в верхнюю строку окна
CB_GETHORIZONTALEXTENT	#015D	Передает диапазон горизонтальной прокрутки
CB_SETHORIZONTALEXTENT	#015E	Устанавливает диапазон горизонтальной прокрутки
CB_GETDROPPEDWIDTH	#015F	Передает ширину окна списка
CB_SETDROPPEDWIDTH	#0160	Устанавливает ширину окна списка

Сообщение	*Значение*	*Описание*
CB_INITSTORAGE	#0161	Добавляет в список заданное число элементов и выделяет для каждого из них память
CB_MSGMAX	#0162	См. табл. 8.2
Сообщения, возвращаемые комбинированным списком		
CB_OKAY	0	См. табл. 8.2
CB_ERR	-1	См. табл. 8.2
CB_ERRSPACE	-2	См. табл. 8.2

Прикладная программа дополняет список, посылая сообщение CB_ADDSTRING. Это обязательно делается на стадии инициализации диалога с комбинированным списком. Строка добавляется в конец списка, и список заново сортируется. В этом случае по сути посылается сообщение LB_ADDSTRING. Адрес строки устанавливается параметром **lParam**. Можно использовать сообщение CB_INSERTSTRING для того, чтобы вставить строку в позицию, которая задается параметром **wParam**. Сообщение CB_DELETESTRING эквивалентно сообщению LB_ DELETESTRING, которое удаляет элемент списка. Параметр этого сообщения **wParam** задает номер удаляемой строки.

Используя сообщения CB_FINDSTRING или CB_FINDSTRINGEXACT, прикладная программа может определять позицию строки в списке. Первое из них находит строку текста, которая начинается с определенного набора символов, а второе находит строку, которая точно совпадает с заданной. Параметр **wParam** этих сообщений – номер строки, с которой начинается поиск.

Если требуется повторная инициализация списка, то сначала следует удалить все содержимое, используя сообщение CB_RESETCONTENT.

Когда список создан, ни один из его элементов не выбран. Это верно и для простого, и для раскрывающегося списка. Для выбора элемента прикладная программа посылает сообщение CB_SETCURSEL. Можно также использовать сообщение CB_SELECTSTRING для выбора элемента списка по заданному шаблону. Определить, какой именно элемент в данный момент выбран, можно, посылая сообщение CB_GETCURSEL. Если выбор не сделан, то возвращается значение CB_ERR.

Строка списка с номером, равным значению параметра **wParam**, копируется в буфер сообщением CB_GETLBTEXT. Размер буфера, адрес которого задается параметром **lParam**, должен быть достаточным для размещения строки. Чтобы это гарантировать, прикладная программа может сначала использовать сообщение CB_GETLBTEXTLEN.

Раскрывающиеся комбинированные списки поддерживают альтернативный интерфейс клавиатуры, называемый *расширенным интерфейсом пользователя*. По умолчанию клавиша F4 открывает или закрывает список, а СТРЕЛКА "ВНИЗ" изменяет текущий выбор. В списке с расширенным интерфейсом пользователя клавиша F4 заблокирована. Вместо нее действует клавиша СТРЕЛКА "ВНИЗ". Расширенный интерфейс устанавливается сообщением CB_SETEXTENDEDUI (параметр **wParam** = **.TRUE.**).

Сведения об остальных сообщениях вы сможете получить из справочной системы по мере необходимости.

Кроме специализированных сообщений диалоговая процедура может обрабатывать стандартные сообщения. Наиболее важные из них:

- WM_COPY – копирует содержимое окна редактирования в буфер обмена;
- WM_CUT – удаляет содержимое окна редактирования и помещает в буфер обмена;
- WM_CLEAR – очищает окно редактирования;
- WM_PASTE – помещает содержимое буфера обмена в окно редактирования;
- WM_LBUTTONDOWN – нажата левая клавиша мыши в рабочей области окна списка;
- WM_LBUTTONUP - левая клавиша мыши отпущена;
- WM_LBUTTONDBLCLK – двойной щелчок левой клавиши мыши.

Сообщение WM_COMMAND заслуживает особого внимания, поскольку старшее слово параметра **wParam** содержит код нотификационного сообщения. Все такие сообщения имеют префикс CBN_ Перечень нотификационных сообщений приведен в табл. 10.4.

Таблица 10.4. Нотификационные сообщения окна комбинированного списка

Сообщение	*Значение*	*Описание*
CBN_ERRSPACE	-1	Не хватает памяти
CBN_SELCHANGE	1	Выделенным стал другой элемент
CBN_DBLCLK	2	Пользователь сделал двойной щелчок
CBN_SETFOCUS	3	Список получил фокус клавиатуры
CBN_KILLFOCUS	4	Список потерял фокус клавиатуры
CBN_EDITCHANGE	5	В поле редактирования введен новый текст
CBN_EDITUPDATE	6	В поле редактирования будет изменен текст
CBN_DROPDOWN	7	Список будет открыт
CBN_CLOSEUP	8	Список будет закрыт
CBN_SELENDOK	9	Выбор сделан, и список закрывается
CBN_SELENDCANCEL	10	Выбор сделан, но диалог закрывается

Когда пользователь выбирает новый элемент списка, родительское окно или диалоговая процедура получают сообщение CBN_SELCHANGE. В результате прикладная программа может выполнить некоторую специфическую операцию. Это сообщение не посылается, когда выбор производится с помощью сообщения CB_SETCURSEL.

Если список открывается, то передается сообщение CBN_ DROPDOWN. Когда список закрывается, передается сообщение CBN_ CLOSEUP.

В простом комбинированном списке система посылает сообщение CBN_DBLCLK, когда пользователь дважды нажимает клавишу мыши. В раскрывающемся списке одиночный щелчок закрывает список, поэтому невозможно дважды нажать клавишу на строке списка.

Система посылает нотификационные сообщения CBN_SELENDOK и CBN_SELENDCANCEL, когда пользователь, выбрав элемент списка, закрывает список с помощью кнопки OK или CANCEL. В первом случае выбор должен быть обработан, а во втором – проигнорирован. Система посылает эти сообщения, только если установлен стиль WS_EX_ NOPARENTNOTIFY.

10.3. Подключение диалога

Окончательный текст диалоговой функции приведен ниже. Дополнения в диалоговую функцию, связанные с комбинированным списком, выделены.

```
!*******************************************************************************
integer*4 function DialogFunc4(hDlg, message, wParam, lParam)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_DialogFunc4@16' :: DialogFunc4

integer hDlg, message, wParam, lParam
integer hWndFocus, lInitParam

character(18), dimension(0:4), parameter :: szListItem = &
        (/"Маяк              "C,    &
        "Радио ретро         "C,    &
        "Серебряный дождь    "C,    &
        "Европа плюс         "C,    &
        "Наше радио          "C/)

hDialog = hDlg
select case(message)
 case (WM_INITDIALOG)
!---------- инициализация диалога ----------
  hWndFocus  = wParam
  lInitParam = lParam !на случай использования DialodBoxParam
  ir=CheckRadioButton(hDlg, IDC_RADIO1, IDC_RADIO4, IDC_RADIO1)
  ir=CheckRadioButton(hDlg, IDC_RADIO5, IDC_RADIO6, IDC_RADIO5)
  do i=0, 4
   ir= SendDlgItemMessage(hDlg, IDC_COMBO1, &
        CB_ADDSTRING, 0, LOC(szListItem(i)))
  end do
  ir= SendDlgItemMessage(hDlg, IDC_COMBO1, &
        CB_SETCURSEL, 0, 0)
  DialogFunc4 = 1
  return
 case (WM_COMMAND)
!---------- обработка команд ----------
  select case (MakeLong(LOWORD(wParam), 0))
   case(IDC_RADIO1:IDC_RADIO4)
    ir=CheckRadioButton(hDlg, IDC_RADIO1, IDC_RADIO4, &
                LOWORD(wParam))
   case(IDC_RADIO5:IDC_RADIO6)
    ir=CheckRadioButton(hDlg, IDC_RADIO5, IDC_RADIO6, &
                LOWORD(wParam))
```

```
case(IDC_COMBO1)
   if(HIWORD(wParam)==LBN_DBLCLK) then
    ir= SendDlgItemMessage(hDlg, IDC_COMBO1, &
                  CB_GETCURSEL, 0, 0)
    ir= MessageBox(hDlg, szListItem(ir), "Выбор"C, MB_OK)
   end if
 end select
 case (WM_CLOSE)
  ir = DestroyWindow(hDlg)
end select
DialogFunc4=0
end function DialogFunc4
!*****************************************************************
```

При инициализации диалога заполняется список, из которого будет осуществляться выбор. Затем устанавливается начальное состояние списка. Для этого используется сообщение CB_SETCURSEL, параметр которого задает номер выбранной строки списка. Нумерация строк начинается с нуля. Так что в нашем случае в окне редактирования должна появиться строка **Маяк**.

Не забудьте добавить пункт IDM_DIALOG4 в главное меню, а в оконную функцию – строку:

```
case (IDM_DIALOG4)
  hDlg4=CreateDialog(hInst, LOC("ПАНЕЛЬ"C), ghwndMain,
LOC(DialogFunc4))
```

После запуска приложения получите на экране окно диалога, показанное на рис. 10.1.

Рис. 10.1. Диалог с комбинированным списком и селекторными кнопками

Функции Win32 API позволяют создать комбинированный список файлов с заданными параметрами, содержащихся в текущей директории. Для этой цели предусмотрено сообщение CB_DIR.

Прикладная программа может добавлять имена файлов или подкаталогов, посылая сообщение CB_DIR. Параметр **wParam** для процедуры окна определяет атрибуты файлов, а параметр **lParam** указывает на текстовую строку, которая определяет файл.

Внутри диалогового окна прикладная программа может также использовать функцию **DlgDirListComboBox()**. Если имя файла включает символ дисковода и путь, то функция **DlgDirListComboBox()** изменяет дисковод и каталог в строке имени файла. Новое имя каталога заносится в статический элемент диалога. Затем сбрасывается содержание списка и посылается сообщение CB_DIR.

Чтобы удалять выбранное имя файла, имя каталога или символ дисковода из заполненного комбинированного списка, используется функция **DlgDirSelectComboBoxEx()**.

Попытайтесь создать диалог с комбинированным списком такого типа. Используйте в качестве подсказки материал разд. 8.3.

УРОК

11. Общие элементы управления

В этом уроке мы приступим к рассмотрению одной из наиболее интересных новых возможностей Windows – ***общих элементов управления (common controls)***. Ранее речь шла о стандартных элементах управления, которые поддерживаются как в новых, так и в более ранних версиях Windows. Начиная с версии Windows-95 в систему были введены некоторые новые элементы управления, улучшающие вид приложений и существенно расширяющие возможности пользовательского интерфейса.

Общие элементы управления дополняют возможности стандартных элементов управления и расширяют возможности ваших приложений. Кроме того, они позволяют придать каждому приложению "собственное лицо". В Windows-95 эти элементы управления были названы *общими элементами управления* (по аналогии с общими диалогами, позволяющими выбирать шрифты, файлы и цвета), поскольку они расширяют возможности стандартных элементов управления, которые могут быть использованы несколькими приложениями одновременно.

11.1. Типы общих элементов управления

Если вы работаете с Windows достаточно давно, то, вероятно, уже видели большинство из общих элементов управления. В табл. 11.1 приведен перечень наиболее широко использующихся общих элементов управления.

Таблица 11.1. Общие элементы управления

Тип элемента	*Описание*
Список с перетаскиванием (Drag list box)	Список с возможностью визуального перетаскивания элементов
Заголовок (Header control)	Заголовок колонки таблицы

Тип элемента	*Описание*
Горячая клавиша (Hotkey control)	Поддержка пользовательских горячих клавиш
Список изображений (Image list)	Список растровых изображений
Список иконок (List view control)	Список иконок с метками
Индикатор (Progress bar)	Полоса, заполняемая внутри цветом и отражающая протекание во времени некоторого процесса
Лист свойств (Property sheet)	Диалог свойств объекта
Усложненное окно ввода (Rich edit control)	Окно ввода с возможностью редактирования
Окно состояния (Status window)	Окно, содержащее информацию о состоянии приложения
Закладка (Tab control)	Меню в виде закладок в записных книжках
Панель инструментов (Toolbar)	Меню в виде кнопок с иконками
Подсказка (Tooltip)	Небольшой всплывающий прямоугольник со справочной информацией о назначении кнопок панелей инструментов
Спин (Up-down control. Spin control)	Небольшой элемент управления с изображением стрелок "вверх" и "вниз" (up-down control). Если этот элемент интегрируется с окном ввода, он может называться еще вводом с прокруткой (spin control)
Ползунковый регулятор (Trackbar)	Элемент управления, работающий по принципу линейки прокрутки и имеющий вид ползункового регулятора в аппаратуре
Окно просмотра деревьев (Tree view control)	Специальное окно просмотра древовидных структур данных

Необходимо отметить одно важное свойство общих элементов управления: все они являются порожденными окнами. Они могут быть созданы одним из трех способов: с помощью вызова функции **CreateWindow()** (или **CreateWindowEx()**), при помощи специальной функции API либо с помощью редактора ресурсов. Поскольку общие элементы управления являются окнами, то они ведут себя и управляются так же, как все остальные окна, создаваемые программой.

Каждый элемент управления по сути является окном и требует указания стиля. Для всех *общих элементов управления* установлено несколько специальных стилей. Наиболее часто используемые из них приведены в табл. 11.2.

Таблица 11.2. Стили общих элементов управления

Стиль	*Значение*	*Описание*
CCS_TOP	#00000001	Элемент отображается наверху
CCS_NOMOVEY	#00000002	Допускает изменение только горизонтальных размеров и позиции в ответ на сообщение WM_SIZE
CCS_BOTTOM	#00000003	Элемент отображается внизу
CCS_NORESIZE	#00000004	Запрещает изменение первоначальных размеров
CCS_NOPARENTALIGN	#00000008	Запрещает автоматическое перемещение в родительском окне, сохраняет позицию внутри родительского окна
CCS_ADJUSTABLE	#00000020	Позволяет использовать стандартные средства настройки инструментальной панели

Большинство общих элементов управления при воздействии на них посылают программе сообщение WM_COMMAND или WM_NOTIFY. Многими из них можно управлять с помощью функции API **SendMessage (hWnd, MSG, wParam, lParam)**. Здесь **hWnd** представляет собой дескриптор элемента управления, MSG – сообщение, которое вы хотите направить, **wParam** и **lParam** – дополнительные параметры сообщения.

В этом уроке рассматриваются некоторые теоретические вопросы и основные приемы использования общих элементов управления в программах. В качестве примера мы дополним нашу программу окном состояния. О других общих элементах управления речь пойдет в следующих уроках.

11.2. Подключение и инициализация общих элементов управления

Для использования общих элементов управления при программировании на языке **Visial C++** в MDS предусмотрен стандартный файл определений **commctrl.h**, который содержит все необходимые константы, типы данных и описания функций. Если вы обнаружите что-то похожее на **commctrl.f90** в каталоге **Msdev\Include**, то считайте, что вам повезло. Если

же поиски не дали результата, то придется набраться терпения и создавать такой файл своими силами. Везде ниже предполагается, что нужный файл отсутствует. Вряд ли имеет смысл сразу создавать файл, который полностью заменит **commctrl.h**. Поэтому мы будем дополнять его содержание необходимыми элементами постепенно. Распределим константы, данные и интерфейсы функций по файлам так же, как это было сделано ранее для стандартных элементов приложения, – описания функций будем помещать в файл **commctrl.f90**, а константы и типы данных – в файл **commctrlty.f90**. Оба файла поместим в рабочий каталог приложения и включим явным образом их в проект. Когда вы полностью освоите общие элементы управления, перенесите эти файлы и соответствующие модули **commctrl.mod** и **commctrlty.mod** из рабочего каталога в каталог **Msdev\Include**. Это позволит в дальнейшем не заботиться о включении в проект этих файлов.

Кроме того, при компоновке программы нужно подключить библиотеку общих элементов управления, которая называется **COMCTL32.LIB** и располагается в каталоге **Msdev\Lib**. Для этого надо выполнить следующую цепочку действий: ***Build – Settings – Project Settings – Link***. Затем включить в поля ***Object/library modules*** и ***Common options*** имя библиотеки **COMCTL32.LIB**.

Прежде чем использовать любой из общих элементов управления, приложение должно вызвать функцию **InitCommonControls()**, которая загружает DLL общих элементов управления и инициализирует их подсистему. Чтобы эта функция стала доступной, включим в файл **commctrl.f90** ее интерфейс. Текст модуля **commctrl** приведен ниже.

```
!MS$IF .NOT. DEFINED (COMMCTRL_)
!MS$DEFINE COMMCTRL_

! эта строка только для полной гарантии подкл. библ.
!MS$objcomment lib:"comctl32.lib"
!*****************************************************
module commctrl

use commctrly

!============ инициализация -InitCommonControls ===========
interface
integer(4) function InitCommonControls()
!MS$ ATTRIBUTES STDCALL, ALIAS : '_InitCommonControls@0' ::
InitCommonControls
```

```
end function InitCommonControls
end interface

!=== окно состояний – STATUS  BAR  CONTROL ================

interface
integer(4) function CreateStatusWindow(Style, &
        lpszText, hwndParent, wID)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_CreateStatusWindow@16' ::
CreateStatusWindow
integer(4)   style
integer(4)   lpszText
integer(4)   hwndParent
integer(4)   wID
end function CreateStatusWindow
end interface
interface

integer(4) function DrawStatusText(hDC, lprc, pszText, uFlags)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_DrawStatusText@16' ::
DrawStatusText
integer(4)   hDC
integer(4)   lprc
integer(4)   pszText
integer(4)   uFlags
end function DrawStatusText
end interface
!************************************************
end module commctrl

!MS$ENDIF      ! COMMCTRL_
```

В модуль также добавлены интерфейсы функций **CreateStatus Window()** и **DrawStatusText()**, т. к. первую демонстрацию использования общих элементов управления мы проведем на примере окна состояния. Текст модуля **commctrlty**, который содержит необходимые константы, будет приведен ниже.

Наилучшее место для вызова функции **InitConunonControls()** – непосредственно после регистрации класса главного окна в функции **WinMain()**.

11.3. Окно состояния

Окно состояния (status window, status bar) – это горизонтальное окно внизу родительского окна, в котором прикладная программа может отображать различную информацию о состоянии приложения. Окно состояния может быть разделено на части. В нашем примере окно состояния создается с помощью специализированной функции **CreateStatusWindow()**. Однако с таким же успехом можно использовать и функцию **CreateWindowEx()**, определяя класс окна STATUSCLASSNAME.

После того как окно состояния создано, вы можете делить его на части, выводить текст и управлять видом окна, используя сообщения. По умолчанию окно состояния изображается в нижней части родительского окна. Однако вы можете объявить стиль CCS_TOP, и тогда оно появляется наверху.

Вы можете определять стиль SBARS_SIZEGRIP(#100), чтобы получить возможность оперативного управления размером окна состояния.

Функция окна состояния игнорирует значения, определенные в функции **CreateWindowEx()**, и автоматически устанавливает начальный размер и позицию окна. Ширина равна ширине рабочей области родительского окна, а высота согласуется с метрикой выбранного шрифта и шириной рамок окна. Размеры окна состояния автоматически корректируются, когда оно получает сообщение WM_SIZE. Если размер родительского окна изменяется, родитель посылает это сообщение окну состояния.

Прикладная программа может устанавливать минимальную высоту рабочей области окна состояния, посылая ему сообщение SB_SETMINHEIGHT, которое определяет минимальную высоту, в пикселах. Рабочая область не включает рамки окна. Установка минимальной высоты полезна для окна состояния, которое создается пользователем. Ширину границ окна можно определить, посылая сообщение SB_GETBORDERS. Сообщение включает указатель на массив из трех элементов, который будет содержать значения горизонтальной и вертикальной границ окна и ширину границы между частями.

Окно состояния можно разбить на несколько частей или текстовых панелей, в каждую из которых выводится собственная строка текста. Разделение на части производят, посылая сообщение SB_SETPARTS. Параметр **wParam** определяет количество частей, а параметр **lParam** – это указатель на массив, который содержит координаты правых границ частей. Для каждой части массив содержит один элемент, который указывает расстояние до правого края очередной панели от левого края рабочей области окна со-

стояния. Если при изменении размеров родительского окна необходимо сохранить деление окна состояния на части, то нужно повторно использовать сообщение SB_SETPARTS. Нумерация панелей начинается с нуля. Окно состояния может иметь максимум 255 частей. Маловероятно, что вам когда-либо понадобиться хотя бы половина этого количества.

Если послать сообщение SB_GETPARTS, то можно определить число панелей в окне состояния и получить координаты правого края каждой части. Описания сообщений окна состояния приведены в табл. 11.2.

Таблица 11.2. Сообщения окна состояния

Сообщение	*Значение*	*Описание*
SB_SETTEXT	WM_USER+1	Вывод текста в заданную панель окна; wParam содержит индекс панели в комбинации со значением, которое определяет способ отображения панели, lParam – указатель на строку текста
SB_GETTEXT	WM_USER+2	Получение текста из заданной панели, младшее слово возвращаемого значения содержит количество символов, а старшее – способ отображения текста; wParam – индекс панели, lParam – указатель на символьный буфер, куда будет помещен текст
SB_GETTEXTLENGTH	WM_USER+3	
SB_SETPARTS	WM_USER+4	Устанавливает новое количество частей (текстовых панелей); wParam – количество частей, lParam – указатель на массив координат правых границ частей
SB_GETPARTS	WM_USER+6	Получение координат правых границ текстовых панелей; wParam – число панелей, lParam – указатель на массив координат
SB_GETBORDERS	WM_USER+7	Передает ширину горизонтальной и вертикальной границы окна и разделительных линий; wParam = 0, lParam – указатель на массив из трех чисел

Сообщение	Значение	Описание
SB_SETMINHEIGHT	WM_USER+8	Устанавливает минимальную высоту рабочей области окна; wParam = 0, lParam – значение высоты
SB_SIMPLE	WM_USER+9	Изменяет изображение текста с простого на многоэлементный (wParam = .FALSE.) и обратно (wParam = TRUE.)
SB_GETRECT	WM_USER+10	Передает координаты текстовой панели; wParam – номер панели, lParam – указатель на структуру T_RECT
SB_ISSIMPLE	WM_USER + 14	Сообщает о типе окна

Сообщение SB_SETTEXT позволяет изменять способ вывода текста в окно состояний. Параметр **wParam** содержит комбинацию номера части и значения, приведенного в табл. 11.2.

Таблица 11.2. Способы отображения текста

Параметр	Значение	Описание
SBT_NOBORDERS	#100	Изображение без ограничительных линий
SBT_POPOUT	#200	Выпуклая панель
SBT_RTLREADING	#400	Написание справа налево
SBT_OWNERDRAW	#1000	Текст или другое изображение выводятся родительским окном

В модуль **commctrly** надо поместить все константы и описания типов. Сейчас он будет иметь следующий вид:

```
!MS$IF .NOT. DEFINED (COMMCTRLY_)
!MS$DEFINE COMMCTRLY_

module commctrly

use msfwina ! необходимо для использования констант и типов

!====== COMMON CONTROL STYLES ================

integer, parameter, public :: CCS_TOP          =#00000001
integer, parameter, public :: CCS_NOMOVEY      =#00000002
integer, parameter, public :: CCS_BOTTOM       =#00000003
```

```
integer, parameter, public :: CCS_NORESIZE          =#00000004
integer, parameter, public :: CCS_NOPARENTALIGN  =#00000008
integer, parameter, public :: CCS_ADJUSTABLE      =#00000020
integer, parameter, public :: CCS_NODIVIDER         =#00000040
integer, parameter, public :: CCS_VERT                =#00000080
integer, parameter, public :: CCS_LEFT    =IOR(CCS_VERT, CCS_TOP)
integer, parameter, public :: CCS_RIGHT  =IOR(CCS_VERT, CCS_BOTTOM)
integer, parameter, public :: CCS_NOMOVEX =IOR(CCS_VERT,
CCS_NOMOVEY)

!====== STATUS BAR CONTROL ==========================
!MS$IF .NOT. DEFINED  (NOSTATUSBAR)

!---------- имя класса ----------------------------------------------------
 character(19), parameter, public :: STATUSCLASSNAME    =
"msctls_statusbar32"C

!----------  константы ----------------------------------------------------
integer, parameter, public:: SBARS_SIZEGRIP             = #100

integer, parameter, public :: SB_GETTEXT                = (WM_USER+2)
integer, parameter, public :: SB_SETTEXT                = (WM_USER+1)
integer, parameter, public :: SB_GETTEXTLENGTH    = (WM_USER+3)
integer, parameter, public :: SB_SETPARTS             = (WM_USER+4)
integer, parameter, public :: SB_GETPARTS             = (WM_USER+6)
integer, parameter, public :: SB_GETBORDERS       = (WM_USER+7)
integer, parameter, public :: SB_SETMINHEIGHT     = (WM_USER+8)
integer, parameter, public :: SB_SIMPLE               = (WM_USER+9)
integer, parameter, public :: SB_GETRECT              = (WM_USER+10)
integer, parameter, public :: SB_ISSIMPLE             = (WM_USER+14)

integer, parameter, public :: SBT_OWNERDRAW       = #1000
integer, parameter, public :: SBT_NOBORDERS       = #100
integer, parameter, public :: SBT_POPOUT          = #200
integer, parameter, public :: SBT_RTLREADING      = #400

!MS$ENDIF  ! END STATUS BAR CONTROL
!-------------------------------------------------------------------------
end module commctrly

!MS$ENDIF  ! COMMCTRLY_
```

В дальнейшем мы будем постепенно дополнять его все новыми и новыми фрагментами.

11.4. Инициализация окна состояния и взаимодействие с ним

Для инициализации окна состояния удобно использовать специализированную функцию **CreateStatusWindow(Style, lpszText, hwndParent, wID)**. Она создает окно состояния, которое по умолчанию появляется внизу родительского окна и содержит некоторый текст. Первый параметр, **Style** задает стиль окна и должен обязательно включать стиль WS_CHILD и, если это необходимо, стиль WS_VISIBLE. Следующий параметр, **lpszText** указывает на С-строку, которая будет выведена в первой текстовой панели. Далее следуют: параметр **hwndParent**, который задает дескриптор родительского окна, и параметр **wID** – идентификатор окна состояния. При успешном выполнении функция возвращает дескриптор окна состояния. Функция **CreateStatusWindow()** – это функция обращения к функции **Create Window()**. Она передает все необходимые параметры, устанавливает позицию, ширину и высоту окна.

Функцию **InitStatusWnd()**, которая инициализирует окно состояния нашего приложения, удобно включить в файл **MyPr_1inc.f90**. Текст функции приведен ниже.

```
!*****************************************************************************/
integer(4) function InitStatusWnd()

integer(4)     :: dxStWnd
character(50)  :: Text

  ir = GetClientRect(ghwndMain, rcWnd)
  dxStWnd = rcWnd%right-rcWnd%left
  Part(0)=INT(dxStWnd/2.)
  Part(1)=Part(0)+ INT(dxStWnd/5.)
  Part(2)=Part(0)+ INT(2.*dxStWnd/5.)
  Part(3)=Part(0)+ Part(0)
  Text="Это окно состояния"C

  hStWnd = CreateStatusWindow(WS_CHILD, LOC(Text), &
                              ghwndMain, iDStWnd)
  ir=SendMessage(hStWnd, SB_SETPARTS, 4, LOC(Part))
  ir = ShowWindow(hStWnd, SW_SHOW)
  InitStatusWnd=hStWnd
end function InitStatusWnd
!*****************************************************************************/
```

Прежде чем создать окно, формируем массив значений координат каждой его панели. В примере окно разбито на четыре части. Координаты частей вычисляются, исходя из размеров родительского окна, которые определяются с помощью функции **GetClientRect(ghwndMain, rcWnd)**. Первый параметр – это дескриптор родительского окна, а второй – переменная типа T_RECT. После создания окна состояния ему посылается сообщение **SendMessage(hStWnd, SB_SETPARTS, 4, LOC(Part))**. Строка **"Text ="Это окно состояния"C"** задает текст на первой текстовой панели окна состояния.

Для удобства взаимодействия с окном переменные **hStWnd** и **Part** объявлены в модуле **MyPr_1inc** следующим образом:

```
integer, dimension(0:3)      :: Part
integer(4)                   :: hStWnd.
```

После запуска приложения на экране появляется окно состояния, показанное на рис. 11.1.

Рис. 11.1. Фрагмент окна приложения с окном состояния

Приложение может взаимодействовать с окном состояния, посылая ему сообщения. Для этого необходимо сделать соответствующие дополнения в тексте оконной функции **MainWndProc()**. В качестве примера введем следующий фрагмент:

```
case (WM_SIZE)
  ir= SendMessage(hStWnd, WM_SIZE, SIZE_RESTORED, 0).
```

Теперь при изменении размеров родительского окна окно состояния также будет изменять размеры. Однако его внешний вид при этом будет меняться, т. к. координаты текстовых панелей остаются неизменными. Попытайтесь самостоятельно добиться неизменности внешнего вида окна, используя сообщение SB_SETPARTS так же, как это было сделано в функции **InitStatusWnd()**.

Вид окна состояния можно оперативно поменять используя сообщения, приведенные в табл. 11.2. Изменим следующим образом текст оконной функции **MainWndProc()**:

```
case (WM_SIZE)
  ir= SendMessage(hStWnd, WM_SIZE, SIZE_RESTORED, 0)
  i = IOR(1, SBT_POPOUT)
  ir= SendMessage(hStWnd, SB_SETTEXT, i, LOC(szText))
  ir= SendMessage(hStWnd, SB_GETTEXT, 1, LOC(szText)).
```

Теперь, после изменения размеров главного окна, текст второй части окна состояния будет выглядеть так, как это показано на рис. 11.2.

Рис. 11.2. Окно состояния с выпуклым текстом

УРОК

12. Панель инструментов

Панель инструментов (toolbar), или ***инструментальная панель***, – это окно управления, которое содержит одну или несколько кнопок. Каждая кнопка посылает сообщение родительскому окну, когда пользователь ее выбирает. Обычно кнопки в инструментальной панели соответствуют пунктам меню прикладной программы и обеспечивают дополнительный, более быстрый путь для обращения к командам прикладной программы. Инструментальная панель располагается, как правило, ниже панели меню.

12.1. Создание панели инструментов

Инструментальную панель можно создать, используя функцию **CreateWindowEx()** или специализированную функцию **CreateToolbar Ex()** и объявляя класс окна TOOLBARCLASSNAME. Класс TOOLBARCLASSNAME будет зарегистрирован, если загружена библиотека общих элементов управления. Чтобы это гарантировать, надо использовать функцию **InitCommonControls()**. Мы уже говорили с вами об этой функции в уроке 11.

Удобнее для создания панели инструментов использовать специализированную функцию **CreateToolbarEx(hwnd, ws, wID, nBitmaps, hBMInst, wBMID, lpButtons, iNumButtons, dxButton, dyButton, dxBitmap, dyBitmap, uStructSize)**. Здесь **hwnd** – дескриптор родительского окна, ws – стиль окна панели инструментов, **wID** – идентификатор панели инструментов. Число изображений кнопок задается параметром **nBitmaps**, а дескриптор приложения – **hBMInst**. Параметр **wBMID** – это идентификатор ресурса, который содержит изображения кнопок. Параметр **lpButtons** – это указатель на массив структур типа T_TBBUTTON, который содержит информацию о кнопках. Число кнопок определяется значением **iNumButtons** (не путать с **nBitmaps**). Размеры кнопок и изображений устанавливаются значениями параметров **dxButton**, **dyButton**, **dxBitmap**, **dyBitmap**. Наконец, параметр **uStructSize** – это размер структуры T_TBBUTTON. При успешном выполнении функция возвращает дескриптор окна.

Инструментальная панель, как всякое окно, может иметь различные стили, но объявление стиля WS_CHILD обязательно. Если вы используете **CreateWindowEx()**, то этот стиль указывается явно. Функция **CreateToolbarEx()** включает стиль WS_CHILD по умолчанию. При создании инструментальной панели автоматически устанавливается ее размер и позиция окна инструментальной панели. Значение высоты согласуется с высотой кнопок, а ширина – с шириной рабочей области пользователя родительского окна. Стили общих элементов управления CCS_TOP и CCS_BOTTOM определяют, установлена ли инструментальная панель в верхней или нижней части рабочей области родительского окна. По умолчанию инструментальная панель имеет стиль CCS_TOP.

Кроме того, для инструментальной панели предусмотрено несколько специфических стилей. Значения этих стилей приведены в табл. 12.1.

Таблица 12.1. Стили инструментальной панели

Стиль	*Значение*	*Описание*
TBSTYLE_BUTTON	#0000	Обычная кнопка
TBSTYLE_SEP	#0001	Разделитель между кнопками
TBSTYLE_CHECK	#0002	При нажатии состояние изменяется на противоположное
TBSTYLE_GROUP	#0004	Обычная кнопка в группе кнопок
TBSTYLE_DROPDOWN	#0008	Всплывающая панель
TBSTYLE_TOOLTIPS	#0100	Панель инструментов снабжена подсказками
TBSTYLE_WRAPABLE	#0200	Кнопки панели располагаются в несколько рядов
TBSTYLE_ALTDRAG	#0400	Пользователь может перемещать кнопку
TBSTYLE_TRANSPARENT	#0800	Прозрачная кнопка
TBSTYLE_FLAT	#1000	Плоская кнопка
TBSTYLE_HOTTRACK	#2000	Используется при изменении позиции
TBSTYLE_NOTEXT	#4000	

Стиль TBSTYLE_ALTDRAG дает возможность пользователю изменять позицию кнопки инструментальной панели, перемещая ее мышью при нажатой клавише ALT. Если этот стиль не определен, то нужно использовать клавишу SHIFT при перемещении кнопки. Обратите внимание, что перемещение кнопок возможно, только если объявлен стиль CCS_ADJUSTABLE.

Устанавливая стиль TBSTYLE_TOOLTIPS, вы снабжаете инструментальную панель подсказками. Подсказка – это маленькое всплывающее окно, которое содержит текст, описывающий кнопку инструментальной панели. Этот элемент управления будет рассмотрен в уроке 14.

Стиль TBSTYLE_WRAPABLE создает инструментальную панель, которая может иметь несколько рядов кнопок. Кнопки могут переноситься на следующую линию, когда их не удается разместить в один ряд.

Каждая кнопка инструментальной панели должна иметь связанную с ней структуру типа T_TBBUTTON, которая задает характеристики кнопки. Ниже приведен фрагмент модуля **commctrlty**, который, в частности, содержит определение этой структуры.

У структуры типа T_TBBUTTON шесть полей. Поле **iBitmap** определяет индекс изображения, связанного с кнопкой. Поле **idCommand** определяет команду, ассоциированную с кнопкой.

Начальное состояние кнопки задается в поле **fsState**. Кнопка инструментальной панели может иметь комбинацию состояний, приведенных в табл. 12.2.

Таблица 12.2. Состояния кнопок инструментальной панели

Состояние	*Значение*	*Описание*
TBSTATE_CHECKED	#01	Кнопка нажата
TBSTATE_PRESSED	#02	Кнопка нажата
TBSTATE_ENABLED	#04	Кнопка доступна, иначе она не взаимодействует с пользователем и выглядит "серой"
TBSTATE_HIDDEN	#08	Кнопка невидима
TBSTATE_INDETERMINATE	#10	Кнопка неактивна и выглядит "серой"
TBSTATE_WRAP	#20	Все следующие кнопки будут изображаться в новой строке

Стиль кнопки задается в поле **fsStyle**.

Поле **dwData** может содержать дополнительную информацию, определяемую программистом. Последнее поле **iString** содержит индекс строки, ассоциированной с кнопкой. Если это поле не используется, то его значение должно быть равно нулю.

12.2. Создание шаблона инструментальной панели с помощью редактора ресурсов

Прежде чем использовать панель инструментов в программе, нужно создать описание панели. Удобнее всего использовать для этого средства MDS.

Добавим новый ресурс в проект, используя средства MDS. Для этого выполним следующую цепочку действий: ***Insert – Resource – Toolbar***. На экране появится окно графического редактора. Однако прежде нужно вернуться в окно проекта (***Project Workspace***) и, щелкнув правой кнопкой мыши на имени вновь созданного диалога, открыть окно ***Toolbar Properties***. Заменим английский язык на русский. При необходимости можно заменить идентификатор панели инструментов и имя файла с рисунками для кнопок. Вернемся в графический редактор и приступим к формированию инструментальной панели.

Редактор позволяет создавать новые инструментальные панели и кнопки, а также преобразовывать растровые рисунки в ресурсы инструментальной панели. Функциональные возможности редактора мало чем отличаются от возможностей других графических редакторов.

Прежде всего создадим кнопки инструментальной панели. В верхней части поля редактора имеется панель с изображением кнопки (или кнопок). Новая или "пустая" кнопка отображается, по умолчанию, в правом конце панели. Эту кнопку можно перемещать с помощью мыши. Как только вы сочтете, что еще одна кнопка создана, и освободите клавишу мыши, справа появится другая "пустая" кнопка. Перемещение кнопок можно производить многократно. Если перемещать кнопку при нажатой клавише CTRL, то можно осуществить ее копирование. Как правило, кнопки разделяются визуально на панели на отдельные группы. Для образования зазора между соседними кнопками, надо переместить кнопку так, чтобы она не накрывала соседнюю. Пробелы между кнопками уничтожаются при незначительном наложении изображений друг на друга. Кнопка удаляется, если переместить ее за пределы панели. Действуя таким образом, создаем структуру панели инструментов.

Теперь, переходя от кнопки к кнопке, двойным щелчком открываем окно ***Toolbar Button Properties***. В поле **ID** вводим идентификатор кнопки, который совпадает с соответствующим пунктом меню. Устанавливаем размеры кнопок. Поле **Prompt** может содержать текст, который появиться в окне состояния. Эту возможность можно реализовать только в файлах

ресурса, поддерживающих библиотеку фундаментальных классов Microsoft (MFC).

В завершение надо перейти к созданию рисунков для кнопок. Это очень кропотливая работа, которая к тому же требует определенных художественных навыков. Изображения проще создавать в каком-либо графическом редакторе заранее. Имя файла изображения затем можно занести в описание ресурса. Часто встречающиеся изображения можно создать, используя копирование с экрана.

В нашем примере используются стандартные изображения кнопок, которые хранятся в Системе. Такие изображения задаются специальными идентификаторами, которые имеют префикс STD_ и не требуют выполнения каких-либо графических работ. В этом случае изображения кнопок в ресурсе панели инструментов остаются без рисунков. Идентификаторы стандартных изображений можно обнаружить в приведенном в разд. 12.4 фрагменте модуля **commctrlty**. Смысл каждого из них легко понять.

12.3. Взаимодействие с панелью инструментов

По умолчанию панель инструментов является полностью автоматическим элементом управления и не требует от программиста каких-либо действий по управлению ею. Однако при необходимости можно использовать специальные управляющие сообщения, которые имеют префикс TB_. Все эти сообщения включены в модуль **commctrlty**. Их общее число настолько велико, что нет никакой возможности дать описание каждого из них. В табл. 12.3 приведены сообщения, которые можно посылать панели инструментов.

Таблица 12.3. Сообщения панели инструментов

Сообщение	*Значение*	*Описание*
TB_ENABLEBUTTON	WM_USER + 1	Запрещает или разрешает доступ к кнопке
TB_CHECKBUTTON	WM_USER + 2	Пометить кнопку
TB_PRESSBUTTON	WM_USER + 3	Нажать кнопку
TB_HIDEBUTTON	WM_USER + 4	Спрятать или показать кнопку
TB_INDETERMINATE	WM_USER + 5	Кнопка в неопределенном состоянии?
TB_ISBUTTONENABLED	WM_USER + 9	Кнопка доступна?

Сообщение	*Значение*	*Описание*
TB_ISBUTTONCHECKED	WM_USER + 10	Кнопка помечена?
TB_ISBUTTONPRESSED	WM_USER + 11	Кнопка нажата?
TB_ISBUTTONHIDDEN	WM_USER + 12	Кнопка спрятана?
TB_ISBUTTONINDETERMINATE	WM_USER + 13	Кнопка
TB_SETSTATE	WM_USER + 17	Устанавливает состояние кнопки
TB_GETSTATE	WM_USER + 18	Передает состояние кнопки
TB_ADDBITMAP	WM_USER + 19	Добавляет рисунок в список рисунков панели
TB_ADDBUTTONS	WM_USER + 20	Добавляет кнопку в панель
TB_INSERTBUTTON	WM_USER + 21	Вставить кнопку
TB_DELETEBUTTON	WM_USER + 22	Убрать кнопку из панели
TB_GETBUTTON	WM_USER + 23	Передает информацию о кнопке
TB_BUTTONCOUNT	WM_USER + 24	Передает общее число кнопок на панели
TB_COMMANDTOINDEX	WM_USER + 25	Передает номер кнопки, связанной с идентификатором
TB_SAVERESTORE	WM_USER + 26	Сохранить или изменить информацию о кнопке
TB_CUSTOMIZE	WM_USER + 27	Отображает кнопку, созданную программистом
TB_ADDSTRING	WM_USER + 28	Добавляет строку в список строк панели
TB_GETITEMRECT	WM_USER + 29	Передает габариты кнопки
TB_BUTTONSTRUCTSIZE	WM_USER + 30	Устанавливает размер структуры T_TBBUTTON

Сообщение	Значение	Описание
TB_SETBUTTONSIZE	WM_USER + 31	Устанавливает размеры кнопки
TB_SETBITMAPSIZE	WM_USER + 32	Устанавливает размеры рисунка
TB_AUTOSIZE	WM_USER + 33	Кнопка изменяет размеры автоматически
TB_GETTOOLTIPS	WM_USER + 35	Передает дескриптор подсказок
TB_SETTOOLTIPS	WM_USER + 36	Связывает подсказку с кнопкой
TB_SETPARENT	WM_USER + 37	Назначает родительское окно для панели
TB_SETROWS	WM_USER + 39	Устанавливает число линий кнопок
TB_GETROWS	WM_USER + 40	Передает число линий кнопок
TB_SETCMDID	WM_USER + 42	Связывает идентификатор команды с кнопкой
TB_CHANGEBITMAP	WM_USER + 43	Заменяет рисунок для данной кнопки
TB_GETBITMAP	WM_USER + 44	Передает номер рисунка, связанного с кнопкой
TB_GETBUTTONTEXT	WM_USER + 45	Передает текст, связанный с кнопкой
TB_REPLACEBITMAP	WM_USER + 46	Заменяет изображение
TB_SETINDENT	WM_USER + 47	
TB_SETIMAGELIST	WM_USER + 48	
TB_GETIMAGELIST	WM_USER + 49	
TB_LOADIMAGES	WM_USER + 50	
TB_GETRECT	WM_USER + 51	
TB_SETHOTIMAGELIST	WM_USER + 52	
TB_GETHOTIMAGELIST	WM_USER + 53	
TB_SETDISABLEDIMAGELIST	WM_USER + 54	
TB_GETDISABLEDIMAGELIST	WM_USER + 55	

Сообщение	*Значение*	*Описание*
TB_SETSTYLE	WM_USER + 56	Устанавливает стиль
TB_GETSTYLE	WM_USER + 57	Передает стиль
TB_GETBUTTONSIZE	WM_USER + 58	Передает размеры кнопки

Родительское окно устанавливается при создании инструментальной панели, но вы можете его изменить, используя сообщение TB_ SETPARENT. Параметр **wParam** содержит дескриптор родительского окна.

Размер инструментальной панели автоматически корректируется всякий раз, когда она получает сообщения WM_SIZE или TB_AUTOSIZE. Прикладная программа должна посылать любое из этих сообщений всякий раз, когда изменяется размер родительского окна или после посылки сообщения, которое требует последующего изменения размера инструментальной панели, например TB_SETBUTTONSIZE. Параметр **lParam** содержит размеры кнопки в виде **MAKELONG(dxButton, dyButton)**.

Каждая кнопка в инструментальной панели может включать растровое изображение. Инструментальная панель сохраняет информацию об изображениях, которые необходимо выводить на экран, во внутреннем списке. Вызывая функцию **CreateToolbarEx()**, вы указываете имя файла, который содержит начальные изображения. В результате формируется первичный список изображений. Однако вы можете добавлять дополнительные изображения, используя сообщение TB_ADDBITMAP. Параметр wParam указывает общее число кнопок в изображении, а параметр **lParam** – это адрес структуры типа T_TBADDBITMAP, которая определена следующим образом:

```
type T_TBADDBITMAP
 integer(4) :: hInst
 integer(4) :: nID
end type T_TBADDBITMAP
```

Первое поле этой структуры содержит дескриптор приложения, которое содержит ресурс рисунка. Если в это поле ввести значение HINST_COMMCTRL, то можно использовать системные изображения стандартных кнопок. В поле **nID** заносится идентификатор ресурса, содержащего изображение кнопки. Если **hInst** равно нулю, то **nID** – это дескриптор изображения.

Каждое изображение имеет индекс или номер. Нумерация изображений, добавляемых во внутреннем списке, начинается с нуля. Сообщение TB_ADDBITMAP добавляет одно или несколько изображений в конец

списка и возвращает индекс первого них. Индексы изображения используются для связи изображения с кнопкой и указываются в структуре типа T_TBBUTTON.

Если вы используете функцию **CreateWindowEx()** для создания инструментальной панели, то перед добавлением любых кнопок вы обязательно должны послать сообщение TB_BUTTONSTRUCTSIZE, которое передает размер структуры T_TBBUTTON через параметр **wParam**. Функция **CreateWindowEx()** создает панель, которая первоначально не содержит никаких кнопок. Вы добавляете кнопки, используя сообщения TB_ADDBUTTONS или TB_INSERTBUTTON. Адрес структуры T_TBBUTTON содержится в параметре **lParam**. Параметр **wParam** в первом случае равен числу добавляемых кнопок, а во втором – номеру кнопки, перед которой будет вставлена новая.

Прикладная программа может использовать сообщения TB_GETSTATE и TB_SETSTATE для определения или установления состояния кнопки. И в том и в другом случае параметр **wParam** – это идентификатор кнопки. При установке состояния параметр **lParam** содержит одно из значений, приведенных в табл. 12.2.

Кроме того, существует несколько сообщений, которые относятся к конкретным состояниям кнопки.

Каждая кнопка имеет идентификатор команды, связанный с ней. Когда пользователь выбирает какую-либо кнопку, инструментальная панель посылает родительскому окну сообщение WM_COMMAND, которое содержит идентификатор команды этой кнопки. Оконная функция анализирует идентификатор и выполняет нужную команду.

Кроме обычных сообщений инструментальная панель посылает родительскому окну в форме нотификационных сообщений WM_NOTIFY. Основная часть таких сообщений обслуживает процедуру настройки панели инструментов, которая имеет стиль CCS_ADJUSTABLE. Для нас представляет интерес сообщение TTN_NEEDTEXT, которое посылает инструментальная панель, снабженная подсказками (стиль TBSTYLE_TOOLTIPS). Более подробно мы рассмотрим этот вопрос в следующем уроке.

12.4. Включение инструментальной панели в приложение

Теперь мы можем добавить панель инструментов в наше приложение. Для этого прежде всего расширим возможности модулей **commctrl** и **commctrlty**.

В примере, который приведен ниже, инструментальная панель создается функцией **CreateWindowEx()**. Однако функция **CreateToolbarEx()** наверняка пригодится вам в будущем. Поэтому представленный ниже текст следует добавить в модуль **commctrl**.

```
!====== TOOLBAR CONTROL ===================================

interface
integer(4) function CreateToolbarEx(hwnd, ws, wID, nBitmaps, hBMInst, &
        wBMID, lpButtons, iNumButtons, dxButton, dyButton, dxBitmap,
dyBitmap, uStructSize)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_CreateToolbarEx@52' ::
CreateToolbarEx
use commctrly

integer(4)      hwnd
integer*4       ws
integer*4       wID
integer*4       nBitmaps
integer*4       hBMInst
integer*4       wBMID
integer(4)      lpButtons
integer*4       iNumButtons
integer*4       dxButton
integer*4       dyButton
integer*4       dxBitmap
integer*4       dyBitmap
integer*4       uStructSize
end function
end interface
```

В модуль **commctrlty** следует добавить константы и описания типов, которые имеют отношение к функционированию и описанию панели инструментов. Фрагмент обновленного текста модуля приведен ниже.

```
!====== TOOLBAR CONTROL ===================================
!MS$IF .NOT. DEFINED (NOTOOLBAR)

!---------- имя класса ----------------------------------
character(16), parameter, public :: TOOLBARCLASSNAME    =
"ToolbarWindow32"C

!----------    константы ----------------------------------
integer, parameter, public :: CMB_MASKED                = #02

integer, parameter, public :: RT_TOOLBAR                = 241
```

```
integer, parameter, public :: TBSTATE_CHECKED          = #01
integer, parameter, public :: TBSTATE_PRESSED          = #02
integer, parameter, public :: TBSTATE_ENABLED          = #04
integer, parameter, public :: TBSTATE_HIDDEN           = #08
integer, parameter, public :: TBSTATE_INDETERMINATE    = #10
integer, parameter, public :: TBSTATE_WRAP             = #20

integer, parameter, public :: TBSTYLE_BUTTON           = #0
integer, parameter, public :: TBSTYLE_SEP              = #01
integer, parameter, public :: TBSTYLE_CHECK            = #02
integer, parameter, public :: TBSTYLE_GROUP            = #04
integer, parameter, public :: TBSTYLE_DROPDOWN         = #08

integer, parameter, public :: TBSTYLE_TOOLTIPS         = #0100
integer, parameter, public :: TBSTYLE_WRAPABLE         = #0200
integer, parameter, public :: TBSTYLE_ALTDRAG          = #0400
integer, parameter, public :: TBSTYLE_TRANSPARENT      = #0800
integer, parameter, public :: TBSTYLE_FLAT             = #1000
integer, parameter, public :: TBSTYLE_HOTTRACK         = #2000
integer, parameter, public :: TBSTYLE_NOTEXT           = #4000

integer, parameter, public :: TB_ENABLEBUTTON          = (WM_USER + 1)
integer, parameter, public :: TB_CHECKBUTTON           = (WM_USER + 2)
integer, parameter, public :: TB_PRESSBUTTON           = (WM_USER + 3)
integer, parameter, public :: TB_HIDEBUTTON            = (WM_USER + 4)
integer, parameter, public :: TB_INDETERMINATE         = (WM_USER + 5)
integer, parameter, public :: TB_ISBUTTONENABLED       = (WM_USER + 9)
integer, parameter, public :: TB_ISBUTTONCHECKED       = (WM_USER + 10)
integer, parameter, public :: TB_ISBUTTONPRESSED       = (WM_USER + 11)
integer, parameter, public :: TB_ISBUTTONHIDDEN        = (WM_USER + 12)
integer, parameter, public :: TB_ISBUTTONINDETERMINATE = (WM_USER + 13)
integer, parameter, public :: TB_SETSTATE              = (WM_USER + 17)
integer, parameter, public :: TB_GETSTATE              = (WM_USER + 18)
integer, parameter, public :: TB_ADDBITMAP             = (WM_USER + 19)

integer, parameter, public :: TB_ADDBUTTONS            = (WM_USER + 20)
integer, parameter, public :: TB_INSERTBUTTON          = (WM_USER + 21)
integer, parameter, public :: TB_DELETEBUTTON          = (WM_USER + 22)
integer, parameter, public :: TB_GETBUTTON             = (WM_USER + 23)
integer, parameter, public :: TB_BUTTONCOUNT           = (WM_USER + 24)
integer, parameter, public :: TB_COMMANDTOINDEX        = (WM_USER + 25)

integer, parameter, public :: TB_SAVERESTORE           = (WM_USER + 26)
integer, parameter, public :: TB_CUSTOMIZE             = (WM_USER + 27)
integer, parameter, public :: TB_ADDSTRING             = (WM_USER + 28)
```

```
integer, parameter, public :: TB_GETITEMRECT          = (WM_USER + 29)
integer, parameter, public :: TB_BUTTONSTRUCTSIZE = (WM_USER + 30)
integer, parameter, public :: TB_SETBUTTONSIZE        = (WM_USER + 31)
integer, parameter, public :: TB_SETBITMAPSIZE        = (WM_USER + 32)
integer, parameter, public :: TB_AUTOSIZE             = (WM_USER + 33)
integer, parameter, public :: TB_GETTOOLTIPS          = (WM_USER + 35)
integer, parameter, public :: TB_SETTOOLTIPS          = (WM_USER + 36)
integer, parameter, public :: TB_SETPARENT            = (WM_USER + 37)
integer, parameter, public :: TB_SETROWS              = (WM_USER + 39)
integer, parameter, public :: TB_GETROWS              = (WM_USER + 40)
integer, parameter, public :: TB_SETCMDID             = (WM_USER + 42)
integer, parameter, public :: TB_CHANGEBITMAP         = (WM_USER + 43)
integer, parameter, public :: TB_GETBITMAP            = (WM_USER + 44)
integer, parameter, public :: TB_GETBUTTONTEXT        = (WM_USER + 45)
integer, parameter, public :: TB_REPLACEBITMAP        = (WM_USER + 46)
integer, parameter, public :: TB_SETINDENT            = (WM_USER + 47)
integer, parameter, public :: TB_SETIMAGELIST         = (WM_USER + 48)
integer, parameter, public :: TB_GETIMAGELIST         = (WM_USER + 49)
integer, parameter, public :: TB_LOADIMAGES           = (WM_USER + 50)
integer, parameter, public :: TB_GETRECT              = (WM_USER + 51)
integer, parameter, public :: TB_SETHOTIMAGELIST      = (WM_USER + 52)
integer, parameter, public :: TB_GETHOTIMAGELIST      = (WM_USER + 53)
integer, parameter, public :: TB_SETDISABLEDIMAGELIST = (WM_USER + 54)
integer, parameter, public :: TB_GETDISABLEDIMAGELIST = (WM_USER + 55)
integer, parameter, public :: TB_SETSTYLE             = (WM_USER + 56)
integer, parameter, public :: TB_GETSTYLE             = (WM_USER + 57)
integer, parameter, public :: TB_GETBUTTONSIZE        = (WM_USER + 58)

!--------- стандартные изображения ----------------------

integer, parameter, public :: HINST_COMMCTRL          = -1
integer, parameter, public :: IDB_STD_SMALL_COLOR     = 0
integer, parameter, public :: IDB_STD_LARGE_COLOR     = 1
integer, parameter, public :: IDB_VIEW_SMALL_COLOR    = 4
integer, parameter, public :: IDB_VIEW_LARGE_COLOR    = 5
integer, parameter, public :: IDB_HIST_SMALL_COLOR    = 8
integer, parameter, public :: IDB_HIST_LARGE_COLOR    = 9

integer, parameter, public :: STD_CUT                 = 0
integer, parameter, public :: STD_COPY                = 1
integer, parameter, public :: STD_PASTE               = 2
integer, parameter, public :: STD_UNDO                = 3
integer, parameter, public :: STD_REDOW               = 4
integer, parameter, public :: STD_DELETE              = 5
```

```
integer, parameter, public :: STD_FILENEW      = 6
integer, parameter, public :: STD_FILEOPEN     = 7
integer, parameter, public :: STD_FILESAVE     = 8
integer, parameter, public :: STD_PRINTPRE     = 9
integer, parameter, public :: STD_PROPERTIES   = 10
integer, parameter, public :: STD_HELP         = 11
integer, parameter, public :: STD_FIND         = 12
integer, parameter, public :: STD_REPLACE      = 13
integer, parameter, public :: STD_PRINT        = 14

!---------- структуры -----------------------------------------------
type T_TBBUTTON
 integer(4) :: iBitmap
 integer(4) :: idCommand
 byte       :: fsState
 byte       :: fsStyle
 integer(4) :: dwData
 integer(4) :: iString
end type T_TBBUTTON

type T_TBADDBITMAP
 integer(4) ::  hInst
 integer(4) ::  nID
end type T_TBADDBITMAP

type T_TBNOTIFY
 type(T_NMHDR)   :: hdr
 integer(4)      :: iItem
 type(T_TBBUTTON) :: tbButton
 integer(4)      :: cchText
 integer(4)      :: pszText
end type T_TBNOTIFY

type T_TBSAVEPARAMS
 integer(4)      :: hkr
 integer(4)      :: pszSubKey
 integer(4)      :: pszValueName
end type T_TBSAVEPARAMS

!MS$ENDIF  ! NOTOOLBAR
```

Функцию **InitToolBar()**, которая будет создавать панель инструментов, включим в модуль **MyPr_1inc** в раздел **contains**.

Прежде всего необходимо загрузить ресурс панели инструментов, шаблон которого мы создали с помощью графического редактора. Для

этого используем функции **FindResource (hInst, IDR_TOOLBAR1, RT_TOOLBAR)**, **LoadResource (hInst, hResInfo)** и **LockResource (hResData)**. Первая их них возвращает дескриптор ресурса заданного типа **hResInfo**, значение которого используется второй функцией для загрузки ресурса в глобальную память. Последняя функция фиксирует в памяти ресурс и возвращает указатель на первый байт блока памяти (т. е. адрес). Зафиксированный объект не перемещается в памяти и не выгружается. Теперь, обращаясь к ячейкам памяти, можно получить данные, полностью описывающие панель инструментов. Данные представлены двухбайтовыми числами в следующем порядке:

integer(2) wVersion – номер версии редактора,
integer(2) wWidth – ширина кнопки,
integer(2) wHeight – высота кнопки,
integer(2) wItemCount – число кнопок,
integer(2) items – идентификатор команды для каждой из кнопок.

Для того чтобы в Фортране-90 обратиться к ячейкам памяти по указателю (адресу), надо использовать целочисленный указатель (не путать с указателем). Целочисленный указатель надо связать с переменной – **(POINTER (ip, dt))**, которая будет использоваться затем в программе. В данном случае это будет переменная **integer(2) dt**. Затем надо преобразовать адрес в целочисленный указатель Фортрана-90 (**ip = TRANSFER(ir, ip)**). Теперь значение переменной равно числу, записанному по заданному адресу. Этот прием можно использовать и для более сложных типов переменных, например для структур.

Теперь мы имеем возможность разметить панель инструментов, заполняя структуры типа T_TBBUTTON.

Для того чтобы использовать для кнопок стандартные системные рисунки, необходимо в структуре типа T_TBADDBITMAP в поле **hInst** указать значение HINST_COMMCTRL, а в поле **nID** – IDB_STD_SMALL_COLOR.

Изображения связываются с кнопками при заполнении поля **iBitmap** структуру типа T_TBBUTTON.

Поскольку предполагается использовать стандартные изображения для кнопок, то панель будем создавать с помощью функции **Create WindowEx()**. Панель не отображается на экране сразу, т. к. пока еще ни кнопки, ни изображения не загружены. Поэтому стиль WM_VISIBALE не используется.

Нужное количество изображений и кнопок добавляем, используя сообщения TB_ADDBITMAP и TB_ADDBUTTONS.

Окончательный текст функции **InitToolBar()**, которая инициализирует панель инструментов в нашем приложении, выглядит следующим образом:

```
!*****************************************************************************
integer(4) function InitToolBar()
!_____________________________________________________________________________
 type(T_TBADDBITMAP) :: tbab
 type (T_TBBUTTON)   :: tbBs

 integer iButtons, iBitmaps, iW, iH

 integer(2) dt
 POINTER(ip, dt)

!---- загружаем ресурс инструментов ----------------
ir= FindResource(hInst, IDR_TOOLBAR1, RT_TOOLBAR)
ir =LoadResource(hInst, ir)
ir =LockResource(ir)

ir=ir+2
ip = TRANSFER(ir, ip)
iW=dt

ir=ir+2
ip = TRANSFER(ir, ip)
iH=dt

ir=ir+2
ip = TRANSFER(ir, ip)
iButtons =dt

!---- размечаем панель инструментов ----------------
do i=1, iButtons
 ip = TRANSFER(ir+2*i, ip)
 tbButtons(i)%idCommand = dt
 tbButtons(i)%fsStyle   = TBSTYLE_BUTTON
 if(dt==0) &
 tbButtons(i)%fsStyle   = TBSTYLE_SEP
 tbButtons(i)%fsState   = TBSTATE_ENABLED
end do
!------ используем стандартные иконки ----------------
 tbab%hInst = HINST_COMMCTRL
 tbab%nID   = IDB_STD_SMALL_COLOR
```

```
!----- формируем изображения кнопок ------------------
    tbButtons(1)%iBitmap    = STD_FILENEW
    tbButtons(2)%iBitmap    = STD_FILEOPEN
    tbButtons(3)%iBitmap    = STD_FILESAVE
    tbButtons(4)%iBitmap    = 0
    tbButtons(5)%iBitmap    = STD_CUT
    tbButtons(6)%iBitmap    = STD_COPY
    tbButtons(7)%iBitmap    = STD_PASTE
    tbButtons(8)%iBitmap    = STD_PRINT
    tbButtons(9)%iBitmap    = 0,
    tbButtons(10)%iBitmap   = STD_HELP

hTBWnd = CreateWindowEx(0, TOOLBARCLASSNAME, ""C,          &
           IOR(WS_CHILD, TBSTYLE_TOOLTIPS),                &
           0, 0, 0, 0, ghwndMain, IDR_TOOLBAR1, hInst, NULL)

 ir = SendMessage(hTBWnd, TB_BUTTONSTRUCTSIZE, 20, 0)
 ir= SendMessage(hTBWnd, TB_ADDBITMAP, 8, LOC(tbab))
 ir= SendMessage(hTBWnd, TB_ADDBUTTONS, 10, LOC(tbButtons))
 ir= ShowWindow(hTBWnd, SW_SHOW);
 InitToolBar=hTBWnd
end function InitToolBar
!*************************************************
```

Никаких изменений в оконную функцию пока вносить не будем. После создания исполняемого файла и его запуска на экране возникает картинка, показанная на рис. 12.1.

Рис. 12.1. Панель инструментов

УРОК

13. Закладки

Закладки (tab control) – это элементы управления, которые имитируют записную или адресную книжку. Окно этих элементов содержит собственно закладку (метку) с текстом или рисунком и рабочую область (страницу). Используя закладки, прикладная программа может употреблять одну и ту же область окна для отображения многостраничной информации. Каждая страница может содержать как информацию, так и группы средств управления, которые прикладная программа отображает, когда пользователь выбирает ту или иную страницу. Существует специальный вид закладки, который имеет вид кнопки. Нажатие кнопки приводит к выполнению команды без отображения страницы.

Закладки являются особым видом окна стиля WC_TABCONTROL. Этот класс будет зарегистрирован, если загружена библиотека общих элементов управления. Чтобы это гарантировать, надо использовать функцию **InitCommonControls()**. Мы уже говорили с вами об этом.

13.1. Создание диалога с закладками

Вы можете создавать закладки, вызывая функцию **CreateWindowEx()** и определяя класс окна WC_TABCONTROL. При создании окна следует указать стиль. В дополнение к стандартным для закладок установлены специальные стили, которые имеют префикс TCS_. В табл. 13.1 приведены значения стилей закладок.

Таблица 13.1. Стили закладок

Стиль	*Значение*	*Описание*
TCS_RIGHTJUSTIFY	#0000	Ширина каждой закладки увеличивается так, чтобы каждая строка закладок заполняла всю ширину страницы
TCS_SCROLLOPPOSITE	#0001	Стиль прокрутки
TCS_BOTTOM	#0002	Метка внизу

Стиль	*Значение*	*Описание*
TCS_RIGHT	#0002	Метка справа
TCS_FORCEICONLEFT	#0010	Устанавливает выравнивание иконки по левому краю, оставляя центрированную метку
TCS_FORCELABELLEFT	#0020	Устанавливает выравнивание и иконки и метку по левому краю
TCS_HOTTRACK	#0040	Не используется
TCS_VERTICAL	#0080	Вид закладок
TCS_BUTTONS	#0100	Создает закладки в виде кнопок
TCS_MULTILINE	#0200	Создает многострочные закладки
TCS_FIXEDWIDTH	#0400	Размеры закладок устанавливаются по ширине самой широкой из них
TCS_RAGGEDRIGHT	#0800	Ширина многострочных закладок не устанавливается равной ширине окна
TCS_FOCUSONBUTTONDOWN	#1000	Закладка активизируется нажатием клавиши мыши
TCS_OWNERDRAWFIXED	#2000	Устанавливает, что родительское окно прорисовывает закладки
TCS_TOOLTIPS	#4000	Создает закладки с подсказками
TCS_FOCUSNEVER	#8000	Запрещает активизацию закладки

После этого можно сформировать сами закладки и наделить их необходимыми свойствами непосредственно из приложения, используя сообщения. Однако удобнее воспользоваться средствами MDS.

Прежде всего создадим модальный диалог **ABOUTBOX**, выполнив цепочку действий, описанную ранее. После этого добавим закладки, используя окно редактора ресурсов с управляющими элементами (меню ***Controls***).

Выбираем в меню элемент ***Tab Control*** и перемещаем его на поле диалога. Двойным щелчком левой клавиши мыши открываем окно ***Tab Control Properties*** и устанавливаем стиль окна ***Visible***.

Переходим в окно ***Styles*** для того, чтобы установить стили кнопки. Это окно содержит несколько полей, значения которых описаны ниже.

Поле **Alignment** устанавливает стиль закладок и может иметь одно из следующих значений:

Right Justify (стиль TCS_RIGHTJUSTIFY) – ширина каждой закладки увеличивается так, чтобы каждая строка закладки заполняла всю ширину страницы;

Fixed Width (стиль TCS_FIXEDWIDTH) – размеры закладок устанавливаются по ширине самой широкой из них;

Ragged Right (стиль TCS_RAGGEDRIGHT) – ширина многострочных закладок не устанавливается равной ширине окна.

Поле **Focus** может принимать одно из следующих значений:

Default – разрешает использование клавиши табуляции;

On Button Down (стиль TCS_FOCUSONBUTTONDOWN)- закладка активизируется нажатием клавиши мыши;

Never (стиль TCS_FOCUSNEVER) – закладка никогда не активизируется.

Поле **Buttons** устанавливает, что закладки выглядят и действуют как кнопки (стиль TCS_BUTTONS).

Поле **Tool Tips** разрешает использование подсказок (стиль TCS_TOOLTIPS).

Поле **Multiline** устанавливает многострочный режим отображения закладок (стиль TCS_MULTILINE).

Поле **Force Icon Left** устанавливает выравнивание иконки по левому краю, оставляя центрированную метку (стиль TCS_FORCEICONLEFT).

Поле **Force Label Left** устанавливает выравнивание и иконки и метки по левому краю (стиль TCS_FORCELABELLEFT).

Поле **Owner Draw Fixed** устанавливает, что закладки прорисовывает родительское окно (стиль TCS_OWNERDRAWFIXED)

Поле **Border** создает рамку вокруг окна закладки.

В нашем примере ограничимся только установкой поля **Tool Tips**.

13.2. Взаимодействие с закладками

Управление закладками, так же как и любыми другими элементами управления, обеспечивается посредством обмена сообщениями.

Для управления закладками необходимо посылать сообщения окну. Кроме стандартных сообщений с префиксом WM_ существуют специализированные сообщения, имеющие префикс TCM_. Описание таких сообщений приведено в табл. 13.2.

Таблица 13.2. Сообщения, посылаемые закладкам

Сообщение	*Значение*	*Описание*
TCM_FIRST	#1300	
TCM_GETIMAGELIST	TCM_FIRST + 2	Получить дескриптор списка изображений
TCM_SETIMAGELIST	TCM_FIRST + 3	Связать список изображений с закладками; wParam = 0, lParam – дескриптор списка
TCM_GETITEMCOUNT	TCM_FIRST + 4	Определить число закладок
TCM_GETITEM	TCM_FIRST + 5	Получить информацию о закладке, wParam – индекс закладки, lParam – указатель на структуру типа TC_ITEM
TCM_SETITEM	TCM_FIRST + 6	Установить атрибуты закладки; wParam – индекс закладки, lParam – указатель на структуру типа TC_ITEM
TCM_INSERTITEM	TCM_FIRST + 7	Вставить закладку; wParam – индекс закладки, lParam – указатель на структуру типа TC_ITEM
TCM_DELETEITEM	TCM_FIRST + 8	Удалить закладку; wParam – индекс закладки, lParam =0
TCM_DELETEALLITEMS	TCM_FIRST + 9	Удалить все закладки; wParam= lParam =0
TCM_GETITEMRECT	TCM_FIRST + 10	Передает габаритные размеры закладки; wParam=0, lParam – указатель на структуру типа
TCM_GETCURSEL	TCM_FIRST + 11	Получить индекс закладки; wParam=lParam =0
TCM_SETCURSEL	TCM_FIRST + 12	Установить закладку; wParam – индекс закладки
TCM_HITTEST	TCM_FIRST + 13	Определяет, есть ли в заданной позиции закладка; wParam=0, lParam – указатель на структуру типа T_TCHITTESTINFO

Сообщение	*Значение*	*Описание*
TCM_SETITEMEXTRA	TCM_FIRST + 14	Установка размер дополнительных данных; wParam – число байтов, lParam =0
TCM_ADJUSTRECT	TCM_FIRST + 40	Получить размеры окна, согласованные с размером рабочей области, или наоборот; wParam – тип операции, lParam – указатель на структуру типа T_RECT
TCM_SETITEMSIZE	TCM_FIRST + 41	Устанавливает новые размеры метки
TCM_REMOVEIMAGE	TCM_FIRST + 42	Удалить изображение из списка; wParam – индекс изображения, lParam =0
TCM_SETPADDING	TCM_FIRST + 43	Устанавливает ширину свободной области вокруг изображения или метки; wParam – величина пробела, lParam =0
TCM_GETROWCOUNT	TCM_FIRST + 44	Передает число строк
TCM_GETTOOLTIPS	TCM_FIRST + 45	Передает дескриптор подсказки; wParam=lParam =0
TCM_SETTOOLTIPS	TCM_FIRST + 46	Связывает подсказки с закладками; wParam – дескриптор подсказки, lParam =0
TCM_GETCURFOCUS	TCM_FIRST + 47	Передает индекс активной закладки; wParam=lParam =0
TCM_SETCURFOCUS	TCM_FIRST + 48	Активизировать закладку
TCM_SETMINTABWIDTH	TCM_FIRST + 49	Установить минимальную ширину метки
TCM_DESELECTALL	TCM_FIRST + 50	Отменить выбор всех закладок

После создания средствами MDS шаблона диалога с закладками можно изменить стиль окна, добавить или уничтожить закладку, определить или установить состояние закладки и т. п. Большая часть установок производится диалоговой функцией на сообщение WM_INITDIALOG.

На примере закладок рассмотрим, как можно изменить стиль окна. В Win32 API для этого предусмотрена функция **SetWindowLong (hWnd, nIndex, dwNewLong)**, которая устанавливает атрибуты окна и оперирует с дополнитнльной памятью. Первый параметр, **hWnd** – это дескриптор окна, второй, **nIndex** – указатель для доступа к нужной величине, третий **dwNewLong** – новое значение. Если для установки новых атрибутов требуются сведения о старых, то следует воспользоваться функцией **GetWindowLong (hWnd, nIndex)**. Для изменения стиля закладок установите значение **nIndex**, равное GWL_STYLE. Новое значение можно сформировать в виде комбинации со старым.

При инициализации диалога необходимо произвести разметку закладок. Для этого надо использовать сообщение TCM_INSERTITEM. Параметр **wParm** задает индекс закладки, а **lParam** – указатель структуры типа TC_ITEM. Нумерация закладок начинается с нуля. Структура TC_ITEM имеет семь полей. Два поля зарезервированы для последующего использования. Поле **mask** указывает, в каком из полей структуры – **pszText**, **iImage** или **lParam** – содержится значение, определяющее элемент закладки. Это поле может содержать одно из приведенных в табл. 13.3 значений (или их комбинацию).

Таблица 13.3. Флаги, определяющие вид и поведение закладок

Флаг	*Значение*	*Описание*
TCIF_ALL	#0	Все три поля содержат информацию
TCIF_TEXT	#1	Данные содержатся в поле pszText
TCIF_IMAGE	#2	Данные содержатся в поле iImage
TCIF_RTLREADING	#4	Текст отображается справа налево
TCIF_PARAM	#8	Данные содержатся в поле lParam
TCIF_STATE	#10	

Если на метке страницы отображается текст, то поле **pszText** содержит указатель на текстовую строку, а в поле **chTextMax** указывается длина этой строки.

Чтобы связать изображение с закладками, в поле **iImage** указывается индекс изображения, связанного с данной страницей. Если списка изображений нет, то в этом поле надо обязательно установить -1.

Поле **lParam** может содержать любые данные, задаваемые пользователем.

Наиболее простой способ реализации отдельных закладок состоит в создании для каждой страницы индивидуального диалога. В примере, текст которого представлен ниже, трем закладкам поставлены в соответствие три немодальных диалога с идентификаторами IDD_DILOG1, IDD_DILOG2, IDD_DILOG3 и диалоговыми функциями **DL1, DL2, DL2**. Управление этими диалогами осуществляется функциями **CreateDialog()** и **DestroyWindow()**.

13.3. Нотификационные сообщения

При воздействии на закладку генерируется сообщение в форме WM_NOTIFY. Параметр **wParm** этого сообщения передает идентификатор элемента управления, а **lParam** содержит указатель на структуру типа T_NMHDR или T_TCKEYDOWN. Последняя в дополнение к полю **type(T_NMHDR) :: hdr** передает виртуальный код клавиши и флаг состояния клавиши. Нотификационные сообщения закладок приведены в табл. 13.4.

Таблица 13.4. Нотификационные сообщения закладок

Сообщение	*Значение*	*Описание*
TCN_FIRST	-550	
TCN_KEYDOWN	TCN_FIRST – 0	Сообщение о нажатии клавиши; lParam – указатель на структуру типа T_TCKEYDOWN
TCN_SELCHANGE	TCN_FIRST – 1	Сообщение о предстоящем изменении состояния; lParam – указатель на структуру типа T_NMHDR
TCN_SELCHANGING	TCN_FIRST – 2	Состояние изменено; lParam – указатель на структуру типа T_NMHDR

В дальнейшем закладки будут снабжены подсказками (см. следующий урок). Поэтому для обработки нотификационных сообщений используем расширенную структуру типа T_TOOLTIPTEXT, первым полем которой является **type(T_NMHDR) :: hdr**.

Нотификационное сообщение TCN_SELCHANGE используем в диалоговой функции для замены одного диалога на другой:

```
case(TCN_SELCHANGE)
  ir = DestroyWindow(hDl)
```

```
nTab = SendMessage(hTabWnd, TCM_GETCURSEL, 0, 0)
hDl=CreateDialog(hInst, IDD_DIALOG1+nTab, hTabWnd, DlgNb(nTab)).
```

Обработку сообщения TTN_NEEDTEXT обсудим позже.

Дополнение в модуль **commctrlty** выглядит следующим образом:

```
!====== TAB CONTROL ========================
!MS$IF .NOT. DEFINED (NOTABCONTROL)

!---------------------- имя класса ----------------------------
 character(16), parameter, public :: WC_TABCONTROL
="SysTabControl32"C

!-------------------- Стили закладок ---------------------
integer, parameter, public :: TCS_SCROLLOPPOSITE          = #0001
integer, parameter, public :: TCS_BOTTOM                  = #0002
integer, parameter, public :: TCS_RIGHT                   = #0002
integer, parameter, public :: TCS_FORCEICONLEFT           = #0010
integer, parameter, public :: TCS_FORCELABELLEFT          = #0020
integer, parameter, public :: TCS_HOTTRACK                = #0040
integer, parameter, public :: TCS_VERTICAL                = #0080
integer, parameter, public :: TCS_TABS                    = #0000
integer, parameter, public :: TCS_BUTTONS                 = #0100
integer, parameter, public :: TCS_SINGLELINE              = #0000
integer, parameter, public :: TCS_MULTILINE               = #0200
integer, parameter, public :: TCS_RIGHTJUSTIFY            = #0000
integer, parameter, public :: TCS_FIXEDWIDTH              = #0400
integer, parameter, public :: TCS_RAGGEDRIGHT             = #0800
integer, parameter, public :: TCS_FOCUSONBUTTONDOWN       = #1000
integer, parameter, public :: TCS_OWNERDRAWFIXED          = #2000
integer, parameter, public :: TCS_TOOLTIPS                = #4000
integer, parameter, public :: TCS_FOCUSNEVER              = #8000

!-------------------------Сообщения ----------------------------
integer, parameter, public  :: TCM_FIRST                  = #1300
integer, parameter, public  :: TCM_GETIMAGELIST           = TCM_FIRST + 2
integer, parameter, public  :: TCM_SETIMAGELIST           = TCM_FIRST + 3
integer, parameter, public  :: TCM_GETITEMCOUNT           = TCM_FIRST + 4
integer, parameter, public  :: TCM_GETITEM                = TCM_FIRST + 5
integer, parameter, public  :: TCM_SETITEM                = TCM_FIRST + 6
integer, parameter, public  :: TCM_INSERTITEM             = TCM_FIRST + 7
integer, parameter, public  :: TCM_DELETEITEM             = TCM_FIRST + 8
integer, parameter, public  :: TCM_DELETEALLITEMS         = TCM_FIRST + 9
integer, parameter, public  :: TCM_GETITEMRECT            = TCM_FIRST + 10
integer, parameter, public  :: TCM_GETCURSEL              = TCM_FIRST + 11
```

```
integer, parameter, public  :: TCM_SETCURSEL        = TCM_FIRST + 12
integer, parameter, public  :: TCM_HITTEST          = TCM_FIRST + 13
integer, parameter, public  :: TCM_SETITEMEXTRA     = TCM_FIRST + 14
integer, parameter, public  :: TCM_ADJUSTRECT       = TCM_FIRST + 40
integer, parameter, public  :: TCM_SETITEMSIZE      = TCM_FIRST + 41
integer, parameter, public  :: TCM_REMOVEIMAGE      = TCM_FIRST + 42
integer, parameter, public  :: TCM_SETPADDING       = TCM_FIRST + 43
integer, parameter, public  :: TCM_GETROWCOUNT      = TCM_FIRST + 44
integer, parameter, public  :: TCM_GETTOOLTIPS      = TCM_FIRST + 45
integer, parameter, public  :: TCM_SETTOOLTIPS      = TCM_FIRST + 46
integer, parameter, public  :: TCM_GETCURFOCUS      = TCM_FIRST + 47
integer, parameter, public  :: TCM_SETCURFOCUS      = TCM_FIRST + 48
integer, parameter, public  :: TCM_SETMINTABWIDTH = TCM_FIRST + 49
integer, parameter, public  :: TCM_DESELECTALL      = TCM_FIRST + 50

!-------------------- Нотификационные сообщения ---------------
integer, parameter, public  :: TCN_KEYDOWN          = (TCN_FIRST – 0)
integer, parameter, public  :: TCN_SELCHANGE        = (TCN_FIRST – 1)
integer, parameter, public  :: TCN_SELCHANGING      = (TCN_FIRST – 2)

!------------------------ Флаги  внешнего вида ------------------
integer, parameter, public :: TCIF_ALL              = 0
integer, parameter, public :: TCIF_TEXT             = 1
integer, parameter, public :: TCIF_IMAGE            = 2
integer, parameter, public :: TCIF_RTLREADING       = 4
integer, parameter, public :: TCIF_PARAM            = 8
integer, parameter, public :: TCIF_STATE            = 16
!-------------------------------------------------------------
integer, parameter, public :: TCHT_NOWHERE          = #0001
integer, parameter, public :: TCHT_ONITEMICON       = #0002
integer, parameter, public :: TCHT_ONITEMLABEL      = #0004
integer, parameter, public :: TCHT_ONITEM           = #0006

!---------------- структуры ----------------------------------

type   TC_ITEM
  integer(4)  ::  mask
  integer(4)  ::  lpReserved1
  integer(4)  ::  lpReserved2
  integer(4)  ::  pszText
  integer(4)  ::  cchTextMax
  integer(4)  ::  iImage
  integer(4)  ::  lParam
end type TC_ITEM
```

```
type T_HELPINFO
 integer(4)   ::  cbSize
 integer(4)   ::  iContextType
 integer(4)   ::  iCtrlId
 integer(4)   ::  hItemHandle
 integer(4)   ::  dwContextId
 type (T_POINT):: MousePos
end type T_HELPINFO

type T_TCKEYDOWN
 type(T_NMHDR) ::  hdr
 integer(4)   ::  wVKey
 integer(4)   ::  flags
end type T_TCKEYDOWN

type T_TCHITTESTINFO
 type(T_POINT) :: pt
 integer(4)   :: flags
end type T_TCHITTESTINFO

!MS$ENDIF ! NOTABCONTROL
```

13.4. Пример диалога с закладками

Чтобы продемонстрировать изложенное выше, дополним диалог **ABOUT** закладками. Пусть эти закладки должны иметь имена: **Содержание**, **Как пользоваться**, **О программе**. Заголовки в приведенной ниже диалоговой функции заданы как параметры. Для удобства нумерация элементов строкового массива начинается с нуля. Каждой закладке ставится в соответствие свой собственный диалог (диалоговые функции **DL1**, **DL2**, **DL3**). Все функции, обслуживающие диалог с закладками, помещены в модуль **MyPr_1inc**.

Разметка закладок проводится в ответ на команду WM_INITDIALOG. Функция **GetDlgItem(hDlg, IDC_TAB1)** возвращает дескриптор, который необходим для общения с закладками. Далее заполняются поля структуры **type** (TC_ITEM) **:: Tci**, которая предварительно объявляется в модуле **MyPr_1inc**.

В ответ на сообщение TCM_GETCURSEL получаем порядковый номер закладки (нумерация начинается с нуля). После чего создаем необходимый диалог.

При воздействии на закладку генерируется сообщение WM_NOTIFY. В диалоге обрабатываются нотификационные коды TCN_ SELCHANGING, TCN_SELCHANGE и TTN_NEEDTEXT. Первый код используется перед тем, как изменится выбор закладки. В ответ на него надо, если это необходимо, сохранить данные диалога. После выбора новой закладки передается второй код. В этом случае закрывается старый диалог и создается новый. Наконец, последний код используется для того, чтобы снабдить закладки подсказками (см. следующий урок). Текст диалоговых функций для работы с закладками представлен ниже.

```
!****************************************************************
integer*4 function About (hDlg, message, wParam, lParam)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_About@16' :: About

integer             :: hDlg, message, wParam, lParam
integer             :: ir, nTab
integer(4), dimension(0:2):: DlgNb

character(18), dimension(0:2), parameter :: szTabItem = &
          (/"Содержание     "C, &
            "Как пользоваться"C, &
            "О программе     "C/)

type (T_TOOLTIPTEXT) :: TText
POINTER(ip, TText)

!---------------------------------------------------------
lParam = lParam

select case (message)
 case (WM_INITDIALOG)

  !---- Разметка закладок ----------------------
  hTabWnd  = GetDlgItem(hDlg, IDC_TAB1)
  ir=GetClientRect(hTabWnd, rcDlg)
  tci%mask = TCIF_TEXT
  tci%iImage = -1
  do i=0, 2
   tci%pszText =LOC(szTabItem(i))
   ir = SendMessage(hTabWnd, TCM_INSERTITEM, i, LOC(Tci))
  end do
```

```
 DlgNb(0)=   LOC(DL1)
 DlgNb(1)=   LOC(DL2)
 DlgNb(2)=   LOC(DL3)

 nTab = SendMessage(hTabWnd, TCM_GETCURSEL, 0, 0)
 hDl=CreateDialog(hInst, IDD_DIALOG1+nTab, hTabWnd, DlgNb(nTab))

 About = 1
 return

case (WM_NOTIFY)
 ip = TRANSFER(lParam, ip)
 select case(TText%hdr%code)
  case(TTN_NEEDTEXT)
   nTab = TText%hdr%idFrom
   TText%lpszText=LOC(szTabItem(nTab))
  case(TCN_SELCHANGING)
  case(TCN_SELCHANGE)
   ir = DestroyWindow(hDl)
    nTab = SendMessage(hTabWnd, TCM_GETCURSEL, 0, 0)
   hDl=CreateDialog(hInst, IDD_DIALOG1+nTab, hTabWnd, DlgNb(nTab))

 end select
case (WM_COMMAND)
 select case (wParam)
   case(IDOK)
    ir = DestroyWindow(hDl)
     ir = EndDialog(hDlg, wParam)
 end select
end select

About = 0
end function About

!***************************************************************
integer*4 function DL1 (hDlg, message, wParam, lParam)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_DL1@16' :: DL1
integer hDlg, message, wParam, lParam
integer ir

lParam = lParam

select case (message)
 case (WM_INITDIALOG)
      DL1 = 1
      return
```

```
   case (WM_COMMAND)
      if (wParam == IDOK) then
      end if
end select

DL1 = 0
end function DL1
!**************************************************************
integer*4 function DL2 (hDlg, message, wParam, lParam)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_DL2@16' :: DL2

integer hDlg, message, wParam, lParam
integer ir

lParam = lParam

select case (message)
 case (WM_INITDIALOG)
      DL2 = 1
      return

   case (WM_COMMAND)
      if (wParam == IDOK) then
      end if
end select

DL2 = 0
end function DL2
!**************************************************************
integer*4 function DL3 (hDlg, message, wParam, lParam)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_DL3@16' :: DL3

integer hDlg, message, wParam, lParam
integer ir

lParam = lParam

select case (message)
 case (WM_INITDIALOG)
      DL3 = 1
      return

   case (WM_COMMAND)
      if (wParam == IDOK) then
      end if
end select
```

```
DL3 = 0
end function DL3
!*****************************************************************************/
```

Результат выполнения программы показан на рис. 13.1.

Рис. 13.1. Диалог с закладками

УРОК

14. Подсказки

Подсказка (tooltip) – это маленькое окно, которое появляется на несколько секунд на кнопках некоторых панелей инструментов и содержит краткое описание назначения кнопок. Подсказки помогают пользователю при работе с приложением.

Подсказка появляется только тогда, когда пользователь помещает курсор на кнопку панели инструментов и задерживает его там приблизительно на полсекунды. Она появляется рядом с курсором и исчезает, как только пользователь нажмет кнопку мыши или удалит курсор с панели инструментов. Одна и та же подсказка может обслуживать любое число инструментальных средств.

Вообще-то, подсказкой можно снабдить любое окно, элемент управления или просто заданную область внутри окна. Однако для панели инструментов имеются специализированные средства, которые облегчают создание подсказок. Именно поэтому мы начнем с подключения подсказок к инструментальной панели.

14.1. Подключение подсказок к инструментальной панели

Для подключения подсказок к кнопкам панели инструментов достаточно задать стиль TBSTYLE_TOOLTIPS (см. предыдущий урок). Устанавливая этот стиль, вы снабжаете инструментальную панель подсказками и одновременно разрешаете посылку сообщения WM_NOTIFY, параметр **lParam** которого содержит указатель на структуру TOOLTIPTEXT.

Структура TOOLTIPTEXT определена в файле **commctrl.h**. Для того чтобы можно было использовать подсказки в программах, написанных на языке Фортран-90, следует включить в модуль **commctrlty** описание структуры типа TOOLTIPTEXT:

```
type   T_TOOLTIPTEXT
 type (T_NMHDR)  :: hdr
 integer(4)      :: lpszText
 character(80)   :: szText
 integer(4)      :: hInst
```

```
 integer(4)          :: uFlags
end type  T_TOOLTIPTEXT.
```

Первым полем структуры является структура T_NMHDR:

```
type  T_NMHDR
   integer(4)  :: hwndFrom
   integer(4)  :: idFrom
   integer(4)  :: code
end type T_NMHDR
```

Если запрашивается подсказка, то поле **code** будет содержать сообщение TTN_NEEDTEXT, а поле **idFrom** – идентификатор кнопки, для которой требуется подсказка. Текст подсказки можно задать одним из трех способов: скопировать его в поле **szText** структуры T_TOOLTIPTEXT, либо записать в поле **lpszText** адрес текстовой строки, либо, наконец, задать идентификатор ресурса строки. В последнем случае адрес записывается в поле **lpszText**, а поле содержит дескриптор приложения. Очевидно, что первый способ проще.

Для того чтобы появилась подсказка, необходимо добавить в оконную функцию обработку запросов. Фрагмент текста файла **MyPr_1.f90**, ответственный за эту процедуру, выглядит следующим образом:

```
type (T_TOOLTIPTEXT)  TText
POINTER(ip, TText)

!---------- начало процедуры обработки сообщений ---------
select case (message)
 case (WM_NOTIFY)
  ip = TRANSFER(lParam, ip)
  if(TText%hdr%code==TTN_NEEDTEXT) then

  ir= LoadString(hInst, TText%hdr%idFrom, szBuf, 16)
   TText%lpszText=LOC(szBuf)
  end if
```

Подсказки загружаются из ресурса строк. Для этого используется функция **LoadString()**. Для удобства в приложении установлены одни и те же идентификаторы для строк, кнопок и команд меню.

Для того чтобы обратиться к структуре **T_TOOLTIPTEXT** по указателю **lParam** используем целочисленный указатель **ip**, который связываем с переменной **Ttext**. Затем преобразуем адрес в целочисленный указатель с помощью функции **TRANSFER(lParam, ip)**.

Тексты подсказок создаем в виде ресурса. Добавим новый ресурс в проект, используя средства MDS. Для этого выполним следующую цепочку действий: ***Insert – Resource – Sting Table***. На экране появится окно редактора. Прежде чем заполнять строки вернемся в окно проекта (***Project Workspace***), откроем окно свойств ***Sting Table Properties*** и установим язык ресурса. Теперь можно вернуться в окно редактора и заполнить таблицу.

Теперь можно создать исполнительный модуль. Если текст не содержит ошибок, то после запуска приложения вы сможете получить на экране изображение, показанное на рис. 14.1.

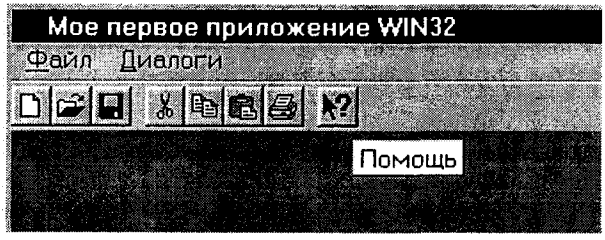

Рис. 14.1. Подсказки в инструментальной панели

Не представляет особой сложности подключение подсказок к диалогу с закладками. Для этого достаточно обработать сообщение WM_NOTIFY, которое посылают закладки. В ответ на код TTN_NEEDTEXT надо выполнить те же процедуры, что и для подсказок, связанных с инструментальной панелью.

Следующий фрагмент текста диалога **ABOUTBOX** обеспечивает обработку кода TTN_NEEDTEXT:

```
case (WM_NOTIFY)
 ip = TRANSFER(lParam, ip)
 select case(TText%hdr%code)
  case(TTN_NEEDTEXT)
   nTab = TText%hdr%idFrom
   TText%lpszText=LOC(szTabItem(nTab)).
```

В данном случае текст подсказки передается непосредственно в виде строки. Для удобства строки сгруппированы в массив **character(18), dimension(0:2), parameter :: szTabItem**. На рис. 14.2 показан диалог с подсказками.

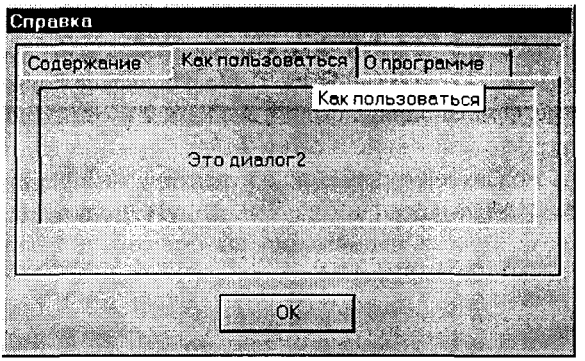

Рис. 14.2. Подсказка в диалоге с закладками

14.2. Инициализация подсказок

В общем случае использования подсказок требуется инициализация этого общего элемента управления. Поскольку подсказка – это окно особого типа, то для этой цели используются функции **CreateWindow()** или **CreateWindowEx()**. В качестве класса окна задается TOOLTIPS_CLASS, который предварительно определяется в модуле **commctrlty**: **character(17), parameter, public :: TOOLTIPS_CLASS = "tooltips_ class32"C**. Класс будет наверняка зарегистрирован, если загружена библиотека общих элементов управления, т. е. если в тело вашей программы включена функция **InitCommonControls()**. В противном случае нет гарантии, что общие элементы управления будут доступны вашему приложению.

Как и для всякого окна, для подсказки должен быть задан стиль. Кроме стандартных стилей есть несколько специальных. Все они имеют префикс TTS_.

Размеры и позиция окна для каждой подсказки устанавливаются автоматически. Высота согласуется с высотой выбранного шрифта, а ширина – с длиной строки. Подсказки могут иметь два специфических стиля: TTS_ALWAYSTIP и TTS_NOPREFIX.

Стиль TTS_ALWAYSTIP создает подсказку, которая появляется, когда курсор находится на инструментальном средстве, независимо от того, является ли окно владельца подсказки активным или неактивным. В противном случае подсказка появляется, только если окно владельца активно.

Если стиль TTS_NOPREFIX не установлен, то символ & не воспроизводится в тексте подсказки. Это необходимо, когда одна и та же строка используется и в подсказках и в меню.

Стили WS_POPUP и WS_EX_TOOLWINDOW устанавливаются автоматически.

14.3. Взаимодействие с подсказками

Взаимодействие с подсказками осуществляется посредством обмена сообщениями. Все специфические сообщения подсказок имеют префикс TTM_. Наиболее важные типы сообщений приведены в табл. 14.1.

Таблица 14.1. Сообщения, посылаемые подсказками

Сообщение	*Значение*	*Описание*
TTM_ACTIVATE	WM_USER + 1	Активизация подсказки; wParam = .TRUE., lParam=0
TTM_SETDELAYTIME	WM_USER + 3	Задает временные интервалы; lParam – новое значение
TTM_ADDTOOL	WM_USER + 4	Добавляет инструмент; lParam – адрес T_TOOLINFO
TTM_DELTOOL	WM_USER + 5	Уничтожает инструмент; lParam – адрес T_TOOLINFO
TTM_NEWTOOLRECT	WM_USER + 6	Задает границы окна подсказки; lParam – адрес T_TOOLINFO
TTM_RELAYEVENT	WM_USER + 17	Передает сообщение от мыши окну подсказки; lParam – адрес T_ MSG
TTM_GETTOOLINFO	WM_USER + 8	Возвращает информацию об инструменте; lParam – адрес T_TOOLINFO
TTM_SETTOOLINFO	WM_USER + 8	Устанавливает информацию об инструменте

Сообщение	*Значение*	*Описание*
TTM_HITTEST	WM_USER + 10	Устанавливает, попадает ли точка на инструмент, и передает информацию об инструменте; lParam – адрес T_TOOLINFO
TTM_GETTEXT	WM_USER + 11	Передает текст подсказки; lParam – адрес T_TOOLINFO
TTM_UPDATETIPTEXT	WM_USER + 12	Обновляет текст подсказки; lParam – адрес T_TOOLINFO
TTM_GETTOOLCOUNT	WM_USER + 13	Передает число инструментов данной подсказки
TTM_ENUMTOOLS	WM_USER + 14	Перебирает инструменты данной подсказки; wParam – номер инструмента, lParam – адрес T_TOOLINFO
TTM_GETCURRENTTOOL	WM_USER + 15	Передает информацию о текущем инструменте; lParam – адрес T_TOOLINFO
TTM_WINDOWFROMPOINT	WM_USER + 16	Позиция подсказки задается параметрами сообщения; lParam – адрес T_POINT

Подсказка может быть или активна, или неактивна. Если она активна, то появляется, когда курсор находится на элементе. Если же нет, то подсказка не появится, даже если курсор находится на инструменте. Посылая сообщение TTM_ACTIVATE, можно изменить состояние подсказки.

Каждая подсказка может поддерживать любое число инструментальных средств. Чтобы поддерживать какой-либо специфический инструмент, вы должны зарегистрировать его, посылая сообщение TTM_ADDTOOL. Сообщение содержит адрес структуры T_TOOLINFO, которая включает информацию о подсказке. Эта структура должна быть определена в модуле **commctrlty** следующим образом:

```
type T_TOOLINFO
 integer(4)  :: cbSize
```

```
integer(4)  :: uFlags
integer(4)  :: hwnd
integer(4)  :: uId
type (T_RECT):: rect
integer(4)  :: hInst
integer(4)  :: lpszText
end type T_TOOLINFO
```

Первое поле содержит размер структуры в байтах. Второе, **Uflags** – это набор битовых флагов. Оно может содержать комбинацию значений, некоторые из которых приведены в табл. 14.2.

Таблица 14.2. Флаги структуры T_TOOLINFO

Флаг	*Значение*	*Действие*
TTF_IDISHWND	#0001	Указывает, что uId – дескриптор окна. В противном случае uId – идентификатор инструмента
TTF_CENTERTIP	#0002	Центрирует подсказку
TTF_RTLREADING	#0004	Только Win95. Отображает текст справа налево
TTF_SUBCLASS	#0010	Устанавливает, что подсказка является подклассом инструмента и обрабатывает сообщение WM_MOUSEMOVE.

Поле **hwnd** – дескриптор окна инструмента. Если поле **lpszText** включает значение LPSTR_TEXTCALLBACK, то **hwnd** – дескриптор окна, которое получает сообщение TTN_NEEDTEXT.

Поле **uId** задает идентификатор инструмента. Если **uFlags** включает значение TTF_IDISHWND, то **uId** задает программу обработки инструмента.

Поле **rect** – это координаты прямоугольника, обрамляющего инструмент. Координаты задаются относительно левого верхнего угла рабочей области пользовательского окна с дескриптором **hwnd**. Если **uFlags** включает значение TTF_IDISHWND, то это поле игнорируется.

Поле **hInst** – дескриптор приложения. Принимается во внимание, если только поле **lpszText** содержит идентификатор ресурса таблицы строк.

Последнее поле, **lpszText** – это указатель на буфер, который содержит текст для инструмента или идентификатора ресурса таблицы строк.

Подсказка поддерживает инструментальные средства, выполненные как в виде окна, так и в виде просто прямоугольной области окна пользо-

вателя. В последнем случае **hwnd** должен содержать дескриптор окна, которое содержит область, а поле **rect** – ее координаты. Кроме того, в поле **uId** нужно задать идентификатор инструмента.

Когда вы добавляете инструмент, выполненный в виде окна, в поле **uId** заносится дескриптор окна инструмента. В поле **uFlags** нужно занести значение TTF_IDISHWND, которое указывает, что поле **uId** – это дескриптор окна.

Поле **lpszText** структуры TOOLINFO содержит адрес строки текста подсказки. Текст можно оперативно изменять после добавления инструмента, используя сообщение TTM_UPDATETIPTEXT. Если же старшее слово **lpszText** – нуль, то слово младшего разряда должно быть идентификатором строкового ресурса. В этом случае поле **hInst** не равно нулю.

Программа может получить текст подсказки из структуры T_TOOLINFO с помощью сообщения TTM_GETTEXT.

Для того чтобы отобразить подсказку, необходимо получить сообщение от мыши. Система посылает сообщения только тому окну, на котором находится курсор. Поэтому программа должна использовать сообщение TTM_RELAYEVENT для того, чтобы передать сообщения мыши окну подсказки. Окно подсказки обрабатывает только следующие сообщения:

WM_LBUTTONDOWN,
WM_MOUSEMOVE,
WM_LBUTTONUP,
WM_RBUTTONDOWN,
WM_MBUTTONDOWN,
WM_RBUTTONUP,
WM_MBUTTONUP.

Если инструмент выполнен как прямоугольная область в определенном прикладной программой окне, процедура окна получает сообщения мыши и может передавать их подсказке. Однако если инструмент выполнен как системное окно (например, в виде кнопки), сообщения мыши посылаются окну и недоступны прикладной программе. В этом случае следует либо перехватывать сообщение до его выполнения (**Hooking**), либо заменять оконную функцию (**Subclassing**).

Функция **SetWindowsHookEx()** помещает определенную в приложении процедуру перехвата сообщений в очередь. В результате ваша прикладная программа получает возможность контролировать некоторые типы событий.

Для замены оконной функции используется функция **SetWindowLong()** с флагом GWL_WNDPROC. Процедура подкласса может входить в модуль

прикладной программы или в DLL. Функция **SetWindowLong()** возвращает адрес процедуры оригинала окна. Прикладная программа должна сохранить этот адрес и использовать его для передачи прерванных сообщений первоначальной оконной процедуре с помощью функции **CallWindowProc()**. За более подробными сведениями следует обратиться к справочной системе MDS.

Когда окно подсказки получает сообщение WM_MOUSEMOVE, оно определяет, находится ли курсор в прямоугольнике, обрамляющем инструмент. Если – да, то включается таймер. В конце заданного интервала позиция курсора снова проверяется, и если она не изменилась, то текст подсказки отображается на экране. Подсказка остается на экране до тех пор, пока окно подсказки не получит сообщения об изменении состояния мыши.

Временные интервалы устанавливаются сообщением TTM_SETDELAYTIME. Параметр **lParam** задает интервал времени в миллисекундах, а **wParam** задает процедуру установки: TTDT_AUTOMATIC – все интервалы вычисляются автоматически, TTDT_RESHOW – устанвливается интервал повторного отображения, TTDT_AUTOPOP – устанавливается интервал отображения, TTDT_INITIAL – устанавливается интервал инициализации.

Если прикладная программа добавляет инструмент, выполненный в виде прямоугольной области, то ее размер и позиция устанавливаются сообщением TTM_NEWTOOLRECT. Для инструментов в виде окна в этом нет необходимости, поскольку оконная функция самостоятельно определяет положение курсора на инструменте.

Сообщения TTM_GETCURRENTTOOL и TTM_GETTOOLINFO используются для получения информации об инструменте через структуру T_TOOLINFO. Изменить информацию об инструменте можно, посылая сообщение TTM_SETTOOLINFO. Наконец, сообщение TTM_DELTOOL удаляет инструмент.

Следующий фрагмент текста показывает, как организовать подсказки для некоторого количества прямоугольных областей.

```
! Создаем окно подсказок

hwndTT = CreateWindow(TOOLTIPS_CLASS, NULL, TTS_ALWAYSTIP,          &
CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, &
    NULL, NULL, hInst, NULL)

! Разбиваем рабочую область окна на прямоугольные области
! и добавляем для каждой из них подсказку.
```

```
do row = 0, MAX_ROWS
 do col = 0, MAX_COLS
        ti.cbSize = 40
        ti.uFlags = 0
        ti.hwnd = hwndOwner
        ti.hinst = hInst
        ti.uId =  iD
        ti.lpszText = LOC(szTips(id))
        ti.rect.left = col * CX_COLUMN
        ti.rect.top = row * CY_ROW
        ti.rect.right = ti.rect.left + CX_COLUMN
        ti.rect.bottom = ti.rect.top + CY_ROW

  if (.NOT.SendMessage(hwndTT, TTM_ADDTOOL, 0,
                       LOC(ti)) return
 end do
end do
```

В примере создается сетка прямоугольников в рабочей области окна, а затем с помощью сообщения TTM_ADDTOOL каждый прямоугольник снабжается подсказкой. Следует иметь в виду, что процедура родительского окна должна обрабатывать сообщения мыши и передавать их, используя сообщение TTM_RELAYEVENT.

Поскольку подсказка является общим элементом управления, она посылает нотификационные сообщения в форме WM_NOTIFY. Самое важное из них – это сообщение TTN_NEEDTEXT, о котором уже шла речь выше. Кроме того, перед появлением на экране подсказка посылает родительскому окну сообщение TTN_SHOW, а сообщение TTN_POP – перед исчезновением. Эти сообщения можно использовать для оперативного взаимодействия с закладкой.

14.4. Использование подсказок в диалогах

Организация системы подсказок во многих случаях связана с диалогами. Набор подпрограмм, которые снабжают диалоговое окно подсказками, представлен ниже.

Функция **DoCreateDialogTooltip()** создает закладки и использует функцию **EnumChildWindows()**, чтобы зарегистрировать средства управления диалоговым окном. Процедура **EnumChildProc()** связывает с каждым элементом управления подсказку. Для каждой подсказки диалоговое окно объявляется родительским и включается значение LPSTR_ TEXTCALLBACK. В результате диалоговое окно получает сооб-

щение WM_NOTIFY, которое содержит код TTN_NEEDTEXT. Обработка сообщения осуществляется функцией **OnWMNotify**. Однако можно обрабатывать нотификационные сообщения в теле диалоговой функции так, как мы это делали ранее.

Закладка должна получить сообщения мыши, которые система посылает элементам управления. Для этого в функции **DoCreateDialogTooltip()** организуется процедура перехвата типа WH_GETMESSAGE. Процедура перехвата **GetMSGProc()** контролирует поток сообщений и пересылает сообщения, предназначенные для одного из элементов управления, подсказке.

Тексты процедур, которые снабжают диалог подсказками, выглядят таким образом.

```
logical function DoCreateDialogTooltip()
    hwndTT = CreateWindowEx(0, TOOLTIPS_CLASS,  NULL,              &
    TTS_ALWAYSTIP, CW_USEDEFAULT, CW_USEDEFAULT,                   &
    CW_USEDEFAULT,CW_USEDEFAULT,  hDlg, NULL, hInst, NULL);

  if (hwndTT == NULL)  then
   DoCreateDialogTooltip  = .FALSE.
   return\
  end if
!Регистрация дочерних окон
  if (.NOT.EnumChildWindows(hDlg, LOC(EnumChildProc), 0)) then
   DoCreateDialogTooltip  = .FALSE.
   return\
  end if
! Организация процедуры перехвата сообщений мыши и передачи
! их диалоговой процедуре.
   hhk = SetWindowsHookEx(WH_GETMESSAGE, GetMsgProc,              &
                          NULL, GetCurrentThreadId());

  if (hhk == NULL) then
   DoCreateDialogTooltip  = .FALSE.
   return\
  end if
    DoCreateDialogTooltip  = .TRUE.
end function
!-------------------------------------------------------------------------------
logical function EnumChildProc(hwndCtrl, lParam)
!EmumChildProc – регистрирует элементы управления и связывает
!их с подсказками, используя TTM_ADDTOOL .
!_____________________________________________________________
```

```
  type (T_ TOOLINFO)  :: ti
   charecter(64)          :: szClass

   ir =  GetClassName(hwndCtrl, szClass, 64);
   if (szClass == "STATIC") then
      ti.cbSize =40
      ti.uFlags = TTF_IDISHWND
      ti.hwnd = hDlg
      ti.uId =   hwndCtrl;
      ti.hinst = 0;
      ti.lpszText = LPSTR_TEXTCALLBACK;
      SendMessage(hwndTT, TTM_ADDTOOL, 0, LOC(ti))
   end if
 EnumChildProc  = .TRUE.
end function
!-------------------------------------------------------------------------
integer(4) function GetMsgProc(nCode,  wParam,  lParam)
!MS$ATTRIBUTES STDCALL, ALIAS : '_ GetMsgProc @12' :: GetMsgProc
! GetMsgProc –  взаимодействует с потоком сообщений и передает
! сообщения мыши диалоговому окну.
! lParam – адрес структуры T_MSG
!_____________________________________________________

type (T_MSG)         :: lpmsg
POINTER(ip, lpmsg)

ip = TRANSFER(lParam, ip)
 if (nCode < 0 .and. .NOT.(IsChild(hDlg, lpmsg%hwnd)))  theh
   GetMsgProc  =  CallNextHookEx(g_hhk, nCode, wParam, lParam)
    return
 end if
   select case(lpmsg%message)
      case WM_MOUSEMOVE
      case WM_LBUTTONDOWN
      case WM_LBUTTONUP
      case WM_RBUTTONDOWN
      case WM_RBUTTONUP
         if (hwndTT /= NULL) then
            type(T_MSG)   :: msg

            msg%lParam = lpmsg%lParam
            msg%wParam = lpmsg%wParam
            msg%message = lpmsg&message
            msg&hwnd = hwnd
```

```
            SendMessage(hwndTT, TTM_RELAYEVENT, LOC(msg))
          end if
        case DEFAULT
      end select
  GetMsgProc  =CallNextHookEx(hhk, nCode, wParam, lParam)
end function
 !-------------------------------------------------------------------
integer function OnWMNotify(lParam)
 !___________________________________________________
integer                         :: idCtrl

type(T_TOOLTIPTEXT) :: lpttt
POINTER(ip, lpttt)

ip = TRANSFER(lParam, ip)
      if (lpttt%hdr%code) == TTN_NEEDTEXT) then
       select case (lpttt%hdr%from)
         case (ID_HORZSCROLL)
            lpttt%lpszText = "A horizontal scroll bar.";
            return;

         case (ID_CHECK)
            lpttt%lpszText = "A check box.";
            return;

         case (ID_EDIT)
            lpttt%lpszText = "An edit control.";
            return;
       end select
    end if
end function
```

Для того чтобы использовать все возможности окон подсказок, необходимо дополнить текст модуля **commctrlty** следующим фрагментом:

```
!====== TOOLTIPS CONTROL ==========================================

!MS$IF .NOT. DEFINED  (NOTOOLTIPS)

integer, parameter, public :: TTN_FIRST    = -520  ! tooltips
integer, parameter, public :: TTN_LAST     = -549

!------------------- имя класса -------------------------------
 character(17), parameter, public ::  TOOLTIPS_CLASS  = "tooltips_class32"C

integer, parameter, public  :: TTS_ALWAYSTIP    = 1
integer, parameter, public  :: TTS_NOPREFIX     = 2
```

```
!---------- Сообщения ----------------------------------------
integer, parameter, public :: TTM_ACTIVATE          = (WM_USER + 1)
integer, parameter, public :: TTM_SETDELAYTIME      = (WM_USER + 3)
integer, parameter, public :: TTM_ADDTOOL           = (WM_USER + 4)
integer, parameter, public :: TTM_DELTOOL           = (WM_USER + 5)
integer, parameter, public :: TTM_NEWTOOLRECT       = (WM_USER + 6)
integer, parameter, public :: TTM_RELAYEVENT        = (WM_USER + 7)
integer, parameter, public :: TTM_GETTOOLINFO       = (WM_USER + 8)
integer, parameter, public :: TTM_SETTOOLINFO       = (WM_USER + 9)
integer, parameter, public :: TTM_HITTEST           = (WM_USER +10)
integer, parameter, public :: TTM_GETTEXT           = (WM_USER +11)
integer, parameter, public :: TTM_UPDATETIPTEXT     = (WM_USER +12)
integer, parameter, public :: TTM_GETTOOLCOUNT      = (WM_USER +13)
integer, parameter, public :: TTM_ENUMTOOLS         = (WM_USER +14)
integer, parameter, public :: TTM_GETCURRENTTOOL    = (WM_USER + 15)
integer, parameter, public :: TTM_WINDOWFROMPOINT   = (WM_USER + 16)
integer, parameter, public :: TTM_TRACKACTIVATE     = (WM_USER + 17)
integer, parameter, public :: TTM_TRACKPOSITION     = (WM_USER + 18)
integer, parameter, public :: TTM_SETTIPBKCOLOR     = (WM_USER + 19)
integer, parameter, public :: TTM_SETTIPTEXTCOLOR   = (WM_USER + 20)
integer, parameter, public :: TTM_GETDELAYTIME      = (WM_USER + 21)
integer, parameter, public :: TTM_GETTIPBKCOLOR     = (WM_USER + 22)
integer, parameter, public :: TTM_GETTIPTEXTCOLOR   = (WM_USER + 23)
integer, parameter, public :: TTM_SETMAXTIPWIDTH    = (WM_USER + 24)
integer, parameter, public :: TTM_GETMAXTIPWIDTH    = (WM_USER + 25)
integer, parameter, public :: TTM_SETMARGIN         = (WM_USER + 26)
integer, parameter, public :: TTM_GETMARGIN         = (WM_USER + 27)

!-------------------- Нотификационные сообщения ----------------
integer, parameter, public :: TTN_GETDISPINFO = (TTN_FIRST - 0)
integer, parameter, public :: TTN_SHOW        = (TTN_FIRST - 1)
integer, parameter, public :: TTN_POP         = (TTN_FIRST - 2)

integer, parameter, public :: TTN_NEEDTEXT    = TTN_GETDISPINFO

!------------ структуры ------------------------------------
type T_TOOLINFO
 integer(4)  :: cbSize
 integer(4)  :: uFlags
 integer(4)  :: hwnd
 integer(4)  :: uId
 type (T_RECT):: rect
 integer(4)  :: hInst
```

```
 integer(4)  :: lpszText
end type T_TOOLINFO

type    T_TOOLTIPTEXT
 type (T_NMHDR):: hdr
 integer(4)   :: lpszText
 character(80) :: szText
 integer(4)   :: hInst
 integer(4)   :: uFlags
end type  T_TOOLTIPTEXT

!MS$ENDIF  ! NOTOOLTIPS
```

УРОК

15. Окна просмотра деревьев

Окна просмотра деревьев (tree view control) – это окна, в которых отображается иерархический список каких-либо элементов, например файлов и каталогов на диске. Каждый элемент включает метку и необязательное растровое изображение. Любой из них может иметь свой список элементов, которые называют *порожденными* или *дочерними*. Череда дочерних элементов образует ветвь.

Элемент, который имеет один или более порожденных элементов, называется *родительским* или *корневым узлом*. Дочерние элементы отображаются ниже своего родителя и выравниваются так, чтобы была ясна их принадлежность. Элемент без родителя имеет наивысшую иерархию и называется *корнем*.

Окно просмотра деревьев является универсальным и очень мощным элементом управления. Поэтому оно оснащено обширной системой стилей, сообщений и структур. Изучение всех возможностей взаимодействия с окном просмотра деревьев представляет собой самостоятельную задачу, которая не может быть решена в рамках данной книги. Можно надеяться, что вы сможете существенно продвинуться в деле освоения этого элемента управления при создании собственных приложений.

15.1. Создание окна просмотра деревьев

Поскольку окно просмотра деревьев – это окно особого класса, то его можно создать непосредственно, используя функцию **CreateWindowEx()** и задавая класс WC_TREEVIEW. Этот класс будет зарегистрирован, только если загружена DLL. Для этого необходимо использовать функцию **InitCommonControls()**. Этот вопрос мы уже неоднократно обсуждали.

Вы уже знаете, что если повезет, то для создания общего элемента управления удобнее воспользоваться средствами MDS. Для окна просмотра деревьев нам с вами повезло.

Добавим древовидную структуру в диалог **ABOUTBOX**. Для этого откроем диалоговый ресурс IDD_DIALOG1 (диалоговая функция – **DL1**). При необходимости вы можете обратиться к разд. 13.4.

Используя окно редактора ресурсов с управляющими элементами (меню ***Controls***), поместим в рабочую область окна диалога окно просмотра деревьев. Двойным щелчком левой клавиши мыши открываем окно ***Tree Control Properties*** и устанавливаем стиль окна ***Visible***.

Переходим в окно ***Styles*** для того, чтобы установить стили кнопки. Все константы, задающие стиль окна просмотра деревьев, имеют префикс TVS_. В табл. 15.1 приведены стили окна просмотра деревьев.

Таблица 15.1. Стили окна просмотра деревьев

Стиль	*Значение*	*Описание*
TVS_HASBUTTONS	#0001	Ветви имеют кнопки
TVS_HASLINES	#0002	Элементы соединены линиями
TVS_LINESATROOT	#0004	Корневые узлы соединены линиями
TVS_EDITLABELS	#0008	Разрешается редактировать метки
TVS_DISABLEDRAGDROP	#0010	Блокируется перемещение элементов
TVS_SHOWSELALWAYS	#0020	Запоминается сделанный ранее выбор

Окно ***Styles*** содержит несколько полей, значение которых описано ниже.

В поле **Has Buttons** устанавливается стиль TVS_HASBUTTONS. При установке этого стиля слева от изображения каждой ветви добавляется кнопка. Пользователь может использовать кнопку для того, чтобы разворачивать и сворачивать ветвь.

В поле **Has Lines** устанавливается стиль TVS_HASLINES. В окно выводятся линии, которые связывают элементы древовидной структуры.

В поле **Border** устанавливается стиль WS_BORDER, и вокруг элемента управления создается рамка.

В поле **Lines at Root** устанавливается стиль TVS_LINESATROOT. В результате корневые узлы соединяются линиями. Чтобы связать линиями элементы дерева, следует объединить стили TVS_HASLINES и TVS_LINESATROOT.

В поле **Edit Labels** устанавливается стиль TVS_EDITLABELS. Это позволяет редактировать метки элементов дерева.

В поле **Disable Drag Drop** устанавливается стиль TVS_ DISABLEDRAGDROP. Установка этого стиля блокирует перемещение элемента дерева, запрещая генерацию сообщения TVN_BEGINDRAG.

В поле **Show Selection Always** устанавливается стиль TVS_ SHOWSELALWAYS. Этот стиль обеспечивает сохранение сделанного выбора, когда окно просмотра деревьев исчезает с экрана. В результате при повторном открытии окна выбранный ранее элемент будет подсвечен.

Далее следует перейти в окно ***Extend Styles*** и установить дополнительные характеристики окна.

На этом создание описания ресурса заканчивается.

15.2. Взаимодействие с окнами просмотра деревьев

После создания окна просмотра дерева вы можете добавить, удалить, упорядочить элементы, посылая сообщения. Табл. 15.2 содержит описания этих сообщений.

Таблица 15.2. Сообщения окна просмотра деревьев

Сообщение	*Значение*	*Описание*
TV_FIRST	#1100	
TVM_INSERTITEM	TV_FIRST + 1	Вставить элемент; wParam =0, lParam – ссылка на T_TV_INSERTSTRUCT
TVM_DELETEITEM	TV_FIRST + 2	Удаление элемента
TVM_EXPAND	TV_FIRST + 3	Свернуть или развернуть ветвь
TVM_GETITEMRECT	TV_FIRST + 4	Передает координаты прямоугольника, ограничивающего элемент списка
TVM_GETCOUNT	TV_FIRST + 5	Возвращает значение числа элементов
TVM_GETINDENT	TV_FIRST + 6	Возвращает значение отступа
TVM_SETINDENT	TV_FIRST + 7	Устанавливает значение отступа
TVM_GETIMAGELIST	TV_FIRST + 8	Возвращает дескриптор списка изображений
TVM_SETIMAGELIST	TV_FIRST + 9	Связывает список изображений с окном

Сообщение	*Значение*	*Описание*
TVM_GETNEXTITEM	TV_FIRST + 10	Передает информацию о следующем элементе дерева
TVM_SELECTITEM	TV_FIRST + 11	Выбрать элемент
TVM_GETITEM	TV_FIRST + 12	Получить информацию об элементе
TVM_SETITEM	TV_FIRST + 13	Установить параметры элемента
TVM_EDITLABEL	TV_FIRST + 14	Редактировать текст элемента
TVM_GETEDITCONTROL	TV_FIRST + 15	Передает дескриптор для последующего редактирования текста
TVM_GETVISIBLECOUNT	TV_FIRST + 16	Передает число элементов, которые видны в окне
TVM_HITTEST	TV_FIRST + 17	Определяет положение по отношению к окну
TVM_CREATEDRAGIMAGE	TV_FIRST + 18	Создает изображение для перетаскивания
TVM_SORTCHILDREN	TV_FIRST + 19	Сортирует дочерние элементы
TVM_ENSUREVISIBLE	TV_FIRST + 20	Гарантирует, что элемент виден
TVM_SORTCHILDRENCB	TV_FIRST + 21	Процедура приложения сортирует элементы
TVM_ENDEDITLABELNO	TV_FIRST + 22	Закончить процесс редактирования
TVM_GETISEARCHSTRING	TV_FIRST + 23	Находит строку заданного вида
TVM_SETTOOLTIPS	TV_FIRST + 24	Используется для организации окна подсказки
TVM_GETTOOLTIPS	TV_FIRST + 25	Используется для организации окна подсказки

Посылая сообщение TVM_INSERTITEM, вы добавляете новый элемент в дерево. Сообщение возвращает дескриптор этого элемента. Параметр **wParam** равен нулю, а **lParam** – это ссылка на структуру типа T_TV_INSERTSTRUCT, которая определена следующим образом:

```
type T_TV_INSERTSTRUCT
   integer(4) :: hParent
   integer(4) :: hInsertAfter
   type(T_TV_ITEM) item
end type  T_TV_INSERTSTRUCT
```

Поле **hParent** содержит дескриптор родительского элемента. Если добавляемый элемент не имеет родителя, то в это поле следует ввести значение TVI_ROOT(#FFFF0000).

Значение, которое содержится в поле **hInsertAfter**, определяет, каким образом новый элемент должен быть добавлен в дерево. Если это поле содержит дескриптор элемента, то новый элемент будет вставлен после него. Кроме того, поле **hInsertAfter** может принимать значения, которые приведены в табл. 15.3 и задают способ добавления нового элемента.

Таблица 15.3. Значения поля hInsertAfter

Содержимое поля	*Значение*	*Описание*
TVI_FIRST	#FFFF0001	Вставка элемента в начало списка
TVI_LAST	#FFFF0002	Вставка элемента в конец списка
TVI_SORT	#FFFF0003	Вставка элемента с сортировкой в алфавитном порядке

Последнее поле содержит указатель на структуру типа T_TV_ITEM:

```
type T_TV_ITEM
   integer(4) :: mask
   integer(4) :: hItem
   integer(4) :: state
   integer(4) :: stateMask
   integer(4) :: pszText
   integer(4) :: cchTextMax
   integer(4) :: iImage
   integer(4) :: iSelectedImage
   integer(4) :: cChildren
   integer(4) :: lParam
end type  T_TV_ITEM
```

Здесь значение поля **mask** определяет, в каком из полей структуры содержится информация об элементе дерева. Возможные значения флагов приведены в табл. 15.4.

Таблица 15.4. Битовые флаги поля mask

Флаг	*Значение*	*Описание*
TVIF_TEXT	#0001	Информация в полях PszText и cchTextMax
TVIF_IMAGE	#0002	Информация в поле iImage
TVIF_PARAM	#0004	Информация в поле lParam
TVIF_STATE	#0008	Информация в полях state и stateMask
TVIF_HANDLE	#0010	Информация в поле hItem
TVIF_SELECTEDIMAGE	#0020	Информация в поле iSelectedImage
TVIF_CHILDREN	#0040	Информация в поле cChildren

Поле **hItem** содержит дескриптор элемента, к которому относится эта структура.

Поле **state** содержит информацию о состоянии элемента, а поле **stateMask** – какое состояние должно быть установлено или получено. Значения флагов состояния приведены в табл. 15.5.

Таблица 15.5. Битовые флаги внешнего вида и состояния

Флаг	*Значение*	*Описание*
TVIS_SELECTED	#0002	Элемент выбран
TVIS_CUT	#0004	Элемент выбран для копирования
TVIS_DROPHILITED	#0008	Элемент выбран как место назначения для операции перетаскивания
TVIS_BOLD	#0010	Текст написан жирным шрифтом
TVIS_EXPANDED	#0020	Дочерние элементы видны
TVIS_EXPANDEDONCE	#0040	Элемент раскрывается минимум один раз

В поле **pszText** заносится указатель на строку, которая появляется в элементе дерева. Если это поле содержит значение LPSTR_ TEXTCALLBACK, то родительское окно ответственно за формирование текста элемента. В этом случае окно просмотра дерева обменивается с родительским окном сообщениями TVN_GETDISPINFO и TVN_ SETDISPINFO.

В поле **cchTextMax** указывается длина текстового буфера, адрес которого занесен в поле **pszText**.

Поле **iImage** применяется, если с деревом ассоциируется список изображений. В этом случае оно содержит индекс изображения. Если список изображений отсутствует, то **iImage** должен быть равен -1.

Поле **iSelectedImage** содержит индекс изображения выбранного элемента, если такое изображение уже создано. Если в это поле занесено значение I_IMAGECALLBACK, то родительское окно ответственно за формирование изображений. В этом случае окно просмотра дерева посылает родительскому окну сообщения TVN_GETDISPINFO и TVN_ SETDISPINFO.

В поле **cChildren** содержится информация о наличии дочерних элементов. Если такие элементы есть, то значение поля равно 1, иначе оно равно нулю. Значение I_CHILDRENCALLBACK свидетельствует о том, что родительское окно ответственно за прорисовку дочерних элементов.

Поле **lParam** применяется для передачи любой вашей информации в форме 32-битового числа.

В любой момент состояние дерева можно изменить, разворачивая или сворачивая родительские элементы. В развернутом состоянии дочерние элементы отображаются ниже родительского, в свернутом они не отображаются. Если родитель имеет стиль, то состояние переключается автоматически при нажатии кнопки, связанной с родительским элементом. Прикладная программа также может разворачивать и сворачивать элементы, используя сообщение TVM_EXPAND. Параметр **wParam** определяет тип операции. Флаги операций над элементами окна просмотра деревьев приведены в табл. 15.6. Параметр **lParam** содержит дескриптор родительского узла ветви.

Таблица 15.6. Битовые флаги операций с элементами

Флаг	*Значение*	*Описание*
TVE_COLLAPSE	#0001	Ветвь дерева сворачивается
TVE_EXPAND	#0002	Ветвь дерева разворачивается
TVE_TOGGLE	#0003	Переключение состояния на противоположное
TVE_COLLAPSERESET	#8000	Сворачивание с удалением пустых элементов

15.3. Инициализация окна просмотра деревьев и обработка нотификационных сообщений

Приведенная ниже диалоговая процедура демонстрирует использование окна просмотра деревьев.

Инициализации окна просмотра деревьев производится в ответ на стандартное сообщение WM_INITDIALOG. Прежде всего с помощью функции **GetDlgItem(hDlg, IDC_TREE1)** определяем дескриптор, который будет использован для общения с окном просмотра деревьев. Затем добавляются элементы в дерево. Начинаем с корня, затем добавляем "ветви" и, наконец, "листья".

```
!*****************************************************************
integer*4 function DL1 (hDlg, message, wParam, lParam)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_DL1@16' :: DL1

integer hDlg, message, wParam, lParam
integer(4)                      ::ir
integer(4)                      :: i, j
integer(4)                      :: hTreeRoot
integer(4), dimension(4, 4)     :: hTreeWnd

character(16) :: Bfr

type (T_TV_INSERTSTRUCT)  :: Tvs

type (T_NM_TREEVIEW)      :: TView
POINTER(ip, TView)

hDialog = hDlg
lParam = lParam

select case (message)
 case (WM_INITDIALOG)
 !------------ Разметка дерева ---------------------
  hTree  = GetDlgItem(hDlg, IDC_TREE1)

  Tvs%hInsertAfter   =TVI_LAST
  Tvi%mask       =TVIF_TEXT
  Bfr ="Ствол"C
  Tvi%pszText    = LOC(Bfr)
  Tvs%hParent    = TVI_ROOT
```

```
 Tvs%item        = Tvi
 hTreeRoot       = SendMessage(hTree, TVM_INSERTITEM, 0, LOC(Tvs))

 do i=1, 4
  Bfr='Ветви'C
  Tvi%pszText    = LOC(Bfr)
  Tvs%hParent    =hTreeRoot
  Tvs%item       = Tvi
  hTreeWnd(i, 1) = SendMessage(hTree, TVM_INSERTITEM, 0, LOC(Tvs))
  Tvs%hParent    =hTreeWnd(i, 1)
  do j=2, 4
   Bfr='Листья'C
   Tvi%pszText    = LOC(Bfr)
   Tvs%item       = Tvi
   hTreeWnd(i, j) = SendMessage(hTree, TVM_INSERTITEM, 0, LOC(Tvs))
  end do
 end do
 ir= MoveWindow(hDlg, rcDlg%left, rcDlg%top+22, &
           rcDlg%right, rcDlg%bottom -22, .TRUE.)
   DL1 = 1
   return

case (WM_NOTIFY)
 ip = TRANSFER(lParam, ip)
 select case(TView%hdr%code)
  case(TVN_SELCHANGED)
    write (Bfr, '(i10)') TView%itemNew%hItem
    Bfr(10:11)=' 'C
   ir = SetDlgItemText(hDlg, IDC_STATIC3, "Перешли на "//Bfr)
  case(TVN_SELCHANGING)
    write (Bfr, '(i10)') TView%itemOld%hItem
    Bfr(10:11)=' 'C
   ir = SetDlgItemText(hDlg, IDC_STATIC2, "Покидаем "//Bfr)
  case(NM_DBLCLK)
    ir = MessageBox(ghwndMain, "Выбор сделан"C, "Сообщение"C, MB_OK)
 end select
case (WM_COMMAND)
     if (wParam == IDOK) then
      end if
end select

DL1 = 0
end function DL1
!*****************************************************************
```

Окно просмотра деревьев посылает родительскому окну (в данном примере окно диалога IDD_DIALOG1) нотификационные сообщения в формате WM_NOTIFY. Для обслуживания окна просмотра деревьев в полном объеме предусмотрено 12 типов кодов (и еще столько же, если используется UNICOD). Кроме простейших операций можно также производить более сложные, например перетаскивание элемента, содержащего рисунок. Значения кодов приведены в табл. 15.7. Все они имеют префикс TVN_.

Таблица 15.7. Нотификационные сообщения окна просмотра деревьев

Нотификационный код	*Значение*	*Описание*
TVN_FIRST	-400	
TVN_SELCHANGING	TVN_FIRST-1	Элемент должен быть выбран
TVN_SELCHANGED	TVN_FIRST-2	Элемент уже выбран
TVN_GETDISPINFO	TVN_FIRST-3	Получить информацию для отображения элемента
TVN_SETDISPINFO	TVN_FIRST-4	Обновить информацию для отображения элемента
TVN_ITEMEXPANDING	TVN_FIRST-5	Элемент должен быть развернут или свернут
TVN_ITEMEXPANDED	TVN_FIRST-6	Элемент уже развернут или свернут
TVN_BEGINDRAG	TVN_FIRST-7	Начать процедуру перетаскивания изображения левой клавишей мыши
TVN_BEGINRDRAG	TVN_FIRST-8	Начать процедуру перетаскивания изображения правой клавишей мыши
TVN_DELETEITEM	TVN_FIRST-9	Элемент удален
TVN_BEGINLABELEDIT	TVN_FIRST-10	Начать редактирование текста
TVN_ENDLABELEDIT	TVN_FIRST-11	Закончить редактирование
TVN_KEYDOWN	TVN_FIRST-12	
TVN_LAST	-499	

Наиболее часто используются сообщения TVN_SELCHANGING и TVN_SELCHANGED. В диалоговой процедуре, текст которой приведен выше, в ответ на первое сообщение на экран выводится значение идентификатора старого элемента, а на второе – нового. Параметр **lParam** содержит указатель структуры типа T_NM_TREEVIEW:

```
type T_NM_TREEVIEW
 type(T_NMHDR)   :: hdr
 integer(4)      :: action
 type(T_TV_ITEM) :: itemOld
 type(T_TV_ITEM) :: itemNew
 type(T_POINT)   :: ptDrag
end type T_NM_TREEVIEW.
```

Первое поле этой структуры нам уже знакомо. Второе – это код операции. В третьем и четвертом полях находится информация о старом и новом элементах. Последнее поле содержит координаты курсора.

По мере надобности вы самостоятельно сможете освоить все возможности нотификационных сообщений.

Так, например, сообщения TVN_ITEMEXPANDING и TVN_ITEMEXPANDED дают возможность управлять свойствами элемента в процессе сворачивания или разворачивания ветви, а сообщение TVN_GETDISPINFO передает информацию об элементе через структуру типа T_TV_DISPINFO.

Кроме специализированных существует некоторое число кодов общего назначения.

Таблица 15.8. Нотификационные коды общего назначения

Код	*Значение*	*Описание*
NM_FIRST	0	
NM_OUTOFMEMORY	NM_FIRST-1	Операция не завершена из-за недостаточности объема памяти
NM_CLICK	NM_FIRST-2	Пользователь нажал левую кнопку мыши
NM_DBLCLK	NM_FIRST-3	То же, но дважды
NM_RETURN	NM_FIRST-4	Элемент управления активизирован и нажата клавиша ENTER
NM_RCLICK	NM_FIRST-5	Пользователь нажал правую клавишу мыши
NM_RDBLCLK	NM_FIRST-6	То же, но дважды
NM_SETFOCUS	NM_FIRST-7	Элемент управления активизирован
NM_KILLFOCUS	NM_FIRST-8	Элемент управления потерял активность
NM_LAST	-99	

В нашем диалоге используется код NM_DBLCLK для того, чтобы зафиксировать выбор элемента дерева.

Если включить в текст нашего приложения приведенный здесь фрагмент, то при запуске программы можно получить на экране изображение, показанное на рис. 15.1. Однако следует помнить, что, по всей вероятности, на вашем диске нет файла, эквивалентного **commctrl.h**, и поэтому надо внести дополнения в модуль **commctrlty**.

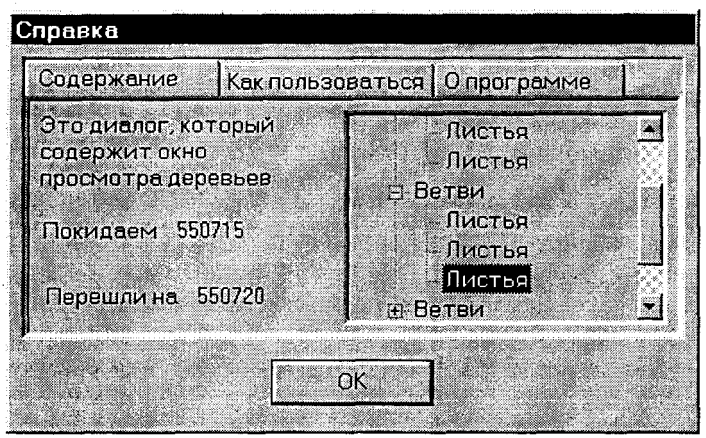

Рис. 15.1. Диалог с окном просмотра деревьев

Дополнения, которые следует внести в файл **commctrlty.f90**, выглядят следующим образом:

```
!======TREE VIEW CONTROL ============================

!MS$IF .NOT. DEFINED (NOTREEVIEW)

!--------------- имя класса ---------------------------------
 character(15), parameter, public :: WC_TREEVIEW   = "SysTreeView32"C

!--------------- costants -----------------------------------

integer, parameter, public :: TVN_FIRST    = -400   ! treeview
integer, parameter, public :: TVN_LAST     = -499

!--------------- стили окна ---------------------------
integer, parameter, public :: TVS_HASBUTTONS              = #0001
integer, parameter, public :: TVS_HASLINES                = #0002
integer, parameter, public :: TVS_LINESATROOT             = #0004
integer, parameter, public :: TVS_EDITLABELS              = #0008
```

```
integer, parameter, public :: TVS_DISABLEDRAGDROP  = #0010
integer, parameter, public :: TVS_SHOWSELALWAYS    = #0020
integer, parameter, public :: TVS_SHAREDIMAGELISTS = #0000
integer, parameter, public :: TVS_PRIVATEIMAGELISTS = #0040
integer, parameter, public :: TVS_NOTOOLTIPS       = #0080
integer, parameter, public :: TVS_CHECKBOXES       = #0100
integer, parameter, public :: TVS_TRACKSELECT      = #0200

!---------- Флаги структуры T_TV_ITEM----------
integer, parameter, public :: TVIF_TEXT            = #0001
integer, parameter, public :: TVIF_IMAGE           = #0002
integer, parameter, public :: TVIF_PARAM           = #0004
integer, parameter, public :: TVIF_STATE           = #0008
integer, parameter, public :: TVIF_HANDLE          = #0010
integer, parameter, public :: TVIF_SELECTEDIMAGE   = #0020
integer, parameter, public :: TVIF_CHILDREN        = #0040
integer, parameter, public :: TVIF_DI_SETITEM      = #1000

!---------- Флаги внешнего вида ----------
integer, parameter, public :: TVIS_SELECTED        = #0002
integer, parameter, public :: TVIS_CUT             = #0004
integer, parameter, public :: TVIS_DROPHILITED     = #0008
integer, parameter, public :: TVIS_BOLD            = #0010
integer, parameter, public :: TVIS_EXPANDED        = #0020
integer, parameter, public :: TVIS_EXPANDEDONCE    = #0040
integer, parameter, public :: TVIS_EXPANDPARTIAL   = #0080
integer, parameter, public :: TVIS_OVERLAYMASK     = #0F00
integer, parameter, public :: TVIS_STATEIMAGEMASK  = #F000
integer, parameter, public :: TVIS_USERMASK        = #F000

!--------- Сообщения ----------
integer, parameter, public :: TVM_INSERTITEM       = TV_FIRST + 0
integer, parameter, public :: TVM_DELETEITEM       = TV_FIRST + 1
integer, parameter, public :: TVM_EXPAND           = TV_FIRST + 2
integer, parameter, public :: TVM_GETITEMRECT      = TV_FIRST + 4
integer, parameter, public :: TVM_GETCOUNT         = TV_FIRST + 5
integer, parameter, public :: TVM_GETINDENT        = TV_FIRST + 6
integer, parameter, public :: TVM_SETINDENT        = TV_FIRST + 7
integer, parameter, public :: TVM_GETIMAGELIST     = TV_FIRST + 8
integer, parameter, public :: TVM_SETIMAGELIST     = TV_FIRST + 9
integer, parameter, public :: TVM_GETNEXTITEM      = TV_FIRST + 10
integer, parameter, public :: TVM_SELECTITEM       = TV_FIRST + 11
integer, parameter, public :: TVM_GETITEM          = TV_FIRST + 12
integer, parameter, public :: TVM_SETITEM          = TV_FIRST + 13
integer, parameter, public :: TVM_EDITLABEL        = TV_FIRST + 14
```

```
integer, parameter, public :: TVM_GETEDITCONTROL    = TV_FIRST + 15
integer, parameter, public :: TVM_GETVISIBLECOUNT   = TV_FIRST + 16
integer, parameter, public :: TVM_HITTEST           = TV_FIRST + 17
integer, parameter, public :: TVM_CREATEDRAGIMAGE   = TV_FIRST + 18
integer, parameter, public :: TVM_SORTCHILDREN      = TV_FIRST + 19
integer, parameter, public :: TVM_ENSUREVISIBLE     = TV_FIRST + 20
integer, parameter, public :: TVM_SORTCHILDRENCB    = TV_FIRST + 21
integer, parameter, public :: TVM_GETISEARCHSTRING  = TV_FIRST + 23
integer, parameter, public :: TVM_ENDEDITLABELNO    = TV_FIRST + 22
integer, parameter, public :: TVM_SETTOOLTIPS       = TV_FIRST + 24
integer, parameter, public :: TVM_GETTOOLTIPS       = TV_FIRST + 25

!---------- Действия над элементом окна ----------
integer, parameter, public :: TVE_COLLAPSE          = #0001
integer, parameter, public :: TVE_EXPAND            = #0002
integer, parameter, public :: TVE_TOGGLE            = #0003
integer, parameter, public :: TVE_EXPANDPARTIAL     = #4000
integer, parameter, public :: TVE_COLLAPSERESET     = #8000

!---------- Нотификационные сообщения ----------
integer, parameter, public :: TVN_SELCHANGING       = TVN_FIRST-1
integer, parameter, public :: TVN_SELCHANGED        = TVN_FIRST-2
integer, parameter, public :: TVN_GETDISPINFO       = TVN_FIRST-3
integer, parameter, public :: TVN_SETDISPINFO       = TVN_FIRST-4

integer, parameter, public :: I_CHILDRENCALLBACK    = -1

integer, parameter, public :: TVSIL_NORMAL          = 0
integer, parameter, public :: TVSIL_STATE           = 2

integer, parameter, public :: TVC_UNKNOWN           = #0000
integer, parameter, public :: TVC_BYMOUSE           = #0001
integer, parameter, public :: TVC_BYKEYBOARD        = #0002

integer, parameter, public :: TVI_ROOT              = #FFFF0000
integer, parameter, public :: TVI_FIRST             = #FFFF0001
integer, parameter, public :: TVI_LAST              = #FFFF0002
integer, parameter, public :: TVI_SORT              = #FFFF0003

integer, parameter, public :: TVGN_ROOT             = #0000
integer, parameter, public :: TVGN_NEXT             = #0001
integer, parameter, public :: TVGN_PREVIOUS         = #0002
integer, parameter, public :: TVGN_PARENT           = #0003
integer, parameter, public :: TVGN_CHILD            = #0004
integer, parameter, public :: TVGN_FIRSTVISIBLE     = #0005
integer, parameter, public :: TVGN_NEXTVISIBLE      = #0006
```

```
integer, parameter, public :: TVGN_PREVIOUSVISIBLE  = #0007
integer, parameter, public :: TVGN_DROPHILITE       = #0008
integer, parameter, public :: TVGN_CARET            = #0009

integer, parameter, public :: TVHT_NOWHERE          = #0001
integer, parameter, public :: TVHT_ONITEMICON       = #0002
integer, parameter, public :: TVHT_ONITEMLABEL      = #0004

integer, parameter, public :: TVHT_ONITEM           = #0046
integer, parameter, public :: TVHT_ONITEMINDENT     = #0008
integer, parameter, public :: TVHT_ONITEMBUTTON     = #0010
integer, parameter, public :: TVHT_ONITEMRIGHT      = #0020
integer, parameter, public :: TVHT_ONITEMSTATEICON  = #0040

integer, parameter, public :: TVHT_ABOVE            = #0100
integer, parameter, public :: TVHT_BELOW            = #0200
integer, parameter, public :: TVHT_TORIGHT          = #0400
integer, parameter, public :: TVHT_TOLEFT           = #0800

!------------ Типы ---------------------------------------------
type T_TV_ITEM
   integer(4) :: mask
   integer(4) :: hItem
   integer(4) :: state
   integer(4) :: stateMask
   integer(4) :: pszText
   integer(4) :: cchTextMax
   integer(4) :: iImage
   integer(4) :: iSelectedImage
   integer(4) :: cChildren
   integer(4) :: lParam
end type  T_TV_ITEM

type T_TV_INSERTSTRUCT
   integer(4) :: hParent
   integer(4) :: hInsertAfter
   type(T_TV_ITEM) item
end type  T_TV_INSERTSTRUCT

type T_TV_HITTESTINFO
 integer(4) :: pt
 integer(4) :: flags
 integer(4) :: hItem
end type T_TV_HITTESTINFO

type T_TV_SORTCB
```

```
 integer(4) :: hParent
 integer(4) :: lpfnCompare
 integer(4) :: lParam
end type T_TV_SORTCB

type T_NM_TREEVIEW
 type(T_NMHDR)   :: hdr
 integer(4)      :: action
 type(T_TV_ITEM) :: itemOld
 type(T_TV_ITEM) :: itemNew
 type(T_POINT)   :: ptDrag
end type T_NM_TREEVIEW

type T_TV_DISPINFO
 type(T_NMHDR)   ::hdr
 type(T_TV_ITEM) :: item
end type T_TV_DISPINFO

type T_TV_KEYDOWN
 type(T_NMHDR)   :: hdr
 integer(4)      :: wVKey
 integer(4)      :: flags
end type T_TV_KEYDOWN

!MS$ENDIF ! NOTREEVIEW
```

Этот фрагмент содержит практически все необходимые константы для обслуживания окна просмотра деревьев. Если у вас возникнут какие-то проблемы, то следует обратиться к справочной системе и файлу **commctrl.h** в каталоге **Msdev\Include**. При определенной настойчивости все проблемы могут быть разрешены.

УРОК

16. Ползунковый регулятор

Ползунковый регулятор (trackbar) – это общий элемент управления в виде линейного регулятора, который часто используется в радиоэлектронной аппаратуре. Будем в дальнейшем употреблять также термин *регулятор*. Когда пользователь перемещает движок регулятора, применяя мышь или клавиши, ползунок посылает нотификационные сообщения, которые указывают его позицию.

Ползунковый регулятор в приложении полезен в тех же случаях, что и в реальной аппаратуре. Например, вы можете с его помощью оперативно изменять скорость выполнения какой-то процедуры.

16.1. Создание ползункового регулятора

Поскольку регулятор – это просто особого вида окно, то для его создания используются функции **CreateWindow()** или **CreateWindowEx()**. Имя класса ползункового регулятора – TRACKBAR_CLASS. Этот класс будет зарегистрирован, только если загружена DLL. Для этого необходимо использовать функцию **InitCommonControls()**.

Регулятор может иметь как вертикальную, так и горизонтальную ориентацию. Шкала располагается над движком, под ним или с обеих сторон. Свойства регулятора устанавливаются при его создании путем задания стиля окна. Все специализированные стили регулятора имеют префикс TBS_. В табл. 16.1 приведены значения и описания специфических стилей окна ползункового регулятора.

Таблица 16.1. Стили окна ползункового регулятора

Стиль	*Значение*	*Описание*
TBS_HORZ	#0000	Регулятор отображается горизонтально
TBS_BOTTOM	#0000	Шкала регулятора располагается под ползунком
TBS_RIGHT	#0000	Шкала регулятора располагается справа от ползунка
TBS_AUTOTICKS	#0001	Шкала отображается автоматически

Стиль	*Значение*	*Описание*
TBS_VERT	#0002	Регулятор отображается вертикально
TBS_TOP	#0004	Шкала регулятора располагается над ползунком
TBS_LEFT	#0004	Шкала регулятора располагается слева
TBS_BOTH	#0008	Шкала регулятора располагается с двух сторон
TBS_NOTICKS	#0010	Регулятор без шкалы
TBS_ENABLESELRANGE	#0020	Регулятор имеет широкую прорезь, выбранные пределы отмечаются треугольниками
TBS_FIXEDLENGTH	#0040	Фиксированная длина прорези
TBS_NOTHUMB	#0080	Регулятор не имеет прорези
TBS_TOOLTIPS	#0100	Регулятор оснащен подсказками

Стили TBS_HORZ и TBS_VERT определяют ориентацию регулятора. По умолчания ориентация горизонтальная.

Если стиль TBS_AUTOTICKS не установлен, то шкала должна размечаться программно. Для этого придется использовать сообщения типа TBM_SETTIC и TBM_SETTICFREQ.

Положение меток шкалы для горизонтального регулятора определяются стилями TBS_BOTTOM и TBS_TOP, для вертикального – TBS_RIGHT и TBS_LEFT (TBS_BOTTOM и TBS_RIGHT – заданы по умолчанию).

Если установлен стиль TBS_ENABLESELRANGE, то можно оперативно менять диапазон изменения, и новые пределы будут отмечены треугольными метками на прорези регулятора. По умолчанию длина прорези регулятора изменяется при изменении границ выбора. Если же установлен стиль TBS_FIXEDLENGTH, то она остается неизменной.

Для создания ползункового регулятора, конечно же, удобнее использовать средства MDS.

Дополним диалог **ЭЛЕМЕНТЫ УПРАВЛЕНИЯ** ползунковым регулятором. Для этого откроем ресурс диалога и, используя окно редактора ресурсов с управляющими элементами (меню ***Controls***), поместим в рабочую область окна диалога окно регулятора. Двойным щелчком левой клавиши мыши открываем окно ***Slider Properties*** и устанавливаем стиль окна ***Visible***. Затем переходим в окно ***Styles*** для того, чтобы установить стили.

В поле **Orientation** окна ***Styles*** устанавливаем ориентацию регулятора (***Horizontal***). Расположение шкалы задается в поле **Point**. Выберем, напри-

мер ***Bottom/Right***. Для того чтобы шкала имела разметку, установим флаг в поле **Tick marks**. Определим стиль окна регулятора в виде комбинации TBS_AUTOTICKS, TBS_ENABLESELRANGE и WS_BORDER, устанавливая флаги в соответствующих полях окна ***Styles***.

В окне ***Extended Styles*** устанавливаются дополнительные стили, которые определяют внешний вид регулятора. Можно, например, придать окну регулятора объемный вид, установив флаг в поле **Client edge**, – стиль WS_EX_CLIENTEDGE.

На этом формирование описания ресурса можно закончить.

16.2. Взаимодействие с ползунковым регулятором

Прикладная программа может обмениваться с регулятором сообщениями. Список сообщений чрезвычайно обширен, и вы имеете возможность задавать или изменять большинство свойств регулятора.

Сообщения регулятору, как и другим общим элементам управления, посылаются с помощью функций **SendMessage()** и **SendDlgItemMessage()**. Все сообщения имеют префикс TBM_. Описание сообщений приведено в табл. 16.2.

Таблица 16.2. Сообщения ползункового регулятора

Сообщение	*Значение*	*Описание*
TBM_GETPOS	WM_USER	Возвращает текущую позицию движка
TBM_GETRANGEMIN	WM_USER+1	Возвращает нижнюю границу диапазона регулятора
TBM_GETRANGEMAX	WM_USER+2	Возвращает верхнюю границу диапазона регулятора
TBM_GETTIC	WM_USER+3	Возвращает позицию метки wParam
TBM_SETTIC	WM_USER+4	Устанавливает позицию метки wParam
TBM_SETPOS	WM_USER+5	Устанавливает движок в позицию lParam
TBM_SETRANGE	WM_USER+6	Устанавливает диапазон регулировки

Сообщение	*Значение*	*Описание*
TBM_SETRANGEMIN	WM_USER+7	Устанавливает нижнюю границу диапазона регулятора
TBM_SETRANGEMAX	WM_USER+8	Устанавливает верхнюю границу диапазона регулятора
TBM_CLEARTICS	WM_USER+9	Удаляет текущую метку
TBM_SETSEL	WM_USER+10	Устанавливает точки выделения в диапазоне регулятора
TBM_SETSELSTART	WM_USER+11	Устанавливает начальную точку выделения
TBM_SETSELEND	WM_USER+12	Устанавливает конечную точку выделения
TBM_GETPTICS	WM_USER+14	Возвращает указатель на массив с позициями меток
TBM_GETTICPOS	WM_USER+15	Возвращает позицию метки wParam в оконных координатах
TBM_GETNUMTICS	WM_USER+16	Возвращает число меток
TBM_GETSELSTART	WM_USER+17	Возвращает начальную точку выделения
TBM_GETSELEND	WM_USER+18	Возвращает конечную точку выделения
TBM_CLEARSEL	WM_USER+19	Сбрасывает выделение
TBM_SETTICFREQ	WM_USER+20	Устанавливает частоту следования меток
TBM_SETPAGESIZE	WM_USER+21	Устанавливает шаг движка
TBM_GETPAGESIZE	WM_USER+22	Возвращает величину шага движка
TBM_SETLINESIZE	WM_USER+23	Устанавливает величину шага для сообщений TB_LINEUP и TB_LINEDOWN
TBM_GETLINESIZE	WM_USER+24	Возвращает величину шага сообщений TB_LINEUP и TB_LINEDOWN
TBM_GETTHUMBRECT	WM_USER+25	Возвращает габариты окна регулятора
TBM_GETCHANNELRECT	WM_USER+26	Возвращает габариты прорези регулятора

Сообщение	*Значение*	*Описание*
TBM_SETTHUMBLENGTH	WM_USER+27	Устанавливает длину окна регулятора
TBM_GETTHUMBLENGTH	WM_USER+28	Возвращает длину окна регулятора
TBM_SETTOOLTIPS	WM_USER+29	Используется с подсказками, устанавливает дескриптор
TBM_GETTOOLTIPS	WM_USER+30	Используется с подсказками, передает дескриптор
TBM_SETTIPSIDE	WM_USER+31	Устанавливает длину меток шкалы
TBM_SETBUDDY	WM_USER+32	Определяет приятельское окно
TBM_GETBUDDY	WM_USER+33	Передает дескриптор приятельского окна

Сразу же после создания окна регулятора удобно задать диапазон его непрерывных значений. В приведенном ниже примере это делается на стадии инициализации диалога. Для этого используется сообщение TBM_SETRANGE. Параметр **lParam** задает границы диапазона, он создается с помощью процедуры **MAKELONG(1, 10)**. Младшее слово содержит верхнюю границу, а старшее – нижнюю. Для того чтобы установить новые значения пределов, **wParam** должно быть отлично от нуля. Пределы можно установить и раздельно с помощью сообщений TBM_SETRANGEMAX и TBM_SETRANGEMIN.

Прикладная программа, в которой диапазон изменяется динамически, может в любой момент определить границы диапазона, используя сообщения TBM_GETRANGEMAX и TBM_GETRANGEMIN.

Для определения и установки позиции движка служат сообщения TBM_GETPOS и TBM_SETPOS. В последнем случае новая позиция указывается в **lParam**.

Для оперативного управления позицией и частотой меток шкалы предусмотрены сообщения TBM_SETTIC и TBM_SETTICFREQ. Последнее сообщение позволяет при необходимости прореживать шкалу регулятора. Так, например, можно для шкалы, имевшей изначально 100 делений, отображать только каждое десятое. Для этого параметр **wParam** должен быть равен 10. Сообщение TBM_GETTIC передает положение метки шкалы.

Дополнительные удобства для пользователя предоставляет стиль TBS_ENABLESELRANGE. Используя сообщения TBM_SETSEL, TBM_SETSELSTART и TBM_SETSELEND, можно установить начальную и конечную метку на шкале регулятора. Сообщения TBM_GETSELSTART и TBM_GETSELEND передают положения меток. Чтобы убрать метки, используется сообщение TBM_CLEARSEL.

Кроме специализированных сообщений регулятор передает ряд стандартных сообщений. Наиболее важное из них – WM_HSCROL. Слово младшего разряда параметра **wParam** содержит нотификационный код сообщения, а старшее слово определяет позицию сдвига.

16.3. Пример диалога с ползунковым регулятором

Текст процедуры диалога с ползунковым регулятором приведен ниже.

```
!*****************************************************************
integer*4 function DialogFunc5(hDlg, message, wParam, lParam)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_DialogFunc5@16' :: DialogFunc5

integer(4)  hDlg, message, wParam, lParam
integer(4)  hTBwnd,
integer(4)  iPos

hDialog = hDlg

select case(message)
 case (WM_INITDIALOG)
!--------- инициализация диалога ----------------
  hTBWnd  = GetDlgItem(hDlg, IDC_SLIDER1)

  ir = SendDlgItemMessage(hDlg, IDC_SLIDER1, &
                  TBM_SETRANGE, 1, MAKELONG(1, 10))
  iPos=1
  ir = SendDlgItemMessage(hDlg, IDC_SLIDER1, TBM_SETPOS, 1, iPos)
  ir=SetDlgItemInt(hDlg, IDC_STATIC2, iPos, .FALSE.)

  DialogFunc5 = 1
  return

case(WM_HSCROLL)
 if (lParam==hTBWnd) then
  ir = SendDlgItemMessage(hDlg, IDC_SLIDER1, TBM_GETPOS, 0, 0)
  iPos = LOWORD(ir)
```

```
  ir=SetDlgItemInt(hDlg, IDC_STATIC2, iPos, .FALSE.)
  select case(LOWORD(wParam))
   case(TB_TOP)
   case(TB_BOTTOM)
   case(TB_ENDTRACK)
   case(TB_LINEDOWN)
   case(TB_LINEUP)
   case(TB_PAGEUP)
   case(TB_THUMBPOSITION)
   case(TB_PAGEDOWN)
  end if
case (WM_COMMAND)
!---------- обработка команд ------------------------
  select case (MakeLong(LOWORD(wParam), 0))
  end select
 case (WM_SYSCOMMAND)
!--------- обработка сообщений системного меню ----------
  if (wParam == SC_CLOSE) then
    ir = EndDialog (hDlg, 1)
  end if
end select
DialogFunc5=0
end function DialogFunc5
!*****************************************************************
```

При инициализации диалога прежде всего определяется дескриптор окна элемента управления. Значение дескриптора затем используется для взаимодействия с регулятором. С помощью сообщений TBM_SETRANGE и TBM_SETPOS устанавливаются границы шкалы и начальное положение движка. Для удобства при создании описания ресурса в диалог было добавлено статическое окно, в которое заносится текущее значение позиции движка с помощью функции **SetDlgItemInt(hDlg, IDC_STATIC2, iPos, .FALSE.)**.

Текущую позицию движка получаем в ответ на сообщение WM_HSCROL, используя функцию **SendDlgItemMessage (hDlg, IDC_SLIDER1, TBM_GETPOS, 0, 0)**. Полученное значение заносим в статическое окно. В простейшем случае нотификационный код можно не использовать. Однако обработка сообщения станет более разнообразной, если принять во внимание значение младшего слова параметра **wParam**. Значения и описания нотификационных кодов даны в табл. 16.3.

Таблица 16.3. Нотификационные коды сообщения WM_HSCROL

Сообщение	*Значение*	*Описание*
TB_LINEUP	#0000	Нажата левая стрелка или стрелка "вверх"
TB_LINEDOWN	#0001	Нажата правая стрелка или стрелка "вниз"
TB_PAGEUP	#0002	Нажата клавиша PAGE UP или щелчок мыши перед движком
TB_PAGEDOWN	#0003	Нажата клавиша PAGE DOWN или щелчок мыши после движка
TB_THUMBPOSITION	#0004	Движок зафиксирован
TB_THUMBTRACK	#0005	Движок продвигается мышью
TB_TOP	#0006	Нажата клавиша HOME, движок устанавливается в начало
TB_BOTTOM	#0007	Нажата клавиша END, движок устанавливается в конец
TB_ENDTRACK	#0008	Движок зафиксирован после продвижения клавиатурой

Для того чтобы можно было воспользоваться всем арсеналом средств создания и общения с ползунковым регулятором, необходимо дополнить файл **commctrlty.f90** следующим фрагментом:

```
!====== TRACKBAR CONTROL ================================
!MS$IF .NOT. DEFINED (NOTRACKBAR)

!---------- имя класса ----------------------------------
 character(18), parameter, public ::TRACKBAR_CLASS="msctls_trackbar32"C

!---------- константы ----------------------------------

integer, parameter, public :: TBS_AUTOTICKS          = #0001
integer, parameter, public :: TBS_VERT               = #0002
integer, parameter, public :: TBS_HORZ               = #0000
integer, parameter, public :: TBS_TOP                = #0004
integer, parameter, public :: TBS_BOTTOM             = #0000
integer, parameter, public :: TBS_LEFT               = #0004
integer, parameter, public :: TBS_RIGHT              = #0000
integer, parameter, public :: TBS_BOTH               = #0008
integer, parameter, public :: TBS_NOTICKS            = #0010
integer, parameter, public :: TBS_ENABLESELRANGE     = #0020
integer, parameter, public :: TBS_FIXEDLENGTH        = #0040
```

```
integer, parameter, public :: TBS_NOTHUMB          = #0080
integer, parameter, public :: TBS_TOOLTIPS         = #0100

!--------- Сообщения ----------------------------------------
integer, parameter, public :: TBM_GETPOS           = (WM_USER)
integer, parameter, public :: TBM_GETRANGEMIN      = (WM_USER+1)
integer, parameter, public :: TBM_GETRANGEMAX      = (WM_USER+2)
integer, parameter, public :: TBM_GETTIC           = (WM_USER+3)
integer, parameter, public :: TBM_SETTIC           = (WM_USER+4)
integer, parameter, public :: TBM_SETPOS           = (WM_USER+5)
integer, parameter, public :: TBM_SETRANGE         = (WM_USER+6)
integer, parameter, public :: TBM_SETRANGEMIN      = (WM_USER+7)
integer, parameter, public :: TBM_SETRANGEMAX      = (WM_USER+8)
integer, parameter, public :: TBM_CLEARTICS        = (WM_USER+9)
integer, parameter, public :: TBM_SETSEL           = (WM_USER+10)
integer, parameter, public :: TBM_SETSELSTART      = (WM_USER+11)
integer, parameter, public :: TBM_SETSELEND        = (WM_USER+12)
integer, parameter, public :: TBM_GETPTICS         = (WM_USER+14)
integer, parameter, public :: TBM_GETTICPOS        = (WM_USER+15)
integer, parameter, public :: TBM_GETNUMTICS       = (WM_USER+16)
integer, parameter, public :: TBM_GETSELSTART      = (WM_USER+17)
integer, parameter, public :: TBM_GETSELEND        = (WM_USER+18)
integer, parameter, public :: TBM_CLEARSEL         = (WM_USER+19)
integer, parameter, public :: TBM_SETTICFREQ       = (WM_USER+20)
integer, parameter, public :: TBM_SETPAGESIZE      = (WM_USER+21)
integer, parameter, public :: TBM_GETPAGESIZE      = (WM_USER+22)
integer, parameter, public :: TBM_SETLINESIZE      = (WM_USER+23)
integer, parameter, public :: TBM_GETLINESIZE      = (WM_USER+24)
integer, parameter, public :: TBM_GETTHUMBRECT     = (WM_USER+25)
integer, parameter, public :: TBM_GETCHANNELRECT   = (WM_USER+26)
integer, parameter, public :: TBM_SETTHUMBLENGTH   = (WM_USER+27)
integer, parameter, public :: TBM_GETTHUMBLENGTH   = (WM_USER+28)
integer, parameter, public :: TBM_SETTOOLTIPS      = (WM_USER+29)
integer, parameter, public :: TBM_GETTOOLTIPS      = (WM_USER+30)
integer, parameter, public :: TBM_SETTIPSIDE       = (WM_USER+31)
integer, parameter, public :: TBM_SETBUDDY         = (WM_USER+32)
integer, parameter, public :: TBM_GETBUDDY         = (WM_USER+33)

! ---------- Флаги подсказок ----------------------
integer, parameter, public :: TBTS_TOP     = 0
integer, parameter, public :: TBTS_LEFT    = 1
```

```
integer, parameter, public :: TBTS_BOTTOM  = 2
integer, parameter, public :: TBTS_RIGHT   = 3

!-------------------- Нотификационные сообщения ----------------
integer, parameter, public :: TB_LINEUP            =0
integer, parameter, public :: TB_LINEDOWN          =1
integer, parameter, public :: TB_PAGEUP            =2
integer, parameter, public :: TB_PAGEDOWN          =3
integer, parameter, public :: TB_THUMBPOSITION     =4
integer, parameter, public :: TB_THUMBTRACK        =5
integer, parameter, public :: TB_TOP               =6
integer, parameter, public :: TB_BOTTOM            =7
integer, parameter, public :: TB_ENDTRACK          =8

!---- флаги прорисовок пользователя ---------------------
integer, parameter, public :: TBCD_TICS            =#0001
integer, parameter, public :: TBCD_THUMB           =#0002
integer, parameter, public :: TBCD_CHANNEL         =#0003

!MS$ENDIF ! (NOtrackbar)
```

Если теперь создать исполнительный файл, запустить его и выбрать в меню позиции ***Диалоги – Элементы управления***, то на экране возникнет окно диалога, показанное на рис. 16.1.

Рис. 16.1. Диалог с ползунковым регулятором

В нашем примере движок регулятора изменяет позицию в ответ на любые разрешенные манипуляции с клавиатурой и мышью. После чего новое значение позиции выводится на экран.

Вы можете усложнить обработку сообщения WM_HSCROL, добавляя вывод текстового сообщения, связанного с каждым из нотификационных кодов. Текст удобно вывести в новое статическое окно с помощью функции **SetDlgItemText (hDlg, IDDlgItem, lpString)**. Здесь **hDlg** – дескриптор окна диалога, **IDDlgItem** – идентификатор элемента контроля (в данном случае, например, IDC_STATIC3) и **lpString** – строка текста, оканчивающаяся нулем.

Попробуйте, используя подходящие сообщения, изменить текст диалоговой процедуры так, чтобы можно было оперативно изменять свойства ползункового регулятора.

УРОК

17. Индикатор

Индикатор (progress bar) – это небольшое окно, в котором отображается степень завершенности некоторого процесса. По внешнему виду это, пожалуй, самый простой элемент управления. Он состоит из прямоугольника, который по мере выполнения какой-либо операции постепенно заполняется слева направо цветными секторами. Индикаторы часто используют, например, в инсталляционных программах, а также в программах сортировки и копирования, перемещения или передачи информации.

17.1. Создание индикатора и взаимодействие с ним

Окно индикатора можно создать непосредственно в приложении, используя функцию **CreateWindow()** или **CreateWindowEx()**. Класс окна индикатора задается в виде PROGRESS_CLASS. Этот класс будет зарегистрирован, если загружена DLL. Для этого необходимо использовать функцию **InitCommonControls()**.

Если вы создаете индикатор средствами MDS, то для этого прежде всего необходимо открыть шаблон выбранного диалога. Добавим индикатор в диалог **ЭЛЕМЕНТЫ УПРАВЛЕНИЯ**.

Откроем диалоговый ресурс и, используя окно редактора ресурсов с управляющими элементами (меню ***Controls***), поместим в рабочую область окна диалога окно индикатора. Установим его размеры и положение. Двойным щелчком левой клавиши мыши открываем окно ***Progress Properties*** и устанавливаем стиль окна ***Visible***.

Для индикатора в редакторе ресурсов MDS не предусмотрена установка каких-либо стилей. Поэтому остается только в окне ***Extended Styles*** установить при желании дополнительные стили, которые определяют внешний вид регулятора. На этом создание шаблона ресурса заканчивается.

Особенность индикатора как общего элемента управления состоит в том, что его функционирование полностью обеспечивается приложением. Установка в заданную позицию и пошаговое продвижение осуществляются посредством передачи сообщений.

Таблица 17.1. Сообщения, посылаемые индикатору

Сообщение	*Значение*	*Описание*
PBM_SETRANGE	WM_USER+1	Установить диапазон значений индикатора
PBM_SETPOS	WM_USER+2	Установить позицию индикатора
PBM_DELTAPOS	WM_USER+3	Продвинуться на заданное расстояние от текущего значения
PBM_SETSTEP	WM_USER+4	Установить шаг
PBM_STEPIT	WM_USER+5	Сделать шаг заданной величины
PBM_SETRANGE32	WM_USER+6	Расширить возможные значения диапазона до #80000000
PBM_GETRANGE	WM_USER+7	Получить границы значений индикатора
PBM_GETPOS	WM_USER+8	Получить позицию индикатора

При инициализации индикатора необходимо задать диапазон значений, который ставится в соответствие всей продолжительности операции. По умолчанию индикатор имеет диапазон от 0 до 100. Но можно установить любой другой в промежутке от -32 768 до 32 767. Вы можете изменять границы диапазона, используя сообщение PBM_SETRANGE. Значения новых границ содержатся в параметре **lParam** в младшем слове – нижняя граница, а в старшем – верхняя. Функция **SendMessage()** возвращает в том же формате прежние границы диапазона.

Текущее положение индикатора задается с помощью сообщения PBM_SETPOS. Новое значение позиции содержится в параметре **wParam**.

Как уже указывалось выше, управление положением индикатора производится приложением. Индикатор можно продвинуть на заданное расстояние, посылая сообщение PBM_DELTAPOS. Значение расстояния, на которое продвигается индикатор, задается параметром **wParam**.

В большинстве случаев индикатор нужно продвигать в пределах диапазона равномерными шагами. Для этого прежде всего надо установить значение шага с помощью сообщения PBM_SETSTEP. Параметр **wParam** содержит новое значение шага. По умолчанию шаг установлен равным 10.

Теперь приложение, посылая при выполнении некоторого условия сообщение PBM_STEPIT, может продвигать индикатор.

Значения границ диапазона и текущей позиции можно определить с помощью сообщений PBM_GETRANGE и PBM_GETPOS.

17.2. Пример диалога с индикатором

Текст процедуры диалога с индикатором приведен ниже.

```
!****************************************************************
integer*4 function DialogFunc5(hDlg, message, wParam, lParam)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_DialogFunc5@16' :: DialogFunc5

integer(4)  hDlg, message, wParam, lParam
integer(4)  hTBwnd, hPBWnd
integer(4)  iPos, iMul, iPr, iTime

hDialog = hDlg
iMul =1

select case(message)
 case (WM_INITDIALOG)
!--------- инициализация диалога ----------------
 hTBWnd =   GetDlgItem(hDlg, IDC_SLIDER1)
 hPBWnd =   GetDlgItem(hDlg, IDC_PROGRESS1)
  ir = SendDlgItemMessage(hDlg, IDC_SLIDER1, &
                  TBM_SETRANGE, 1, MAKELONG(1, 10))
  iPos=1
  ir = SendDlgItemMessage(hDlg, IDC_SLIDER1, TBM_SETPOS, 1, iPos)
  ir=SetDlgItemInt(hDlg, IDC_STATIC2, iPos, .FALSE.)

  ir = SendDlgItemMessage(hDlg, IDC_PROGRESS1, &
                  PBM_SETRANGE, 0, MAKELONG(0, 10))
  ir = SendDlgItemMessage(hDlg, IDC_PROGRESS1, PBM_SETSTEP, 1, 0)
  ir = SendDlgItemMessage(hDlg, IDC_PROGRESS1, PBM_SETPOS, 0, 0)

  DialogFunc5 = 1
  return

case (WM_NOTIFY)
case(WM_HSCROLL)
if (lParam==hTBWnd) then
  ir = SendDlgItemMessage(hDlg, IDC_SLIDER1, TBM_GETPOS, 0, 0)
  iPos = LOWORD(ir)
```

```
  ir=SetDlgItemInt(hDlg, IDC_STATIC2, iPos, .FALSE.)
  select case(LOWORD(wParam))
   case(TB_BOTTOM)
   case(TB_ENDTRACK)
   case(TB_LINEDOWN)
   case(TB_LINEUP)
   case(TB_TOP)
   case(TB_PAGEUP)
   case(TB_THUMBPOSITION)
   case(TB_PAGEDOWN)
   end select
  end if
case (WM_COMMAND)
!---------- обработка команд ------------------------
  select case (MakeLong(LOWORD(wParam), 0))
   case(IDC_BUTTON1)
    iTime= 200*iMul*iPos
     ir=SetTimer(hDlg, 1, iTime, NULL, 0)     ! установить часы
     iPr=0
     ir = SendDlgItemMessage(hDlg, IDC_PROGRESS1, PBM_SETPOS, iPr, 0)
   case(IDC_BUTTON2)
    ir = KillTimer(hDlg, 1)
  end select
 case (WM_SYSCOMMAND)
!--------- обработка сообщений системного меню ----------
  if (wParam == SC_CLOSE) then
   ir = KillTimer(hDlg, 1)
   ir = EndDialog (hDlg, 1)
  end if
 case (WM_TIMER)
  iPr = SendDlgItemMessage(hDlg, IDC_PROGRESS1, PBM_STEPIT, 0, 0)
end select
DialogFunc5=0
end function DialogFunc5
!*****************************************************************
```

В приведенном примере при инициализации диалога устанавливаются диапазон и начальная позиция индикатора.

Функционирование индикатора имитируется с помощью внутреннего таймера. Процесс имитации запускается и приостанавливается кнопками ПУСК и СТОП. Эти кнопки надо добавить в шаблон диалога с помощью редактора ресурсов. Кнопки имеют идентификаторы IDC_BUTTON1 и IDC_BUTTON2.

При обработке сообщения WM_COMMAND по команде ПУСК запускается таймер. Предварительно вычисляется временной интервал. В нашем примере для этого используется положение ползункового регулятора.

После того как часы приведены в действие, каждый раз при обработке сообщения WM_TIMER индикатор продвигается на один шаг.

По команде СТОП часы останавливаются. При новом запуске индикатор переходит в начальную позицию.

Дополнения, которые необходимо внести в модуль **commctrlty**, выглядят следующим образом:

```
!====== PROGRESS CONTROL ================================

!MS$IF .NOT. DEFINED (NOPROGRESS)

!---------- имя класса ----------------------------------
 character(18), parameter, public ::PROGRESS_CLASS =
"msctls_progress32"C

!---------- стили ---------------------------------

integer, parameter, public :: PBS_SMOOTH   =#01
integer, parameter, public :: PBS_VERTICAL  =#04

!--------- Сообщения ------------------------------------
integer, parameter, public :: PBM_SETRANGE     = (WM_USER+1)
integer, parameter, public :: PBM_SETPOS       = (WM_USER+2)
integer, parameter, public :: PBM_DELTAPOS     = (WM_USER+3)
integer, parameter, public :: PBM_SETSTEP      = (WM_USER+4)
integer, parameter, public :: PBM_STEPIT       = (WM_USER+5)
integer, parameter, public :: PBM_SETRANGE32   = (WM_USER+6)
integer, parameter, public :: PBM_GETRANGE     = (WM_USER+7)
integer, parameter, public :: PBM_GETPOS       = (WM_USER+8)

!MS$ENDIF !  NOPROGRESS
```

Если теперь создать исполнительный файл, запустить его и выбрать в меню позиции ***Диалоги – Элементы управления***, то на экране возникнет окно диалога, показанное на рис. 17.1.

Рис.17.1. Диалог с индикатором и ползунковым регулятором

УРОК

18. Спин

Спин (up-down control или spin control) представляет собой не что иное, как особый вид линейки прокрутки (***scroll bar***). Он состоит из пары кнопок со стрелками и не имеет полосы прокрутки. Спин обычно используется для увеличения или уменьшения численного значения какого-либо параметра.

Существует два варианта использования спина. Во-первых, он применяется как отдельный самостоятельный элемент в виде коротенькой линейки прокрутки. Во-вторых, он может работать в сочетании с другим элементом управления, который называют *приятельским окном* (*buddy window*). В качестве приятельского окна часто используется окно редактирования. Именно в этом случае в англоязычной литературе применяется термин *spin control*, в остальных случаях используется термин *up-down control*, которому в русском языке достаточно трудно подыскать удачный эквивалент. Поэтому будем всегда называть этот элемент управления спином. Внешне спин с приятельским окном воспринимается как единый элемент управления.

18.1. Создание спина

Спин можно создать с помощью функций **CreateWindow()** и **CreateWindowEx()** и задавая класс UPDOWN_CLASS. Этот класс будет зарегистрирован, только если загружена DLL. Для этого необходимо использовать функцию **InitCommonControls()**.

Однако лучше использовать специальную функцию **CreateUpDownControl(Style, x, y, cx, cy, hParent, ID, hInst, hBuddy, Min, Max, StartPos)**. Здесь параметр **Style** – стиль окна спина и должен содержать стандартные значения, например WS_CHILD, WS_VISIBLE, WS_BORDER. Кроме того, предусмотрены специализированные стили. Эти стили представлены в табл. 18.1.

Таблица 18.1. Стили окна спина

Стиль	*Значение*	*Описание*
UDS_WRAP	#0001	Прокрутка позиции спина при достижении границы шкалы
UDS_SETBUDDYINT	#0002	Автоматически изменяется текст в приятельском окне
UDS_ALIGNRIGHT	#0004	Спин размещается справа от приятельского окна
UDS_ALIGNLEFT	#0008	То же, только слева
UDS_AUTOBUDDY	#0010	Автоматически определяет предыдущее окно как приятельское
UDS_ARROWKEYS	#0020	Разрешает использование клавиш со стрелками
UDS_HORZ	#0040	Создает горизонтальный спин
UDS_NOTHOUSANDS	#0080	Запрещает отображение запятой
UDS_HOTTRACK	#0100	

Комбинируя приведенные в таблице стили, можно изменять конфигурацию спина. Некоторые из стилей спина требуют дополнительных пояснений.

По умолчанию текущая позиция не изменяется, если пользователь пытается продвинуть ее за установленные границы. Установка стиля UDS_WRAP обеспечивает прокрутку спина.

Спин со стилем UDS_SETBUDDYINT автоматически изменяет целочисленное значение в приятельском окне всякий раз, когда текущая позиция изменяется. Целое число в окне автоматически разделяется запятыми на триады. Если это вам неудобно, то следует установить стиль UDS_NOTHOUSANDS.

Если вы создаете спин, не используя функцию **CreateUpDownControl()**, то для определения приятельского окна следует установить стиль UDS_AUTOBUDDY. В результате предыдущее окно в Z-порядке автоматически становится приятельским. Этим окном может, например, быть предыдущий элемент в шаблоне диалогового окна.

Значение остальных стилей понятно из приведенного в табл. 18.1 описания.

Далее в полях **x**, **y**, **cx** и **cy** функции **CreateUpDownControl()** указываются положение верхнего левого угла и размеры окна спина. Дескриптор

родительского окна заносится в поле **hParent**. Поле **ID** содержит идентификатор спина, а **hInst** – дескриптор приложения.

Если создается спин с приятельским окном, то дескриптор этого окна должен быть указан в поле **hBuddy**.

Удобство использования функции **CreateUpDownControl()** состоит также в том, что заполняя поля **Min**, **Max** и **StartPos** уже при создании спина, можно задать границы прокрутки и начальную установку.

Если вы создаете спин в диалоговом окне средствами MDS, то для этого прежде всего необходимо открыть шаблон выбранного диалога. Добавим спин в диалог **ЭЛЕМЕНТЫ УПРАВЛЕНИЯ**.

Откроем диалоговый ресурс и, используя окно редактора ресурсов с управляющими элементами (меню ***Controls***), поместим в рабочую область окна диалога окно спина. Двойным щелчком левой клавиши мыши открываем окно ***Spin Properties*** и устанавливаем стиль окна ***Visible***. Затем переходим в окно ***Styles***, для того чтобы установить стили.

В поле **Orientation** окна ***Styles*** устанавливаем ориентацию регулятора (***Vertical***). Расположение спина задается в поле **Aligment**. Выберем, например, ***Unattached*** (неопределено).

Начнем с создания спина без приятельского окна. Поэтому сразу же переходим к полям **Wrap** и **Arrow keys**, в которых устанавливаем стили UDS_WRAP и UDS_ARROWKEYS.

В окне ***Extended Styles*** можете установить при желании дополнительные стили, которые определяют внешний вид регулятора.

18.2. Взаимодействие со спином

После создания спина можно изменять его параметры и состояние, посылая сообщения. Табл. 18.2 содержит описания этих сообщений.

Таблица 18.2. Сообщения, посылаемые спину

Сообщение	*Значение*	*Описание*
UDM_SETRANGE	WM_USER+101	Устанавливает диапазон прокрутки спина
UDM_GETRANGE	WM_USER+102	Передает диапазон прокрутки
UDM_SETPOS	WM_USER+103	Устанавливает позицию спина
UDM_GETPOS	WM_USER+104	Передает позицию спина
UDM_SETBUDDY	WM_USER+105	Определяет новое приятельское окно

Сообщение	*Значение*	*Описание*
UDM_GETBUDDY	WM_USER+106	Передает дескриптор приятельского окна
UDM_SETACCEL	WM_USER+107	Устанавливает скорость прокрутки спина
UDM_GETACCEL	WM_USER+108	Передает скорость прокрутки спина
UDM_SETBASE	WM_USER+109	Устанавливает способ представления целого
UDM_GETBASE	WM_USER+110	Передает способ представления целого

Прежде всего необходимо установить границы диапазона прокрутки. Если спин используется в диалоговом окне, то это удобно сделать на стадии инициализации диалога. Для установки границ используется сообщение UDM_SETRANGE. Параметр **lParam** должен содержать максимальное значение в младшем слове и минимальное в старшем. Сообщение UDM_GETRANGE передает пределы прокрутки спина в том же формате через значение, возвращаемое функцией **SendMessage()**.

Сообщение UDM_SETPOS устанавливает текущую позицию спина. Новая позиция передается через параметр **lParam**. Обратите внимание, что в отличие от ползункового регулятора спин делает это автоматически, когда нажата какая-либо из его стрелок. Следовательно, прикладная программа не должна установить позицию при обработке сообщения WM_VSCROLL. Текущую позицию спина можно определить, посылая сообщение UDM_GETPOS. Значение содержится в младшем слове значения, возвращаемого функцией **SendMessage()**.

Если необходимо оперативно создать приятельское окно, то надо направить спину сообщение UDM_SETBUDDY. Параметр **wParam** должен содержать дескриптор приятельского окна. В ответ функция **SendMessage()** вернет дескриптор предыдущего приятельского окна. Это необходимо для возврата к исходному состоянию. Сообщение UDM_ GETBUDDY в младшем слове результата функции **SendMessage()** передает дескриптор текущего приятельского окна. Следует помнить, что у спина и приятельского окна должно быть одно и то же родительское окно.

Если какая-либо клавиша спина постоянно нажата, позиция спина начинает непрерывно изменяться в сторону увеличения или уменьшения. Предусмотрена возможность оперативного изменения скорости прокрут-

ки. Численное значение скорости хранится в структуре типа T_UDACCEL, которая определена следующим образом:

```
type T_UDACCEL
 integer(4) :: nSec
 integer(4) :: nInc
end type T_UDACCEL.
```

Здесь поле **nSec** содержит значение интервала времени в секундах, за который положение спина изменяется на число позиций, указанное в поле **nInc**. Чтобы установить скорость, используется сообщение UDM_SETACCEL. Получить текущие значения можно, посылая сообщение UDM_GETACCEL. И в том и в другом случае **lParam** – это адрес структуры типа T_UDACCEL. При установке скорости параметр **wParam** указывает размерность матрицы структур типа T_UDACCEL. Элементы матрицы должны быть расположены строго в порядке возрастания значения **nSec**.

Для спина предусмотрено два способа представления численного значения позиции: десятичное и шестнадцатеричное. Посылая сообщение UDM_SETBASE, основание системы счисления можно оперативно изменять. Параметр **wParam** может принимать значение либо 10, либо 16. Функция **SendMessage()** возвращает текущее состояние системы счисления при использовании сообщения UDM_GETBASE.

Окно спина кроме специализированных сообщений получает и стандартные. Наибольший интерес представляет сообщение WM_VSCROLL, с которым непосредственно связано единственное нотификационное сообщение спина UDN_DELTAPOS. Операционная система посылает это сообщение родительскому окну перед тем, как позиция спина должна измениться. Это происходит при нажатии какой-либо из стрелок и перед посылкой сообщения WM_VSCROLL, которое фактически изменяет позицию.

Сообщение UDN_DELTAPOS посылается в форме WM_NOTIFY. Параметр **lParam** является указателем на структуру типа T_NM_UPDOWN. Эта структура имеет следующий вид:

```
type T_NM_UPDOWN
 type(T_NMHDR):: hdr
 integer(4)  :: iPos
 integer(4)  :: iDelta
end type.
```

Первое поле – это указатель на структуру типа T_NMHDR, о которой уже шла речь выше. Поле **iPos** содержит текущую позицию спина, **iDelta** –

целое число со знаком, которое указывает предполагаемое изменение позиции. Знак плюс соответствует кнопке ВВЕРХ, а минус – ВНИЗ.

18.3. Пример диалога с общими элементами управления

Добавим в диалог **ЭЛЕМЕНТЫ УПРАВЛЕНИЯ** спин. Начнем с создания спина без приятельского окна. Текущую позицию будем заносить в независимый статический элемент IDC_STATIC4, который предварительно поместим в шаблон диалога. Значение текущей позиции спина будем использовать в качестве множителя для индикатора процесса, который мы создали ранее.

На стадии инициализации диалога устанавливаем границы изменения позиции спина с помощью функции **SendDlgItemMessage(hDlg, IDC_ SPIN1, UDM_SETRANGE, 0, MAKELONG(10, 1))** и начальную позицию с помощью функции **SendDlgItemMessage(hDlg, IDC_SPIN1, UDM_SETPOS, 0, iMul)**.

Обработку данных проводим при получение от спина сообщения WM_VSCROLL. Начинаем с идентификации окна. В общем случае эта операция необходима, т. к. сообщение WM_VSCROLL может генерироваться несколькими элементами управления. Затем извлекаем информацию о позиции спина и выводим ее на экран.

Текст процедуры диалога теперь имеет вид, приведенный ниже.

```
!*****************************************************************
integer*4 function DialogFunc5(hDlg, message, wParam, lParam)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_DialogFunc5@16' :: DialogFunc5

integer(4)  hDlg, message, wParam, lParam
integer(4)  hTBwnd, hUDWnd, hPBWnd
integer(4)  iPos, iMul, iPr, iTime

type (T_NM_UPDOWN)  :: TUpDwn
POINTER(ip, TUpDwn)

hDialog = hDlg

select case(message)
 case (WM_INITDIALOG)
!--------- инициализация диалога ----------------
  hUDWnd  =  GetDlgItem(hDlg, IDC_SPIN1)
  hTBWnd  =  GetDlgItem(hDlg, IDC_SLIDER1)
```

```
 hPBWnd  =  GetDlgItem(hDlg, IDC_PROGRESS1)

 ir = SendDlgItemMessage(hDlg, IDC_SPIN1, &
                  UDM_SETRANGE, 0, MAKELONG(10, 1))
 iMul=1
 ir = SendDlgItemMessage(hDlg, IDC_SPIN1, UDM_SETPOS, 0, iMul)
 ir=SetDlgItemInt(hDlg, IDC_STATIC4, iMul, .FALSE.)

 ir = SendDlgItemMessage(hDlg, IDC_SLIDER1, &
                  TBM_SETRANGE, 1, MAKELONG(1, 10))
 iPos=1
 ir = SendDlgItemMessage(hDlg, IDC_SLIDER1, TBM_SETPOS, 1, iPos)
 ir=SetDlgItemInt(hDlg, IDC_STATIC2, iPos, .FALSE.)

 ir = SendDlgItemMessage(hDlg, IDC_PROGRESS1, &
                  PBM_SETRANGE, 0, MAKELONG(0, 10))
 ir = SendDlgItemMessage(hDlg, IDC_PROGRESS1, PBM_SETSTEP, 1, 0)
 ir = SendDlgItemMessage(hDlg, IDC_PROGRESS1, PBM_SETPOS, 0, 0)

 DialogFunc5 = 1
 return

case (WM_NOTIFY)
 ip = TRANSFER(lParam, ip)
 select case(TUpDwn%hdr%code)
  case(UDN_DELTAPOS)
   ir =1
 end select
case(WM_HSCROLL)
 if (lParam==hTBWnd) then
  ir = SendDlgItemMessage(hDlg, IDC_SLIDER1, TBM_GETPOS, 0, 0)
  iPos = LOWORD(ir)
  ir=SetDlgItemInt(hDlg, IDC_STATIC2, iPos, .FALSE.)
  select case(LOWORD(wParam))
   case(TB_BOTTOM)
   case(TB_ENDTRACK)
   case(TB_LINEDOWN)
   case(TB_LINEUP)
   case(TB_TOP)
   case(TB_PAGEUP)
   case(TB_THUMBPOSITION)
   case(TB_PAGEDOWN)
  end select
 end if
case(WM_VSCROLL)
```

```
  if (lParam==hUDWnd) then
   ir = SendDlgItemMessage(hDlg, IDC_SPIN1, UDM_GETPOS, 0, 0)
   iMul = LOWORD(ir)
   ir=SetDlgItemInt(hDlg, IDC_STATIC4, iMul, .FALSE.)
  end if
 case (WM_COMMAND)
!---------- обработка команд ------------------------
  select case (MakeLong(LOWORD(wParam), 0))
   case(IDC_BUTTON1)
    iTime= 200*iMul*iPos
     ir=SetTimer(hDlg, 1, iTime, NULL, 0)    ! установить часы
     iPr=0
    ir = SendDlgItemMessage(hDlg, IDC_PROGRESS1, PBM_SETPOS, iPr, 0)
   case(IDC_BUTTON2)
    ir = KillTimer(hDlg, 1)
  end select
 case (WM_SYSCOMMAND)
!--------- обработка сообщений системного меню ----------
  if (wParam == SC_CLOSE) then
   ir = KillTimer(hDlg, 1)
   ir = EndDialog (hDlg, 1)
  end if
 case (WM_TIMER)
  iPr = SendDlgItemMessage(hDlg, IDC_PROGRESS1, PBM_STEPIT, 0, 0)
end select
DialogFunc5=0
end function DialogFunc5
!*****************************************************************
```

Для того чтобы можно было пользоваться средствами создания и управления спином, необходимо дополнить файл **commctrlty.f90** следующим фрагментом:

```
!====== UPDOWN CONTROL ============================

!MS$IF .NOT. DEFINED (NOUPDOWN)

!---------- имя класса ---------------------------------
 character(16), parameter, public ::UPDOWN_CLASS ="msctls_updown32"C

!---------- costants -----------------------------------
integer, parameter, public :: UD_MAXVAL              = #7fff
integer, parameter, public :: UD_MINVAL              = -UD_MAXVAL
```

```
integer, parameter, public :: UDS_WRAP          = #0001
integer, parameter, public :: UDS_SETBUDDYINT   = #0002
integer, parameter, public :: UDS_ALIGNRIGHT    = #0004
integer, parameter, public :: UDS_ALIGNLEFT     = #0008
integer, parameter, public :: UDS_AUTOBUDDY     = #0010
integer, parameter, public :: UDS_ARROWKEYS     = #0020
integer, parameter, public :: UDS_HORZ          = #0040
integer, parameter, public :: UDS_NOTHOUSANDS   = #0080
integer, parameter, public :: UDS_HOTTRACK      = #0100

!---------- Сообщения ----------------------------------------
integer, parameter, public :: UDM_SETRANGE      = (WM_USER+101)
integer, parameter, public :: UDM_GETRANGE      = (WM_USER+102)
integer, parameter, public :: UDM_SETPOS        = (WM_USER+103)
integer, parameter, public :: UDM_GETPOS        = (WM_USER+104)
integer, parameter, public :: UDM_SETBUDDY      = (WM_USER+105)
integer, parameter, public :: UDM_GETBUDDY      = (WM_USER+106)
integer, parameter, public :: UDM_SETACCEL      = (WM_USER+107)
integer, parameter, public :: UDM_GETACCEL      = (WM_USER+108)
integer, parameter, public :: UDM_SETBASE       = (WM_USER+109)
integer, parameter, public :: UDM_GETBASE       = (WM_USER+110)

!---------- Нотификационные сообщения ----------
integer, parameter, public :: UDN_DELTAPOS      = (UDN_FIRST - 1)

!---------- Типы ----------------------------------------

type T_UDACCEL
 integer(4) :: nSec
 integer(4) :: nInc
end type T_UDACCEL

type T_NM_UPDOWN
 type(T_NMHDR):: hdr
 integer(4)    :: iPos
 integer(4)    :: iDelta
end type

!MS$ENDIF ! NOUPDOWN.
```

Кроме того, надо добавить описание интерфейса функции **CreateUp DownControl()** на тот случай, если потребуется создать спин, не прибегая к средствам MDS. Фрагмент текста, который надо поместить в файл **commctrl.f90**, имеет следующий вид:

```
!====== UPDOWN CONTROL ========================
interface
 integer(4) function CreateUpDownControl(Style, x, y, cx, cy, &
              hParent, ID, hInst, hBuddy, Min, Max, StartPos)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_CreateUpDownControl@48' ::
CreateUpDownControl
  integer(4) Style
  integer(4) x
  integer(4) y
  integer(4) cx
  integer(4) cy
  integer(4) hParent
  integer(4) ID
  integer(4) hInst
  integer(4) hBuddy
  integer(4) Min
  integer(4) Max
  integer(4) StartPos
 end function
end interface
```

После запуска приложения на экране возникнет окно диалога, показанное на рис. 18.1.

Рис. 18.1. Диалог со спином, регулятором и индикатором

Вывод на экран значений позиции спина программными средствами достаточно обременителен. Этого можно избежать, если объявить статический элемент с идентификатором IDC_STATIC4 приятельским окном спина. Для этого при инициализации диалога надо добавить строку:

```
ir = SendDlgItemMessage (hDlg, IDC_SPIN1, UDM_SETBUDDY,
GetDlgItem(hDlg, IDC_STATIC4), 0).
```

Для того чтобы не было проблем с выводом целочисленных значений в приятельское окно, надо установить в шаблоне диалога стиль UDS_SETBUDDYINT. Теперь строку вывода значений в ответ на сообщение WM_VSCROLL можно убрать, поскольку статическое окно и спин автоматически связываются и работают совместно.

В тексте программы предусмотрена возможность обработки нотификационного сообщения. Попробуйте самостоятельно воспользоваться этой возможностью. Попытайтесь, например, оперативно менять способ представления числа в зависимости от того, какая из клавиш приводится в действие.

УРОК

19. Заголовок

Заголовок (header control) – это окно, которое обычно размещается на первой строке таблицы. Он может быть разделен на части, которые иногда называют *элементами*.

Каждый элемент заголовка может содержать текст или растровое изображение. И текст и изображение помещаются внутри границ элемента. Если элемент содержит одновременно текст и изображение, то изображение располагается выше текста. Если при этом они перекрываются, то текст записывается поверх изображения.

Отдельные части заголовка отделяются друг от друга вертикальными линиями. Пользователь может перемещать разделительные линии и устанавливать таким образом ширину каждого столбца. Каждому столбцу в заголовке ставится в соответствие некоторый текст или рисунок.

19.1. Создание заголовка

Заголовок представляет собой особого вида окно. Поэтому для его создания используется функция **CreateWindow()** или **CreateWindowEx()**. Заголовок объявляется окном класса WC_HEADER. Этот класс будет зарегистрирован, если загружена DLL. Для ее загрузки необходимо использовать функцию **InitCommonControls()**.

К сожалению, для создания заголовка не удастся использовать средства MDS. Так что вам придется потрудиться, для того чтобы создать диалог, который содержит заголовки столбцов таблицы.

Окно заголовка может иметь два стиля: HDS_HORZ = 0 и HDS_BUTTONS = 2.

Стиль HDS_HORZ создает окно заголовка с горизонтальной ориентацией. Если объявлен стиль HDS_BUTTONS, то каждый элемент заголовка выглядит и ведет себя как кнопка. Этот стиль полезен, если в прикладной программе предусмотрено выполнение некоторой процедуры, когда пользователь воздействует на элемент заголовка. Например, прикладная про-

грамма может сортировать информацию по признаку, который зависит от того, какой именно столбец активизирован.

Уже после того, как окно заголовка создано, вы можете поменять начальные стили. Для этого надо использовать функции **GetWindowLong()** и **SetWindowLong()**.

В качестве примера используем заголовок в диалоге со списком. Прежде всего создадим шаблон диалога **ТАБЛИЦА** средства MDS. В окно диалога добавим элемент ***list box***. Окно списка с идентификатором IDC_LIST3 в дальнейшем будет использоваться совместно с окном заголовка.

При инициализации диалога определяем дескриптор окна списка. Дескриптор нужен для взаимодействия со списком. Затем создаем окно заголовка. Этот фрагмент текста имеет следующий вид:

```
case (WM_INITDIALOG)
!--------- инициализация диалога ----------------
 hListWnd  = GetDlgItem(hDlg, IDC_LIST3)
 hHeaderWnd = CreateWindowEx(0, WC_HEADER, " "C,  &
IOR(WS_VISIBLE, IOR(WS_CHILD, IOR(WS_BORDER, IOR
                          (HDS_BUTTONS, HDS_HORZ)))), &
        0, 0, 390, 20, hDlg, IDC_HEADER, hInst, NULL)
```

Окно диалога объявлено родительским окном заголовка. Это позволит обрабатывать сообщения окна заголовка в диалоговой процедуре. Окну заголовка ставится в соответствие некоторый идентификатор. В дальнейшем можно обращаться к окну заголовка с помощью функций **SendMessage()** или **SendDlgItemMessage()**. Первая функция использует дескриптор **hHeaderWnd**, а вторая – идентификатор элемента управления диалога IDC_HEADER.

Следующий шаг состоит в инициализации самого заголовка. Надо описать свойства заголовка, разбить его на нужное число частей, снабдить каждую из них текстом. Вся необходимая для инициализации информация хранится в структуре типа T_HD_ITEM, которая определена следующим образом:

```
type T_HD_ITEM
 integer(4) ::  mask
 integer(4) ::  cxy
 integer(4) ::  pszText
 integer(4) ::  hbm
 integer(4) ::  cchTextMax
 integer(4) ::  fmt
 integer(4) ::  lParam
```

```
 integer(4) ::  iImage
 integer(4) ::  iOrder
end type T_HD_ITEM
```

Структура T_HD_ITEM содержит информацию о каждом элементе заголовка.

В поле **mask** указывается, в каком из полей – **cxy**, **pszText**, **hbm**, **fmt**, **lParam**, **iImage** или **iOrder** – содержатся реальные значения, определяющие вид элемента заголовка. Это поле может содержать значения, приведенные в табл. 19.1.

Таблица. 19.1. Флаги, устанавливаемые в поле mask

Флаг	*Значение*	*Описание*
HDI_WIDTH	#0001	Поле cxy задает ширину
HDI_HEIGHT	#0001	Поле cxy задает высоту
HDI_TEXT	#0002	Информация содержится в поле psztext
HDI_FORMAT	#0004	Информация содержится в поле fmt
HDI_LPARAM	#0008	Информация содержится в поле lparam
HDI_BITMAP	#0010	Заголовок содержит изображение
HDI_IMAGE	#0020	Информация содержится в поле iimage
HDI_DI_SETITEM	#0040	Информация об элементе сохраняется Системой
HDI_ORDER	#0080	Информация содержится в поле iorder

В поле **cxy** указывается ширина или высота элемента заголовка.

Если элементы заголовка содержат текст, то поле **pszText** – это указатель на текстовую строку. В этом случае в поле **cchTextMax** указывается длина строки.

Поле **hbm** содержит дескриптор растрового изображения.

В поле **fmt** указывается формат элемента заголовка. Значения, которые может содержать это поле, приведены в табл. 19.2.

Таблица 19.2. Флаги формата заголовка

Флаг	*Значение*	*Описание*
HDF_LEFT	#0000	Выравнивание текста или изображения влево
HDF_RIGHT	#0001	Выравнивание текста или изображения вправо

Флаг	*Значение*	*Описание*
HDF_CENTER	#0002	Центрирование текста или изображения
HDF_JUSTIFYMASK	#0003	
HDF_RTLREADING	#0004	Текст справа налево
Флаги, могущие объединяться с предыдущими		
HDF_IMAGE	#0800	С элементом связан список изображений
HDF_BITMAP_ON_RIGHT	#1000	Изображение смещается направо
HDF_BITMAP	#2000	Элемент содержит изображение
HDF_STRING	#4000	Элемент содержит текст
HDF_OWNERDRAW	#8000	За прорисовку отвечает родительское окно

Если с заголовком связан список изображений, то поле **iImage** задает индекс изображения, связанного с элементом. Если список изображений отсутствует, то **iImage** = -1.

Поле **lParam** – это любые данные, которые определяются пользователем.

Поскольку информация о заголовке передается с помощью сообщений, рассмотрим процедуру взаимодействия приложения с окном заголовка.

19.2. Взаимодействие приложения с окном заголовка

Прикладная программа обменивается с окном заголовка сообщениями. Уже после того, как окно заголовка создано, вы можете изменить внешний вид заголовка, удалить или добавить элемент, ввести новый текст в каждый столбец.

Все сообщения имеют префикс HDM_. В табл. 19.3 приведено их описание.

Таблица 19.3. Сообщения, посылаемые окну заголовка

Сообщение	*Значение*	*Описание*
HDM_FIRST	#1200	
HDM_GETITEMCOUNT	HDM_FIRST + 0	Возвращает число элементов в заголовке
HDM_INSERTITEM	HDM_FIRST +1	Вставить новый элемент заголовка

Сообщение	*Значение*	*Описание*
HDM_DELETEITEM	HDM_FIRST + 2	Удалить элемент с индексом wparam
HDM_GETITEMA	HDM_FIRST + 3	Получить свойства элемента
HDM_SETITEM	HDM_FIRST + 4	Установить свойства элемента
HDM_LAYOUT	HDM_FIRST + 5	Разместить заголовок в заданной области
HDM_HITTEST	HDM_FIRST + 6	Устанавливает, есть ли точка с заданными свойствами
HDM_GETITEMRECT	HDM_FIRST + 7	Передает размеры элемента с индексом wparam
HDM_SETIMAGELIST	HDM_FIRST + 8	Объявляет список изображений
HDM_GETIMAGELIST	HDM_FIRST + 9	Передает список изображений
HDM_ORDERTOINDEX	HDM_FIRST + 15	Положение в заголовке элемента с индексом wparam
HDM_CREATEDRAGIMAGE	HDM_FIRST + 16	Создать рисунок для перетаскивания
HDM_GETORDERARRAY	HDM_FIRST + 17	Передает порядок следования элементов
HDM_SETORDERARRAY	HDM_FIRST + 18	Устанавливает порядок следования элементов
HDM_SETHOTDIVIDER	HDM_FIRST + 19	Устанавливает положение точки при перетаскивании

На этапе инициализации используется сообщение HDM_INSERTITEM. Параметр **wParam** указывает индекс элемента, после которого будет помещен новый элемент. Указатель на структуру типа T_HD_ITEM содержится в параметре **lParam**. После обработки этого сообщения возвращается индекс добавленного элемента, который можно использовать в других сообщениях.

Вы можете использовать сообщение HDM_SETITEM для оперативного изменения свойств элемента, а сообщение HDM_GETITEM – для их определения. Параметр **wParam** указывает индекс элемента. Указатель на структуру типа T_HD_ITEM содержится в параметре **lParam**.

Существующий элемент заголовка удаляется с помощью сообщения HDM_DELETEITEM. Параметр **wParam** указывает индекс элемента.

Общее число элементов устанавливается с помощью сообщения HDM_GETITEMCOUNT.

Обычно размеры и положение заголовка согласуются с размерами и координатами выделенной для его размещения области. Для этой цели следует использовать сообщение HDM_LAYOUT. Информация передается в виде структуры типа T_HD_LAYOUT, которая определена следующим образом:

```
type T_HD_LAYOUT
 integer(4) :: prc
 integer(4) :: pwpos
end type T_HD_LAYOUT
```

Первое поле содержит указатель на структуру типа T_RECT с координатами области, выделенной для окна заголовка. Поле **pwpos** – это указатель на структуру типа T_WINDOWPOS.

В ответ на сообщение HDM_LAYOUT производится заполнение полей структуры T_WINDOWPOS значениями, которые согласуются с координатами, заданными в структуре T_RECT. Высота заголовка устанавливается с учетом ширины границ и высоты символов шрифта, которые используются в данный момент в контексте заголовка.

Для установки размеров и позиции заголовка при инициализации его окно должно быть изначально невидимым. После посылки сообщения HDM_LAYOUT полученные значения позиции используются в аргументах функции **SetWindowPos()**.

Возможности управления элементами заголовка значительно расширяются, когда в структуре T_HD_ITEM объявляется формат HDF_OWNERDRAW. Изображение элемента в этом случае формируется приложением в ответ на сообщение WM_DRAWITEM, которое посылается родительскому окну. Более подробную информацию по этому вопросу можно получить через справочную систему MDS.

19.3. Пример диалога с заголовком

Текст процедуры диалога, содержащего заголовок, приведен ниже.

```
!****************************************************************
integer*4 function DialogFunc6(hDlg, message, wParam, lParam)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_DialogFunc6@16' :: DialogFunc6

integer(4)  hDlg, message, wParam, lParam
integer(4)  hListwnd, hHeaderWnd, index

character(10), dimension(0:2), parameter :: HeadText = &
          (/"Ф.И.О. "C, &
            "Возраст"C, &
            "Палата "C/)

type (T_HD_ITEM)  :: hdi

type (T_HD_NOTIFY):: THd
POINTER(ip, THd)

IDC_HEADER = 10000
hDialog = hDlg

select case(message)
 case (WM_INITDIALOG)
!---------- инициализация диалога ----------------
  hListWnd  = GetDlgItem(hDlg, IDC_LIST3)
  hHeaderWnd = CreateWindowEx(0, WC_HEADER, " "C, &
  IOR(WS_VISIBLE, IOR(WS_CHILD, IOR(WS_BORDER,
IOR(HDS_BUTTONS, HDS_HORZ)))), &
            10, 30, 390, 20, hDlg, IDC_HEADER, hInst, NULL)

   hdi%mask = IOR(HDI_FORMAT, IOR(HDI_TEXT, HDI_WIDTH))
   hdi%cxy = 130
   hdi%cchTextMax = 10
   hdi%fmt = IOR(HDF_STRING, HDF_CENTER)
   hdi%iImage = -1

  do i=0, 2
   hdi%pszText = LOC(HeadText(i))
   index = SendDlgItemMessage(hDlg, IDC_HEADER, HDM_INSERTITEM, i,
                              LOC(hdi))
 end do
 index = SendMessage(hHeaderWnd, HDM_GETITEMCOUNT, 0, 0)

  DialogFunc6 = 1
```

```
  return
 case (WM_NOTIFY)
 !------------ нотификационные сообщения ---------------------------------
  ip = TRANSFER(lParam, ip)
  select case(THd%hdr%code)
   case(HDN_ITEMCHANGING)
   case(HDN_BEGINTRACK)
     ir = MessageBox(ghwndMain, "Двигаем?"C, "Сообщение"C, MB_OK)
   case(HDN_ITEMCLICK)
     ir = MessageBox(ghwndMain, "Выбор сделан"C, "Сообщение"C, MB_OK)
  end select
 case (WM_COMMAND)
 !---------- обработка команд ------------------------------------------------
  select case (MakeLong(LOWORD(wParam), 0))
  end select
 case (WM_SYSCOMMAND)
 !--------- обработка сообщений системного меню ----------------------
  if (wParam == SC_CLOSE) then
   ir = EndDialog (hDlg, 1)
  end if
 end select
 DialogFunc6=0
 end function DialogFunc6
 !*********************************************************************
```

В диалоге предусмотрена обработка нотификационных сообщений. Нотификационные сообщения посылаются родительскому окну, когда пользователь нажимает элемент, перемещает разделитель или изменяются атрибуты элемента. Родительское окно получает сообщения в форме сообщений WM_NOTIFY. Значения кодов нотификационных сообщений приведены в табл. 19.4.

Таблица 19.4. Нотификационные сообщения окна заголовка

Сообщение	*Значение*	*Описание*
HDN_FIRST	-300	
HDN_ITEMCHANGING	HDN_FIRST-0	Элемент будет выбран
HDN_ITEMCHANGED	HDN_FIRST-1	Элемент выбран
HDN_ITEMCLICK	HDN_FIRST-2	Щелчок по элементу
HDN_ITEMDBLCLICK	HDN_FIRST-3	Двойной щелчок по элементу
HDN_DIVIDERDBLCLICK	HDN_FIRST-5	Двойной щелчок по разделителю

Сообщение	*Значение*	*Описание*
HDN_BEGINTRACK	HDN_FIRST-6	Начало перемещения разделителя
HDN_ENDTRACK	HDN_FIRST-7	Конец перемещения разделителя
HDN_TRACK	HDN_FIRST-8	Разделитель был перемещен
HDN_GETDISPINFO	HDN_FIRST-9	
HDN_BEGINDRAG	HDN_FIRST-10	Начало процедуры перетаскивания
HDN_ENDDRAG	HDN_FIRST-11	Конец процедуры перетаскивания
HDN_LAST	-399	

В качестве демонстрации в нашем диалоге на экран выводится информация о предстоящем перемещении разделителя (код HDN_ BEGINTRACK) и о выбранном элементе (код HDN_ITEMDBLCLICK).

Если теперь создать исполнительный файл, запустить его и выбрать в меню позиции ***Диалоги – Таблица***, то на экране возникнет окно диалога, показанное на рис. 19.1. Мы видим здесь заголовок таблицы.

Рис. 19.1. Таблица с заголовком

Для того чтобы можно было общаться с заголовком таблицы, необходимо дополнить файл **commctrlty.f90** следующим фрагментом:

```
!====== HEADER CONTROL ==========================================
!MS$IF .NOT. DEFINED (NOHEADER)

!---------- имя класса ---------------------------------

 character(12), parameter, public :: WC_HEADER = "SysHeader32"C

!---------- costants -----------------------------------
```

```
!---------- стиль ----------------------------------
integer, parameter, public :: HDS_HORZ          = 0
integer, parameter, public :: HDS_BUTTONS       = 2
integer, parameter, public :: HDS_HOTTRACK      = 4
integer, parameter, public :: HDS_HIDDEN        = 8

!--------- Сообщения ------------------------------------------
integer, parameter, public :: HDM_GETITEMCOUNT      = HDM_FIRST + 0
integer, parameter, public :: HDM_INSERTITEM        = HDM_FIRST + 1
integer, parameter, public :: HDM_DELETEITEM        = HDM_FIRST + 2
integer, parameter, public :: HDM_GETITEMA          = HDM_FIRST + 3
integer, parameter, public :: HDM_SETITEM           = HDM_FIRST + 4
integer, parameter, public :: HDM_LAYOUT            = HDM_FIRST + 5
integer, parameter, public :: HDM_HITTEST           = HDM_FIRST + 6
integer, parameter, public :: HDM_GETITEMRECT       = HDM_FIRST + 7
integer, parameter, public :: HDM_SETIMAGELIST      = HDM_FIRST + 8
integer, parameter, public :: HDM_GETIMAGELIST      = HDM_FIRST + 9
integer, parameter, public :: HDM_ORDERTOINDEX      = HDM_FIRST + 15
integer, parameter, public :: HDM_CREATEDRAGIMAGE = HDM_FIRST + 16
integer, parameter, public :: HDM_GETORDERARRAY     = HDM_FIRST + 17
integer, parameter, public :: HDM_SETORDERARRAY     = HDM_FIRST + 18
integer, parameter, public :: HDM_SETHOTDIVIDER     = HDM_FIRST + 19

!-------------------- Нотификационные сообщения ---------------
integer, parameter, public :: HDN_ITEMCHANGING      = HDN_FIRST-0
integer, parameter, public :: HDN_ITEMCHANGED       = HDN_FIRST-1
integer, parameter, public :: HDN_ITEMCLICK         = HDN_FIRST-2
integer, parameter, public :: HDN_ITEMDBLCLICK      = HDN_FIRST-3
integer, parameter, public :: HDN_DIVIDERDBLCLICK   = HDN_FIRST-5
integer, parameter, public :: HDN_BEGINTRACK        = HDN_FIRST-6
integer, parameter, public :: HDN_ENDTRACK          = HDN_FIRST-7
integer, parameter, public :: HDN_TRACK             = HDN_FIRST-8
integer, parameter, public :: HDN_GETDISPINFO       = HDN_FIRST-9
integer, parameter, public :: HDN_BEGINDRAG         = HDN_FIRST-10
integer, parameter, public :: HDN_ENDDRAG           = HDN_FIRST-11

!---------- флаги маски ----------------------------------------
integer, parameter, public :: HDI_WIDTH         = #1
integer, parameter, public :: HDI_HEIGHT        = #1
integer, parameter, public :: HDI_TEXT          = #2
integer, parameter, public :: HDI_FORMAT        = #4
integer, parameter, public :: HDI_LPARAM        = #8
integer, parameter, public :: HDI_BITMAP        = #10
integer, parameter, public :: HDI_IMAGE         = #20
```

```
integer, parameter, public :: HDI_DI_SETITEM     = #40
integer, parameter, public :: HDI_ORDER          = #80

!---------- флаги формата ------------------------------------
integer, parameter, public :: HDF_LEFT           = #0
integer, parameter, public :: HDF_RIGHT          = #1
integer, parameter, public :: HDF_CENTER         = #2
integer, parameter, public :: HDF_JUSTIFYMASK    = #3
integer, parameter, public :: HDF_RTLREADING     = #4

integer, parameter, public :: HDF_OWNERDRAW      = #8000
integer, parameter, public :: HDF_STRING         = #4000
integer, parameter, public :: HDF_BITMAP         = #2000
integer, parameter, public :: HDF_BITMAP_ON_RIGHT = #1000
integer, parameter, public :: HDF_IMAGE          = #0800

!------------ флаги результатов опроса --------------------
integer, parameter, public :: HHT_NOWHERE     =#0001
integer, parameter, public :: HHT_ONHEADER    =#0002
integer, parameter, public :: HHT_ONDIVIDER   =#0004
integer, parameter, public :: HHT_ONDIVOPEN   =#0008
integer, parameter, public :: HHT_ABOVE       =#0100
integer, parameter, public :: HHT_BELOW       =#0200
integer, parameter, public :: HHT_TORIGHT     =#0400
integer, parameter, public :: HHT_TOLEFT      =#0800

!------------ Типы ---------------------------------------
type T_HD_LAYOUT
 integer(4) :: prc
 integer(4) :: pwpos
end type T_HD_LAYOUT

type T_HD_HITTESTINFO
 type(T_POINT):: pt
 integer(4)  :: flags
 integer(4)  :: iItem
end type T_HD_HITTESTINFO

type T_HD_ITEM
 integer(4) ::  mask
 integer(4) ::  cxy
 integer(4) ::  pszText
 integer(4) ::  hbm
 integer(4) ::  cchTextMax
 integer(4) ::  fmt
 integer(4) ::  lParam
```

```
 integer(4) ::  iImage
 integer(4) ::  lOrder
end type T_HD_ITEM

type T_ HD_NOTIFY
 type(T_NMHDR)   :: hdr
 integer(4)      :: iItem
 integer(4)      :: iButton
 type(T_HD_ITEM) :: pitem
end type T_ HD_NOTIFY

type  T_NMHDDISPINFO
 type(T_NMHDR):: hdr
 integer(4)  :: iItem
 integer(4)  :: mask
 integer(4)  :: pszText
 integer(4)  :: cchTextMax
 integer(4)  :: iImage
 integer(4)  :: lParam
end type T_NMHDDISPINFO

!MS$ENDIF !  NOHEADER
```

УРОК

20. Списки изображений

Список изображений (image list) – это совокупность изображений одинакового размера, каждое из которых имеет собственный номер или индекс. Списки изображений используются для эффективного управления большими наборами пиктограмм или рисунков. Все рисунки списка помещаются в расширенном едином растровом изображении. Кроме того, список может включать одноцветные маски, используемые при перемещении изображения.

Интерфейс Microsoft Win32 API предоставляет возможность вывода изображения, создания и уничтожения списка изображения, добавление и удаление изображения, замену изображения, их объединение и перемещение.

Список изображений не является окном и используется совместно с другими элементами управления. Этот элемент всего лишь структура в памяти, которая обеспечивает простой доступ к изображениям.

Различают два типа списков изображений: немаскируемый и маскируемый. Немаскируемый список содержит только цветные изображения. Маскируемый список состоит из двух рисунков равного размера. Первый, цветной содержит изображения, а второй, одноцветный содержит ряд масок для каждого из цветных рисунков.

Когда выводится немаскируемое изображение, оно помещается поверх фона. Маскируемое изображение выводится в виде прозрачной картинки, через которую видно прежнее изображение.

Для того чтобы работать со списком изображений, вы должны включить в файлы **commctrl.f90** и **commctrlty.f90** интерфейсы функций и константы. Кроме того, в приложении надо использовать функцию **InitCommonControls()**, чтобы загрузить динамическую библиотеку общих элементов управления.

20.1. Создание списка изображений

Список изображений создается функцией **ImageList_Create (cx, cy, flags, cInitial, cGrow)**. Интерфейс этой функции выглядит следующим образом:

```
interface
integer(4) function ImageList_Create(cx, cy, flags, cInitial, cGrow)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_ImageList_Create@20' ::
ImageList_Create
  integer cx, cy, flags, cInitial, cGrow
end function
end interface
```

Первые два аргумента, **cx** и **cy** этой функции определяют размер каждого изображения, а третий устанавливает тип списка изображений. Описание флагов, устанавливающих тип, приведены в табл. 20.1.

Таблица 20.1. Флаги типа списка изображений

Флаг	*Значение*	*Описание*
ILC_COLOR	#0000	Флаг по умолчанию, обычно ILC_COLOR4
ILC_MASK	#0001	Создается маскированное изображение
ILC_COLOR4	#0004	16-цветный рисунок
ILC_COLOR8	#0008	256-цветный рисунок
ILC_COLOR16	#0010	65 536 цветов
ILC_COLOR24	#0018	2**24 цветов
ILC_COLOR32	#0020	2**32 цветов
ILC_COLORDDB	#00 FE	Устаревший формат, зависящий от устройства

Четвертый аргумент содержит число рисунков. Система использует это число, совместно с **cx** и **cy**, для определения размера растрового изображения. Если на каком-то этапе число картинок достигнет предельного значения, то Система автоматически расширит объем памяти так, чтобы можно было разместить еще **cGrow** элементов списка.

Для немаскируемого списка изображений, функция создает одиночный растр, достаточно большой для того, чтобы разместить заданное число изображений данных размеров. Затем она создает экранно-совместимый контекст устройства и объявляет этот растр в качестве текущего.

Для маскируемого списка изображений, функция создает два растра и два экранно-совместимых контекста устройства. Она помещает растр изображения в один контекст и растр маски в другой.

В случае успешного завершения функция **ImageList_Create()** возвращает программе обработки дескриптор списка **hImageList**.

20.2. Управление списком изображений

После создания списка изображений вы имеете возможность управлять списком. Для этой цели в Win32 API предусмотрено три десятка специальных функций. Интерфейсы этих функций следует описать в модуле **commctrl**. Текст фрагмента, который следует добавить, приведен ниже.

```
!====== IMAGE APIS ========================================

!MS$IF .NOT. DEFINED  (NOIMAGEAPIS)

interface
integer(4) function ImageList_Create(cx, cy, flags, cInitial, cGrow)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_ImageList_Create@20' ::
ImageList_Create
  integer cx, cy, flags, cInitial, cGrow
end function
end interface

interface
logical function ImageList_Destroy(himl)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_ImageList_Destroy@4' ::
ImageList_Destroy
  integer himl
end function
end interface

interface
integer(4) function ImageList_GetImageCount(himl)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_ImageList_GetImageCount@4' ::
ImageList_GetImageCount
  integer himl
end function
end interface

interface
logical function ImageList_SetImageCount(himl, uNewCount)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_ImageList_SetImageCount@8' ::
ImageList_SetImageCount
 integer himl, uNewCount
end function
end interface
```

```
interface
integer(4) function ImageList_Add(himl, hbmImage, hbmMask)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_ImageList_Add@12' :: ImageList_Add
  integer himl, hbmImage, hbmMask
end function
end interface

interface
integer(4) function ImageList_ReplaceIcon(himl, i, hicon)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_ImageList_ReplaceIcon@12' ::
ImageList_ReplaceIcon
  integer himl, i, hicon
end function
end interface

interface
integer(4) function ImageList_SetBkColor(himl, clrBk)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_ImageList_SetBkColor@8' ::
ImageList_SetBkColor
  integer himl, clrBk
end function
end interface

interface
integer(4) function ImageList_GetBkColor(himl)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_ImageList_GetBkColor@4' ::
ImageList_GetBkColor
  integer himl
end function
end interface

interface
logical function ImageList_SetOverlayImage(himl, iImage, iOverlay)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_ImageList_SetOverlayImage@12' ::
ImageList_SetOverlayImage
 integer himl, iImage, iOverlay
end function
end interface

interface
logical function ImageList_Draw(himl, i, hdcDst, x, y, fStyle)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_ImageList_Draw@24' ::
ImageList_Draw
```

```
 integer himl, i, hdcDst, x, y, fStyle
end function
end interface

interface
integer(4) function ImageList_Replace(himl, i, hbmImage, hbmMask)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_ImageList_Replace@16' ::
ImageList_Replace
  integer himl, i, hbmImage, hbmMask
end function
end interface

interface
integer(4) function ImageList_AddMasked(himl, hbmImage, crMask)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_ImageList_AddMasked@12' ::
ImageList_AddMasked
  integer himl, hbmImage, crMask
end function
end interface

interface
logical function ImageList_DrawEx(himl, i, hdcDst, x, y, dx, dy, rgbBk, rgbFg,
fStyle)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_ImageList_DrawEx@40' ::
ImageList_DrawEx
 integer himl, i, hdcDst, x, y, dx, dy, rgbBk, rgbFg, fStyle
end function
end interface

interface
logical function ImageList_DrawIndirect(pimldp)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_ImageList_DrawIndirect@4' ::
ImageList_DrawIndirect
 integer pimldp
end function
end interface

interface
logical function ImageList_Remove(himl, i)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_ImageList_Remove@8' ::
ImageList_Remove
 integer himl, i
end function
```

```
end interface

interface
integer(4) function ImageList_GetIcon(himl, i, flags)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_ImageList_GetIcon@12' ::
ImageList_GetIcon
 integer himl, i, flags
end function
end interface

interface
integer(4) function ImageList_LoadImage(hi, lpbmp, cx, cGrow, crMask, uType,
uFlags)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_ImageList_LoadImage@28' ::
ImageList_LoadImage
 integer hi, lpbmp, cx, cGrow, crMask, uType, uFlags
end function
end interface


interface
logical function ImageList_Copy(himlDst, iDst, himlSrc, iSrc, uFlags)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_ImageList_Copy@20' ::
ImageList_Copy
 integer himlDst, iDst, himlSrc, iSrc, uFlags
end function
end interface

interface
logical function ImageList_BeginDrag(himlTrack, iTrack, dxHotspot, dyHotspot)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_ImageList_BeginDrag@16' ::
ImageList_BeginDrag
 integer himlTrack, iTrack, dxHotspot, dyHotspot
end function
end interface

interface
integer(4) function ImageList_EndDrag()
!MS$ ATTRIBUTES STDCALL, ALIAS : '_ImageList_EndDrag@0' ::
ImageList_EndDrag
end function
end interface
```

```
interface
logical function ImageList_DragEnter(hwndLock, x, y)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_ImageList_DragEnter@12' ::
ImageList_DragEnter
 integer hwndLock, x, y
end function
end interface

interface
logical function ImageList_DragLeave(hwndLock)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_ImageList_DragLeave@4' ::
ImageList_DragLeave
 integer hwndLock
end function
end interface

interface
logical function ImageList_DragMove(x, y)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_ImageList_DragMove@8' ::
ImageList_DragMove
 integer x, y
end function
end interface

interface
logical function ImageList_SetDragCursorImage(himlDrag, iDrag, dxHotspot,
dyHotspot)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_ImageList_SetDragCursorImage@16'
:: ImageList_SetDragCursorImage
 integer himlDrag, iDrag, dxHotspot, dyHotspot
end function
end interface

interface
integer(4) function ImageList_GetDragImage(ppt, pptHotspot)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_ImageList_GetDragImage@8' ::
ImageList_GetDragImage
 integer ppt, pptHotspot
end function
end interface
```

```
interface
logical function ImageList_DragShowNolock(fShow)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_ImageList_DragShowNolock@4' ::
ImageList_DragShowNolock
 logical fShow
end function
end interface

interface
logical function ImageList_GetIconSize(himl, cx, cy)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_ImageList_GetIconSize@12' ::
ImageList_GetIconSize
 integer himl, cx, cy
end function
end interface

interface
logical function ImageList_SetIconSize(himl, cx, cy)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_ImageList_SetIconSize@12' ::
ImageList_SetIconSize
 integer himl, cx, cy
end function
end interface

interface
logical function ImageList_GetImageInfo(himl, i, pImageInfo)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_ImageList_GetImageInfo@12' ::
ImageList_GetImageInfo
 integer himl, i, pImageInfo
end function
end interface

interface
integer(4) function ImageList_Merge(himl1, i1, himl2, i2, dx, dy)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_ImageList_Merge@8' ::
ImageList_Merge
 integer himl1, i1, himl2, i2, dx, dy
end function
end interface

!MS$ENDIF !   NOIMAGEAPIS

!***************************************************
```

Простейшая из них – это функция **ImageList_Destroy(hImL)**. Ее единственный аргумент – это дескриптор списка изображений. Ее назначение вполне очевидно.

Вы можете добавлять растровые изображения, иконки, курсоры к списку изображений. Для того чтобы добавить в список растровое изображение, необходимо использовать функцию **ImageList_Add (hImL, hBmImage, hBmMask)**. Первый аргумент функции – это дескриптор списка изображений, второй – дескриптор добавляемого изображения. Третий аргумент – это дескриптор растрового изображения маски. Для немаскируемых списков изображений последний аргумент игнорируется.

Создание рисунка маски требует дополнительной художественной работы. Ее можно избежать, если использовать функцию **ImageList_AddMasked (hImL, hBmImage, crMask)**, которая создает маску автоматически. Третий аргумент этой функции задает цвет пикселов, которые должны быть прозрачными.

Поскольку функции **ImageList_Add()** и **ImageList_AddMasked()** добавляют к списку новое изображение, то они назначают каждому из них индекс(порядковый номер). Первое изображение в списке имеет индекс нуль. При добавлении изображения функции возвращают индекс.

Функция **ImageList_Replace(hImL, i, hBmImage, hBmMask)** заменяет изображение в списке. Аргумент **i** – это номер изображения, которое надо заменить. Значения остальных аргументов совпадают с соответствующими аргументами функции **ImageList_Add()**.

Функция **ImageList_Remove(hImL, i)** удаляет изображение с номером **i** из списка с дескриптором **hImL**.

Для вывода изображения предусмотрена пара функций – **ImageList_Draw (hImL, i, hdcDst, x, y, fStyle)** и **ImageList_DrawEx (hImL, i, hdcDst, x, y, dx, dy, rgbBk, rgbFg, fStyle)**. Первый аргумент этих функций, **hImL** – это дескриптор списка изображений, второй, **i** – номер изображения, третий, **hdcDst** – дескриптор контекста, на который будет копироваться изображение, далее следуют координаты рисунка. Последний аргумент, **fStyle** – это флаг прорисовки, возможные значения которого приведены в табл. 20.2.

Таблица 20.2. Флаги прорисовки отдельного изображения

Флаг	*Значение*	*Описание*
ILD_NORMAL	#0000	Обычное копирование изображения
ILD_TRANSPARENT	#0001	Каждый белый бит маски делает бит рисунка прозрачным
ILD_BLEND25	#0002	Снижение интенсивности цветов на 25 %
ILD_BLEND50	#0004	Снижение интенсивности цветов на 50 %
ILD_MASK	#0010	Прорисовка маски
ILD_IMAGE	#0020	Прорисовка изображения
ILD_SELECTED	#0004	Ild_blend50
ILD_FOCUS	#0002	Ild_blend25
ILD_BLEND	#0004	Ild_blend50

Функция **ImageList_DrawEx()** позволяет задать размер изображения (аргументы **dx**, **dy**) и цвета подложки и изображения(аргументы **rgbBk**, **rgbFg**).

Если вы определяете стиль ILD_TRANSPARENT то, функции **ImageList_Draw()** и **ImageList_DrawEx()** сначала выполняют логическую операцию **AND** битов изображения и битов маски, а затем – логическую операцию **XOR** результата первой операции и битов фона. В результате на изображении появляются прозрачные области.

Обширность списка функций не позволяет описать каждую из них. Поэтому вам придется по мере надобности обращаться к справочной системе MDS.

Привлекательность списка изображений состоит в том, что для его обслуживания предусмотрена группа специальных функций, которые дают возможность пользователю перемещать изображения(***drag – and – drop***) на экране с минимальными затратами кода. Функции перемещают изображение без скачков, в цвете, без высвечивания курсора и без заметного мерцания. Перемещаться могут и маскируемые и немаскируемые изображения.

Эта процедура может оказаться удобной, например, при создании изображений сложной схемы из набора стандартных элементов. Продемонстрируем эту возможность на примере диалога, описание которого приведено ниже.

20.3. Пример диалога со списком изображений

Добавим в проект диалог **СПИСОК_ИЗОБРАЖЕНИЙ**, используя средства MDS. Список изображений создаем при инициализации диалога. Предварительно стандартными средствами создадим растровое изображение, которое содержит все рисунки будущего списка. В нашем примере список содержит три рисунка электрических цепей, размером 63 x 63 пиксела.

После вызова функции **hImList = ImageList_Create(63, 63, ILC_COLOR, 4, 4)** добавляем изображения в список **ir = ImageList_Add (hImList, hImbmp, NULL)**.

Создание списка и заполнение его рисунками вовсе не означает, что изображение появится на экране. Для того чтобы это произошло, необходимо использовать функцию **ImageList_Draw()**. Прорисовку удобно произвести в ответ на сообщение WM_PAINT. В этом случае контекст устройства необходимо получить с помощью функции **hDCDlg = BeginPaint (hDlg, ps)**, а обработку сообщения обязательно завершить функцией **ir = EndPaint (hDlg, ps)**.

Окончательный текст диалоговой функции имеет следующий вид:

```
!*********************************************************************
integer*4 function DialogFunc7(hDlg, message, wParam, lParam)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_DialogFunc7@16' :: DialogFunc7

integer(4)          :: hDlg, message, wParam, lParam
integer(4)          :: hImList, hImListWn, hDCDlg, iNb, iP

logical             :: bCapture

type (T_PAINTSTRUCT)     :: ps
type (T_RECT), parameter  :: rLittleRect=T_RECT(0, 0, 64, 256)
type (T_POINT)           :: Point, pHotSpot

wParam=wParam
hDialog=hDlg
lParam = lParam
select case (message)
 case (WM_INITDIALOG)
  iXBorder  = GetSystemMetrics(SM_CXBORDER)+2
  iYBorder  = GetSystemMetrics(SM_CYBORDER)+2
  iYCaption = GetSystemMetrics(SM_CYCAPTION)
  hDC=GetDC(hDlg)
  hImList =ImageList_Create(63, 63, ILC_COLOR, 4, 4)
```

```
 hImbmp  = LoadBitmap(hInst, IDB_BITMAP)
 ir= ImageList_Add(hImList, hImbmp, NULL)
 DialogFunc7 = 1
 return
case (WM_PAINT)
 hDCDlg=BeginPaint(hDlg, ps)
 do i=0, 3
  ir=ImageList_Draw(hImList, i, hDCDlg, 0, 63*i, ILD_NORMAL)
 end do
 ir=EndPaint(hDlg, ps)
 DialogFunc7 = 0
 return
case (WM_LBUTTONDOWN)
 Point%x=LOWORD(lParam)
 Point%y=HIWORD(lParam)
 if(.NOT.PtInRect(rLittleRect, Point)) then
  bCapture =.FALSE.
 else
  iNb=INT(Point%y/63)
  pHotSpot%x=Point%x
  pHotSpot%y=Point%y-iNb*63
  ir=ImageList_BeginDrag(hImList, iNb, pHotSpot%x, pHotSpot%y)
  ir=ImageList_DragEnter(hDlg, Point%x+iXBorder, &
                  Point%y+iYBorder+iYCaption)
  ir=SetCapture(hDlg)
  bCapture =.TRUE.
 end if

case (WM_LBUTTONUP)
 if(bCapture) then
  Point%x=LOWORD(lParam)- pHotSpot%x
  Point%y=HIWORD(lParam)- pHotSpot%y

  ir=ImageList_EndDrag()
  ir=ImageList_DragLeave(hDlg)
  ir=ImageList_Draw(hImList, iNB, hDC, Point%x, Point%y, ILD_NORMAL)
  ir =ReleaseCapture()
  bCapture =.FALSE.
 end if
case (WM_MOUSEMOVE)
 Point%x=LOWORD(lParam)
 Point%y=HIWORD(lParam)
 if(bCapture) then
  ir=ImageList_DragMove(LOWORD(lParam)+iXBorder, &
```

```
    HIWORD(lParam)+iYBorder+iYCaption)
  end if
 case (WM_COMMAND)
!--------- обработка команд ------------------------
  select case (MakeLong(LOWORD(wParam), 0))
  end select
 case (WM_SYSCOMMAND)
!-------- обработка сообщений системного меню ---------------
  if (wParam == SC_CLOSE) then
   do i=0, 3
    ir = DeleteObject(hImbmp)
   end do
  ir = ReleaseDC(hDlg, hDC)
  ir = ImageList_Destroy(hImList)
  ir = EndDialog (hDlg, 1)
  end if
end select
DialogFunc7 = 0
end function DialogFunc7
!*****************************************************************
```

Для перемещения изображения используются функции **ImageList_BeginDrag()**, **ImageList_DragMove()** и **ImageList_EndDrag()**.

Функция **ImageList_BeginDrag (hImlTrack, iTrack, dxHotspot, dyHotspot)** начинает операцию перемещения, связывая рисунок с текущим положением курсора. Для правильной прорисовки изображения фиксируется положение курсора относительно угла рисунка (так называемое горячее пятно). Параметры включают дескриптор списка изображений **hImlTrack**, индекс перемещаемого рисунка **iTrack** и расположение горячего пятна **dxHotspot**, **dyHotspot**. Обычно прикладная программа устанавливает горячее пятно так, чтобы оно совпало с курсором.

Функция **ImageList_DragMove(x, y)** перемещает рисунок в новое положение. Ее аргументы не нуждаются в пояснениях. Функция **ImageList_EndDrag()** заканчивает перемещение и заставляет Систему прорисовать всю оставшуюся часть экрана.

Прежде чем начать перемещение, необходимо задать начальное положение рисунка. Функция **ImageList_DragEnter (hwndLock, x, y)** задает начальную позицию перемещаемого изображения внутри окна и устанавливает рисунок в эту позицию. Первый аргумент – это дескриптор окна, к которому принадлежит список изображений.

> Внимание: координаты начальной позиции внутри окна задаются относительно левого верхнего угла физического окна, а не рабочей области.

Это означает, что вы самостоятельно должны учесть размеры всех элементов окна (рамки, заголовка, линейки меню и т. п.). Если **hwndLock = NULL**, то все функции перемещения выводят изображение относительно левого верхнего угла экрана.

Функция **ImageList_DragEnter()** блокирует любые модификации окна на все время перемещения. Если это нужно сделать, то вы можете временно скрывать перемещенное изображение, используя функцию **ImageList_DragLeave(hwndLoc)**.

В нашем примере процедура перемещения начинается в ответ на сообщение WM_LBUTTONDOWN. Если курсор находится внутри прямоугольника, ограничивающего изображение, то определяется номер рисунка, вычисляются координаты горячего пятна. Затем инициируется собственно операция перемещения с помощью функции **ImageList_BeginDrag()**. Для того чтобы сообщения мыши были адресованы данному окно (в нашем случае – окну диалога), необходимо использовать функцию **ir = SetCapture(hDlg)**. После того как процесс перемещения будет закончен, необходимо восстановить исходное состояние с помощью функции **ReleaseCapture()**. Если этого не сделать, то вас ожидает масса неприятностей уже после закрытия вашего приложения. Будьте очень внимательными при использовании этой пары функций.

Перемещение осуществляется в ответ на сообщение WM_ MOUSEMOVE. Координаты курсора используются для вычисления положения рисунка.

Все заканчивается, когда диалоговая процедура получает сообщение WM_LBUTTONUP. Процесс перемещения завершается функцией **ImageList_EndDrag()**. Перемещаемое изображение скрывается, а на его место помещается нужный рисунок.

Для того чтобы можно было пользоваться средствами создания и управления списком изображений, надо дополнить файл **commctrlty.f90** следующим фрагментом:

```
!====== IMAGE APIS ==========================================
!MS$IF .NOT. DEFINED (NOIMAGEAPIS)

integer, parameter, public :: CLR_NONE          = #FFFFFFFF
```

```
integer, parameter, public :: CLR_DEFAULT        = #FF000000

!--------- флаги списка изображений ------------------------
integer, parameter, public :: ILC_MASK           = #0001
integer, parameter, public :: ILC_COLOR          = #0000
integer, parameter, public :: ILC_COLORDDB       = #00FE
integer, parameter, public :: ILC_COLOR4         = #0004
integer, parameter, public :: ILC_COLOR8         = #0008
integer, parameter, public :: ILC_COLOR16        = #0010
integer, parameter, public :: ILC_COLOR24        = #0018
integer, parameter, public :: ILC_COLOR32        = #0020

!------------ флаги перерисовки ---------------------
integer, parameter, public :: ILD_NORMAL         = #0000
integer, parameter, public :: ILD_TRANSPARENT    = #0001
integer, parameter, public :: ILD_MASK           = #0010
integer, parameter, public :: ILD_IMAGE          = #0020
integer, parameter, public :: ILD_ROP            = #0040
integer, parameter, public :: ILD_BLEND25        = #0002
integer, parameter, public :: ILD_BLEND50        = #0004
integer, parameter, public :: ILD_OVERLAYMASK    = #0F00
integer, parameter, public :: ILD_SELECTED       = ILD_BLEND50
integer, parameter, public :: ILD_FOCUS          = ILD_BLEND25
integer, parameter, public :: ILD_BLEND          = ILD_BLEND50
integer, parameter, public :: CLR_HILIGHT        = CLR_DEFAULT

integer, parameter, public :: ILCF_MOVE          = #00000000
integer, parameter, public :: ILCF_SWAP          = #00000001

!------------ Типы ----------------------------------------
type T_IMAGELISTDRAWPARAMS
 integer(4) :: cbSize
 integer(4) :: himl
 integer(4) :: i
 integer(4) :: hdcDst
 integer(4) :: x
 integer(4) :: y
 integer(4) :: cx
 integer(4) :: cy
 integer(4) :: xBitmap
 integer(4) :: yBitmap
 integer(4) :: rgbBk
 integer(4) :: rgbFg
 integer(4) :: fStyle
 integer(4) :: dwRop
end type T_IMAGELISTDRAWPARAMS
```

```
type  T_IMAGEINFO
 integer(4)  :: hbmImage
 integer(4)  :: hbmMask
 integer(4)  :: Unused1
 integer(4)  :: Unused2
 type(T_RECT):: rcImage
end type  T_IMAGEINFO

!MS$ENDIF !   NOIMAGEAPIS
```

После запуска исполняемого модуля на экране можно получить изображение, показанное на рис. 20.1.

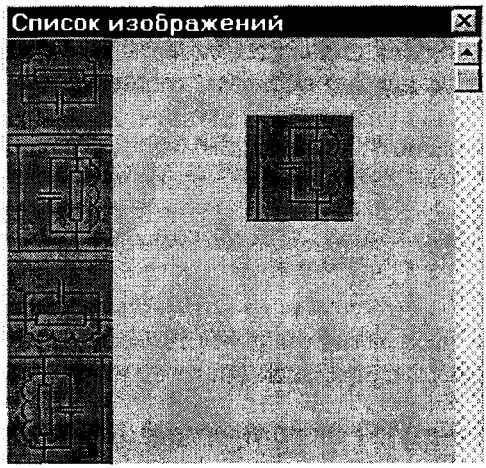

Рис. 20.1. Диалог со списком изображений

УРОК

21. Реестр

Реестр (registry) – это определенная Системой база данных, которая используется ею и прикладными программами для хранения параметров конфигурации.

Вы, наверное, обращали внимание на то, что подавляющее число программных продуктов для Windows сохраняет данные при завершении работы с ними. Это очень удобно, т. к. позволяет начать новый сеанс точно с предшествующего состояния или предшествующей конфигурации.

Для хранения данных, необходимых для выполнения приложения, в WIN32 предусмотрен специально разработанный механизм, который получил название *реестра*. Данные сохраняются в двоичных файлах. Прикладная программа может использовать функции API для управления этими данными.

Объем данных, которые можно сохранять непосредственно в одном элементе реестра, не должен превышать 1 Кбайт. В противном случае данные следует сохранять в отдельном файле, а в реестре – ссылку на него.

21.1. Структура реестра и форма хранения данных

Реестр имеет древовидную иерархическую структуру. Каждый узел в дереве называется *ключом*. Каждый ключ может иметь подключи и конечные элементы данных, называемые *значениями*. При необходимости приложение открывает ключ и использует данные, которые он содержит. Иногда ключ содержит все необходимые данные; в других случаях прикладная программа открывает ключ и использует значения, связанные с данным ключом. Ключ может иметь любое число значений с произвольным форматом.

Каждый ключ имеет имя, состоящее из одного или большего количества символов. Имена ключей не могут включать пробел, левую наклонную черту (\) или символы "*", "?". Имена, начинающиеся с многоточия, зарезервированы Системой. Имя каждого подключа уникально в пределах ключа, которому он подчинен.

Хотя число технических ограничений на формат и размеры данных незначительно, все же следует придерживаться некоторых практических принципов, для того чтобы эффективность Системы оставалась на должном уровне. Прикладная программа может хранить в реестре конфигурацию и данные инициализации. Что касается других типов данных, то они должны быть сохранены иным способом.

Как уже говорилось выше, данные объемом более 1– 2 Кбайт должны храниться в файле, на который ссылается ключ. Никогда не следует использовать реестр для хранения исполняемого двоичного кода.

Для хранения значений требуется намного меньше места, чем для отдельного ключа. Поэтому рекомендуется для экономии ресурсов ПК группировать однородные данные в структуры. Представление данных в двоичной форме позволяет прикладной программе сохранять данные в одном формате. Это удобнее и экономнее, т. к. иначе может возникнуть необходимость объединения нескольких несовместимых типов.

При инициализации Системы создаются, по крайней мере, четыре ключа: HKEY_LOCAL_MACHINE, HKEY_USERS, HKEY_CLASSES_ROOT, HKEY_CURRENT_USER. Дескрипторы этих ключей доступны всем модификациям реестра.

Предопределенные ключи помогают прикладной программе ориентироваться в реестре и делают возможным разработку инструментальных средств, которые позволяют Системе управлять различными категориями данных. Прикладные программы, добавляющие данные в реестр, должны всегда работать в рамках предопределенных ключей так, чтобы инструментальные средства могли находить и использовать эти новые данные.

Ключ HKEY_LOCAL_MACHINE содержит сведения о физическом состоянии компьютера, включая данные относительно типа шины, памяти и установленных аппаратных средств и программного обеспечения.

Подключи, подчиненные ключу HKEY_CLASSES_ROOT, определяют типы (или классы) файлов, связанных с этими типами. Свойства классов задаются программистом. Обычно эти данные применяются прикладными программами, использующими систему Windows.

Ключ HKEY_USERS определяет заданную по умолчанию конфигурацию для новых пользователей на локальном компьютере и конфигурации текущего пользователя.

Подключи, подчиненные ключу HKEY_CURRENT_USER, задают установки, сделанные текущим пользователем и касающиеся окружения, данных о принтерах, сетевых подключениях и т. п. Кроме того, здесь же хранятся установки, сделанные конкретным приложением.

Использование ключей HKEY_CURRENT_USER, HKEY_LOCAL_MACHINE и HKEY_USERS зависит от конкретной реализации реестра. Для некоторых модификаций Системы в дополнение к перечисленным ключам могут быть добавлены и другие, например ключи с дескрипторами HKEY_PERFORMANCE_DATA, HKEY_CURRENT_CONFIG, HKEY_DYN_DATA. Численные значения дескрипторов предопределенных ключей приведены в табл. 21.1.

Таблица 21.1. Численные значения предопределенных ключей

Ключ	*Значение*
HKEY_CLASSES_ROOT	#80000000
HKEY_CURRENT_USER	#80000001
HKEY_LOCAL_MACHINE	#80000002
HKEY_USERS	#80000003
HKEY_PERFORMANCE_DATA	#80000004

До размещения данных в реестре необходимо решить, к какой из двух категорий они относятся: машинным или пользовательским.

Когда прикладная программа устанавливается, она должна записать специфические машинные данные под ключом HKEY_LOCAL_MACHINE. В частности, сюда заносятся названия компании, имя программы и номера версий так, как показано в следующем примере:
HKEY_LOCAL_MACHINE\Software\MyCompany\M yProduct\1.0

Определенные пользователем данные записываются под ключом HKEY_CURRENT_USER, например следующим образом:

HKEY_CURRENT_USER\Software\MyCompany\My Product\1.0\ data

21.2. Взаимодействие с реестром

Для управления реестром предусмотрено несколько функций WIN32 API. Описание некоторых из них приведены ниже.

Пара функций **RegOpenKey()** и **RegOpenKeyEx()** позволяет открыть ключ, а функции **RegCreateKey()** и **RegCreateKeyEx()** – создать новый ключ.

Функция **RegCreateKeyEx (hKey, lpSubKey, Reserved, lpClass, dwOptions, samDesired, lpSecurityAttributes, phkResult, lpdwDisposition)** создает заданный ключ, а если он уже существует, открывает его. Первый параметр, **hKey** задает дескриптор ключа, а второй, **lpSubKey** – его имя.

Аргумент **lpClass** это строка с именем класса. Пятый параметр, **dwOptions** задает опции ключа. Определены два значения этого аргумента: REG_ OPTION_NON_VOLATILE = #00000000 и REG_OPTION_VOLATILE = = #00000001. Второе значение используется только Windows NT.

Следующий параметр, **samDesired** задает маску доступа к ключу. Флаги доступа приведены в табл. 21.2.

Таблица 21.2. Флаги доступа к ключу

Флаг	*Значение*	*Описание*
KEY_QUERY_VALUE	#00000001	Право запрашивать данные подключей
KEY_SET_VALUE	#00000002	Право устанавливать данные подключей
KEY_CREATE_SUB_ KEY	#00000004	Право создавать подключи
KEY_ENUMERATE_ SUB_KEYS	#00000008	Право перебирать подключи
KEY_NOTIFY	#00000010	Право изменять нотификацию
KEY_CREATE_LINK	#00000020	Право создавать символическую связь
KEY_READ		IAND(IOR(STANDARD_RIGHTS_R EAD, IOR (KEY_QUERY_VALUE, IOR (KEY_ENUMERATE_SUB_KEYS, KEY_NOTIFY))), -1048577) !NOT(SYNCHRONIZE
KEY_WRITE		IAND(IOR(STANDARD_RIGHTS_W RITE. IOR (KEY_SET_VALUE, KEY_ CREATE_SUB_KEY)), -1048577)) !NOT(SYNCHRONIZE
KEY_EXECUTE		(IAND(KEY_READ, -1048577)) !NOT(SYNCHRONIZE)))
KEY_ALL_ACCESS		IAND(IOR(STANDARD_RIGHTS_ ALL, IOR (KEY_QUERY_VALUE, IOR(KEY_SET_VALUE. IOR(KEY_CREATE_SUB_KEY, IOR (KEY_ENUMERATE_SUB_KEYS, IOR(KEY_NOTIFY. KEY_CREATE_LINK))))))), -1048577)) !NOT(SYNCHRONIZE)

Параметр **lpSecurityAttributes** – это структура типа T_SECURITY_ ATTRIBUTES с атрибутами защиты, которые поддерживаются только Windows NT.

Аргумент **phkResult** – адрес дескриптора ключа, а **lpdwDisposition** – адрес переменной, описывающей информацию о том, что произошло с ключом. Если ключ был создан, то **lpdwDisposition = REG_CREATED_ NEW_ KEY = #00000001**. Если же ключ уже существовал, то **lpdw Disposition = REG_OPENED_EXISTING_KEY = #00000002**.

Прикладная программа закрывает ключ с помощью функции **RegClose Key()**. Эта функция не обязательно заносит данные в реестр перед возвратом. При записи данных может потребоваться несколько секунд для их переноса из оперативной памяти на жесткий диск. Прикладная программа может сама немедленно записать данные реестра на жесткий диск, используя функцию **RegFlushKey()**. Эта функция использует значительных ресурсы Системы и должна вызываться только в случае крайней необходимости.

Запись данных в реестр осуществляют функции **RegSetValue()** или **RegSetValueEx()**. Первая работает только с текстовыми строками (значения, имеющие тип **REG_SZ**) и поэтому является устаревшей. Для новых приложений рекомендуется использовать **RegSetValueEx (hKey, lpValue Name, Reserved, dwType, pData, cbData)**, которая может записывать значения любого типа данных. Первый параметр, **hKey** – это дескриптор ключа, следующий, **lpValueName** – строка с именем данных. Третий аргумент, **dwType** устанавливает тип данных. Возможные его значения приведены в табл. 21.3.

Таблица 21.3. Типы сохраняемой информации

Параметр	*Значение*	*Описание*
REG_NONE	0	Тип данных не установлен
REG_SZ	1	Строка с нулевым символом в конце
REG_EXPAND_SZ	2	Строка со ссылкой на окружение
REG_BINARY	3	Бинарные данные
REG_DWORD	4	Двойное слово (32 бита)
REG_DWORD_LITTLE_ENDIAN	4	То же

Параметр	*Значение*	*Описание*
REG_DWORD_BIG_ENDIAN	5	То же, но наиболее значащий – младший байт
REG_LINK	6	Символическая связь
REG_MULTI_SZ	7	Массив строк с двумя нулевыми символами в конце всех строк
REG_RESOURCE_LIST	8	Список драйверов
REG_FULL_RESOURCE_DESCRIPTOR	9	Список ресурсов в виде частей аппаратуры
REG_RESOURCE_REQUIREMENTS_LIST	10	

Последние два параметра задают буфер данных: **lpData** – адрес буфера данных, **cbData** – его размер.

Данные удаляются функцией **RegDeleteValue (hKey, lpValueName)**. Функция удаляет поименованное значение из заданного ключа. Параметры: **hKey** – дескриптор открытого ключа, **lpValueName** – строка с именем данных. Функция возвращает ERROR_SUCCESS или собщение об ошибке. Ключ должен быть открыт с доступом KEY_SET_VALUE.

Можно удалить ключ полностью, используя функцию **RegDeleteKey (hKey, lpSubKey)**.

При всех манипуляциях с ключами и данными необходимо добраться до них. Для этого приходится перебирать элементы дерева реестра с помощью функции **RegEnumKey()** или **RegEnumKeyEx()**. Предпочтение следует отдать последней.

Функция **RegEnumKeyEx (hKey, dwIndex, lpName, pcbName, lpReserved, lpClass, lpcbClass, lpftLastWriteTime)** перебирает подключи открытого ключа. В отличие от функции **RegEnumKey()** передает имя класса подключа, а также дату и время последнего изменения. Первый параметр, **hKey** – дескриптор открытого ключа. Далее следуют: **dwIndex** – индекс подключа, **lpName** – строка с именем подключа, **lpcbName** – длина строки **lpName**, **lpClass** – строка с именем класса, **lpcbClass** – длина строки, **lpftLastWriteTime** – структура типа T_FILETIME, которая описана следующим образом:

```
type  T_FILETIME
   integer dwLowDateTime
   integer dwHighDateTime
end type  T_FILETIME

type(T_FILETIME), pointer :: null_filetime
```

Структура передает значение с 64 битами, которое равно числу интервалов по 100 нс от 1 января, 1601 до текущей даты.

Функция возвращает ERROR_SUCCESS или сообщение об ошибке. Подключ должен быть открыт с доступом KEY_ENUMERATE_SUB_KEYS.

После того как нужный ключ найден, можно извлечь данные, используя функцию **RegEnumValue (hKey, dwIndex, lpValueName, lpcbValueName, lpReserved, lpType, lpData, lpcbData)**. Эта функция перебирает значения для открытого ключа и копирует имя и значение с заданным индексом в буфер.

Первый аргумент, **hKey** – дескриптор открытого ключа. Далее следуют индекс значения и строка с его именем. Длина строки указывается четвертым параметром, **lpcbValueName**. Параметр **lpType** задает формат значений(см. табл. 21.1). Адрес буфера данных и размер буфера устанавливаются двумя последними аргументами функции. Функция возвращает ERROR_SUCCESS или сообщение об ошибке. Подключ должен быть открыт с доступом KEY_QUERY_VALUE.

При обращении к ключам и значениям требуются детализированные данные об их свойствах. Прикладная программа получает эти сведения с помощью функции **RegQueryInfoKey()**.

Функция **RegQueryInfoKey (hKey, lpClass, lpcbClass, lpReserved, lpcSubKeys, lpcbMaxSubKeyLen, lpcbMaxClassLen, pcValues, lpcbMaxValueNameLen, lpcbMaxValueLen, lpcbSecurityDescriptor, lpftLastWriteTime)** передает данные о заданном ключе. Первый параметр, как обычно, – это дескриптор ключа. Имя класса указывается в С-строке **lpClass**, длина которой задается параметром **lpcbClass**. Пятый параметр, **lpcSubKeys** – это адрес переменной, которая передает число подключей. Далее следуют: **lpcbMaxSubKeyLen** – адрес переменной, указывающей максимальную длину имени подключа, **lpcbMaxClassLen** – адрес переменной, указывающей максимальную длину имени класса, **lpcValues** – адрес переменной, указывающей число значений, **lpcbMaxValueNameLen** адрес переменной, указывающей максимальную длину имени данных, **lpcbMaxValueLen** – адрес переменной, указывающей максимальный раз-

мер данных, **lpcbSecurityDescriptor** – адрес переменной, описывающей атрибуты защиты, **lpftLastWriteTime** – структура типа T_FILETIME, которая была описана выше.

Ключ должен быть открыт с доступом KEY_QUERY_VALUE.

Функции **RegQueryValue()** и **RegQueryValueEx()** используются для получения данных открытого в данный момент ключа. Прикладная программа обычно вызывает первую функцию, чтобы определить имена значений, а затем вторую, чтобы получить данные значения с заданным именем.

Прикладные программы могут сохранять часть реестра в файле и затем загружать его содержимое снова в реестр. Файл полезен в том случае, когда объем данных велик или когда данные загружаются временно.

21.3. Пример диалога, взаимодействующего с реестром

Для демонстрации работы с реестром создадим диалог **РЕЕСТР**. Добавим новый ресурс в проект, используя средства MDS. Эту процедуру мы уже неоднократно выполняли. В окно диалога поместим окно просмотра деревьев (IDC_TREE1), которое будет отображать и обслуживать древовидную структуру реестра. Сведения о движении по реестру будем отображать в статическом окне с идентификатором IDC_STATIC1, а хранящуюся информацию – в окне списка IDC_LIST1.

Обслуживание диалога будет производить диалоговая процедура **DialogFunc9()**. Эта процедура вызывается в теле оконной функции **MainWndProc()** по команде меню IDM_DIALOG9 обычным образом:

```
case (IDM_DIALOG9)
  ir= DialogBox(hInst, LOC("PEECTP"C), ghwndMain, LOC(DialogFunc9))
```

Текст диалоговой функции приведен ниже.

```
!****************************************************************
integer(4) function DialogFunc9(hDlg, message, wParam, lParam)
!MS$ ATTRIBUTES STDCALL, ALIAS : '_DialogFunc9@16' :: DialogFunc9
!
integer(4)        :: hDlg, message, wParam, lParam

integer(4)        :: ir
integer(4)        :: hItm
integer(4)        :: hKeyRoot
integer(4)        :: hParentItem
```

```
character(256)    :: Path
character(128)    :: Bfr
character(128)    :: ClName

character(22), dimension(0:6):: szKeys=          &
            (/"HKEY_CLASS_BOOT"C,                &
             &"HKEY_CURREN_USER"C,               &
             &"HKEY_LOCAL_MCHINE"C,              &
             &"HKEY_USERS"C,                     &
             &"HKEY_PERFORMANCE_DATA"C,          &
             &"HKEY_CURRENT_CONFIG"C,            &
             &"HKEY_DYN_DATA"C/)

type (T_TV_INSERTSTRUCT)    :: Tvs

type (T_NM_TREEVIEW)        :: TView
POINTER(ip, TView)

type  (T_FILETIME)          :: LastWriteTime

hDialog = hDlg
lParam = lParam

select case (message)
 case (WM_INITDIALOG)
 !------------ Разметка дерева ---------------------
  hTree  = GetDlgItem(hDlg, IDC_TREE1)

  Tvs%hInsertAfter =TVI_LAST
  Tvi%mask= TVIF_TEXT
  Bfr = "Реестр"C
  Tvi%iImage = -1
  Tvi%cchTextMax = 128
  Tvi%pszText  = LOC(Bfr)
  Tvs%hParent  = TVI_ROOT
  Tvs%item     = Tvi
  hParentItem  = SendMessage(hTree, TVM_INSERTITEM, 0, LOC(Tvs))
  hKeyRoot = 0
  ir = FillTree()
   DialogFunc9 = 1
   return
  case (WM_NOTIFY)
  ip = TRANSFER(lParam, ip)
  select case(TView%hdr%code)
```

```
  case(TVN_ITEMEXPANDED)
  case(TVN_SELCHANGED)
   Tvi%hItem=TView%itemNew%hItem
   ir = SendMessage(hTree, TVM_GETITEM, 0, LOC(Tvi))
   ir = SetDlgItemText(hDlg, IDC_STATIC1, "Перешли: "//Bfr)
   ir = SendDlgItemMessage(hDlg, IDC_LIST1, &
                           LB_RESETCONTENT, 0, 0)
  case(NM_DBLCLK)
   Tvi%hItem = SendMessage(hTree, TVM_GETNEXTITEM, TVGN_CARET, 0)
   ir = SendMessage(hTree, TVM_GETITEM, 0, LOC(Tvi))
   Tvi%hItem = SendMessage(hTree, TVM_GETNEXTITEM, TVGN_PARENT,
Tvi%hItem)
    ir = SendMessage(hTree, TVM_GETITEM, 0, LOC(Tvi))
   if(Bfr(1:INDEX(Bfr, char(0)))=="Реестр"C) then
    ir = MessageBox(ghwndMain, "Вершина дерева"C, "Сообщение"C,
MB_OK)
   else
    ir = KeyPath()
   ir = DataKey()
   end if
 end select
case (WM_SYSCOMMAND)
!-------- обработка сообщений системного меню ----------
 if (wParam == SC_CLOSE) then
    ir = EndDialog(hDlg, wParam)
 end if
end select
DialogFunc9 = 0
!_____________________________________________________
contains
!*****************************************************
integer(4) function FillTree()
!_____________________________________________________
integer(4)              :: i, j

integer(4)              :: hNewParentItem
integer(4)              :: hKey, ClassLen, SubKeys, MaxSubKey, &
                           MaxClass, Values, MaxValueName, &
                           MaxValueData, SecDesc

character*(MAX_PATH)    :: ClassName

ClassLen  = MAX_PATH
 ir =SetCursor(LoadCursor(NULL, IDC_WAIT))
 do i=0, 6
```

```
  hKey=HKEY_CLASSES_ROOT+i
  ir = RegQueryInfoKey (hKey     . & ! Key handle.                    &
        ClassName                . & ! Buffer for class name.         &
        LOC(ClassLen)            . & ! Length of class string.        &
        NULL                     . & ! Reserved.                      &
        LOC(SubKeys)             . & ! Number of sub keys.            &
        LOC(MaxSubKey)           . & ! Longest sub key size.          &
        LOC(MaxClass)            . & ! Longest class string.          &
        LOC(Values)              . & ! Number of values for this key. &
        LOC(MaxValueName)        . & ! Longest Value name.            &
        LOC(MaxValueData)        . & ! Longest Value data.            &
        LOC(SecDesc)             . & ! Security descriptor.           &
        LastWriteTime)               ! Last write time.
  if(ERROR_SUCCESS == ir)then
   Bfr = szKeys(i)
    Tvi%pszText  = LOC(Bfr)
   Tvs%hParent   = hParentItem
    Tvs%item = Tvi
   hNewParentItem   = SendMessage(hTree, TVM_INSERTITEM, 0, LOC(Tvs))
    ir = FillBranch(hKey, SubKeys, hNewParentItem)
  end if
 end do
ir =SetCursor(LoadCursor(NULL, IDC_ARROW))
FillTree =1
end function FillTree
!*******************************************************
integer(4) function FillBranch(hK, SK, hPr)
!_____________________________________________________
integer(4)                  :: hK, SK, hPr, j

character(MAX_PATH)  :: ClName

    if(SK==0) then
     return
    else
     do j=0, SK-1
    ir = RegEnumKey(hK, j, ClName, MAX_PATH)
      call FillSubBranch(hK, ClName, hPr)
   end do
    end if
FillBranch =1
!-------------------------------------------------
end function FillBranch
```

```
!*****************************************************
recursive subroutine FillSubBranch(hK, ClNm, hPr)
!_____________________________________________________

integer(4)              :: hK
integer(4)              :: hNewK
integer(4)              :: hPr
integer(4)              :: SbK
integer(4)              :: hNewPr
integer(4)              :: ClL

character(MAX_PATH) :: CN, ClNm, ClNmNew

    ClL=MAX_PATH
    Bfr = ClNm
   Tvs%item    = Tvi
   Tvs%hParent  = hPr
   hNewPr  = SendMessage(hTree, TVM_INSERTITEM, 0, LOC(Tvs))

    ir = RegOpenKeyEx(hK, ClNm, 0, &
       IOR(KEY_ENUMERATE_SUB_KEYS, &
           IOR(KEY_EXECUTE, KEY_QUERY_VALUE)), &
           LOC(hNewK))

    ir = RegQueryInfoKey (hNewK . &
         CN          . &
         LOC(ClL)    . &
         NULL        . &
         LOC(SbK)    . &
         NULL, NULL, NULL, NULL, NULL, NULL, LastWriteTime)

    if(SbK>0) then
     do    j=0, SbK-1
    ir = RegEnumKey(hNewK, j, ClNmNew, MAX_PATH)
    call FillSubBranch(hNewK, ClNmNew, hNewPr)
     end do
    end if

    ir = RegCloseKey(hNewK)

end subroutine FillSubBranch
!*****************************************************
integer(4) function KeyPath()
!_____________________________________________________

integer(4)          :: in, il
```

```
   Tvi%hItem = SendMessage(hTree, TVM_GETNEXTITEM, TVGN_CARET, 0)
   ir = SendMessage(hTree, TVM_GETITEM, 0, LOC(Tvi))
    il = INDEX(Bfr, char(0))
    Path(1:il)=Bfr(1:il)
    in =il
    do
   Tvi%hItem = SendMessage(hTree, TVM_GETNEXTITEM, &
                       TVGN_PARENT, Tvi%hItem)
    ir = SendMessage(hTree, TVM_GETITEM, 0, LOC(Tvi))
    il = INDEX(Bfr, char(0))
    if(Bfr(1:4)=="HKEY") then
     do i=0, 6
        if(bfr(1:il-1)==szKeys(i)(1:il-1)) then
       hKeyRoot=HKEY_CLASSES_ROOT+i
         exit
        end if
      end do
      exit
     else
      Bfr(il:il) ='\'
      Path(il+1:il+in) =Path(1:in)
      Path(1:il)= Bfr(1:il)
      in=il+in
     end if
    end do
   Tvi%hItem = SendMessage(hTree, TVM_GETNEXTITEM, TVGN_CARET, 0)
   ir = SendMessage(hTree, TVM_GETITEM, 0, LOC(Tvi))
    KeyPath =1
end function KeyPath
!*****************************************************
integer(4) function DataKey()
!_____________________________________________________

integer(4)          :: hDataKey
integer(4)          :: dwLBIndex
integer(4)          :: cbValueName
integer(4)          :: dwType
integer(4)          :: dwcClassLen
integer(4)          :: dwcSubKeys
integer(4)          :: dwcMaxSubKey
integer(4)          :: dwcMaxClass
integer(4)          :: dwcValues
integer(4)          :: dwcMaxValueName
integer(4)          :: dwcMaxValueData
```

```
integer(4)          :: dwcSecDesc
integer(4)          :: cbData

character(MAX_VALUE_NAME) :: ValueName
character(MAX_PATH)       :: ClassName
character(1024)           :: bData
character(1024)           :: buffer

  cbValueName = MAX_VALUE_NAME
  dwcClassLen = MAX_PATH

  ir = SendDlgItemMessage(hDialog, IDC_LIST1, &
                  LB_RESETCONTENT, 0, 0)

  ir = RegOpenKeyEx (hKeyRoot                    .&
                  Path                           .&
                  0                              .&
                  IOR(KEY_ENUMERATE_SUB_KEYS,    &
                  IOR(KEY_EXECUTE                .&
                  KEY_QUERY_VALUE))              .&
                  LOC(hDataKey))

  ir =  RegQueryInfoKey (hDataKey                .&
                   ClassName                     .&
                   LOC(dwcClassLen)              .&
                   NULL                          .&
                   LOC(dwcSubKeys)               .&
                   LOC(dwcMaxSubKey)             .&
                   LOC(dwcMaxClass)              .&
                   LOC(dwcValues)                .&
                   LOC(dwcMaxValueName),         &
                   LOC(dwcMaxValueData),         &
                   LOC(dwcSecDesc)               .&
                   LastWriteTime)
  cbData=dwcMaxValueData

  do dwLBIndex = 0, dwcValues
    ir = RegEnumValue (hDataKey      . &
                  dwLBIndex          . &
                  ValueName          . &
                  LOC(cbValueName),    &
                  NULL               . &
                  LOC(dwType)        . &
                  LOC(bData)         . &
                  LOC(cbData))
```

```
      if(ir==ERROR_SUCCESS) then
       buffer = ValueName(:cbValueName)//' ' //bData(1:cbData)
      ir = SendDlgItemMessage(hDialog, IDC_LIST1, &
                  LB_ADDSTRING, 0, LOC(buffer))
      else if(ir == ERROR_NO_MORE_ITEMS) then
      exit
      end if

  end do

  DataKey =1
end function DataKey
!*****************************************************
end function DialogFunc9
!*****************************************************
```

Приведенный фрагмент приложения создает древовидную структуру реестра. При ее заполнении используется рекурсивная процедура **recursive subroutine FillSubBranch(hK, ClNm, hPr)**, которая перебирает ключи и значения, двигаясь по стволу и ветвям дерева. Это необходимо потому, что строение дерева реестра заранее неизвестно.

Для заполнения реестра приложение обращается к диску, и поэтому этот процесс требует заметного времени. В процедуре **integer(4) function FillTree()** демонстрируется возможность замены курсора на время ожидания. Для этого используется функция API **SetCursor(LoadCursor(NULL, IDC_WAIT))**, которая загружает системный курсор в виде песочных часов. В Системе предусмотрено несколько стандартных курсоров:

IDC_APPSTARTING– стандартная стрелка с маленькими песочными часами,
IDC_ARROW – стандартная стрелка,
IDC_CROSS – перекрестие,
IDC_IBEAM – символ I,
IDC_NO – перечеркнутый круг,
IDC_SIZEALL – перекрестие со стрелками,
IDC_SIZENESW – двойная стрелка, указывающая северо-восток и юго-запад,
IDC_SIZENS – двойная стрелка, указывающая север и юг,
IDC_SIZENWSE – двойная стрелка, указывающая северо-запад и юго-восток,
IDC_SIZEWE – двойная стрелка, указывающая запад и восток,
IDC_UPARROW – вертикальная стрелка,
IDC_WAIT – песочные часы.

После завершения процедуры заполнения дерева стандартный курсор должен быть восстановлен.

Работа с реестром чрезвычайно деликатна. Вы можете получить массу неприятностей после бездумного изменения реестра. Поэтому в примере демонстрируется только процесс извлечения данных. Для этого служит функция **integer(4) function DataKey()**.

Все необходимые константы и интерфейсы функций заданы в файлах **msfwin.f90** и **msfwinty.f90**. Поэтому никаких дополнений, кроме приведенных выше, в наши файлы делать не придется.

Остается создать исполнительный файл и запустить его. Если ошибок в тексте нет, то на экране появится то, что показано на рис. 21.1.

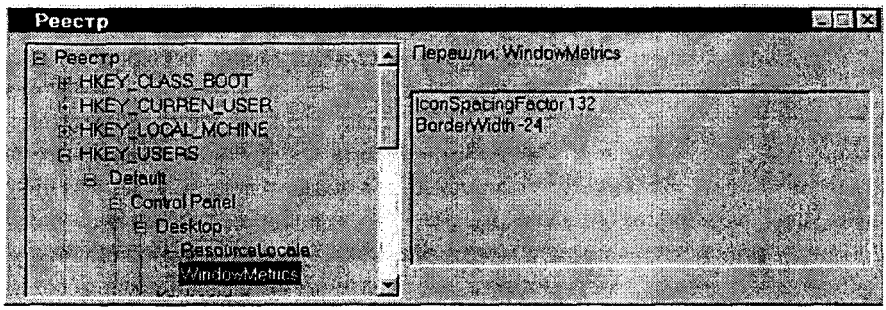

Рис. 21.1. Диалог с реестром

Приложения

Содержат некоторые функции WIN32 API для создания приложений, описанные в файле Msfwin.f90.

П-1. Функции для создания окна и управления им

integer(4) function CreateWindow (lpszClassName, lpszWinowName, dwStyle, x, y, nWidth, nHeight, hWndParent, hMenu, hInstance, lpParam)

Функция создает окно. Она определяет класс окна, заголовок окна, стиль окна, его позицию и размер. Функция также определяет родителя окна или владельца, если он есть, и меню окна. Параметры: **lpszClassName** -С строка с именем класса; **lpszWindowName** – С-строка с именем окна; **dwStyle** – стиль окна; **x, y** – координаты верхнего левого угла; **nWidth, nHeight** – размеры окна; **hWndParent** – дескриптор родительского окна; **hMenu** – дескриптор главного меню; **hInstance** – дескриптор приложения; **lpParam** – указатель на структуру с дополнительной информацией.

integer(4) function GetClassName (hWnd, lpClassName, nMaxCount)

Функция возвращает имя класса, к которому принадлежит заданное окно. Параметры: **hWnd** – дескриптор окна, **lpClassName** – строка с именем класса, **nMaxCount** – длина строки. Возвращает число прочитанных символов.

integer(4) function RegisterClass (lpWndClass)

Функция регистрирует класс окна для последующего использования в обращениях к функциям **CreateWindow** и **CreateWindowEx**. Параметр lpWNDCLASS – структура типа T_WNDCLASS.

logical(4) function ShowWindow (hWnd, nCmdShow)

Функция отображает окно на экране. Параметры: **hWnd** – дескриптор окна, **nCMDShow** – способ отображения.

logical(4) function UpdateWindow (hWnd)

Функция обновляет рабочую область заданного окна, посылая сообщение WM_PAINT, если эта область не пуста. Функция посылает сообщение WM_PAINT непосредственно оконной процедуре, совершая обход пооче-

редно всех прикладных программ. Если область пуста, то сообщение не посылается. Параметр **hWnd** – дескриптор окна.

logical(4) function MoveWindow (hWnd, X, Y, nWidth, nHeight, bRepaint)

Функция изменяет позицию и размеры заданного окна. Для главного окна позиция и размеры задаются относительно левого верхнего угла экрана, а для дочернего – относительно левого верхнего угла рабочей области родительского окна. Параметры: **hWnd** – дескриптор окна; **x**, **y** – координаты левого верхнего угла; **nWidth**, **nHeight** – ширина и высота; **bRepaint** – флаг перерисовки. Если параметр **bRepaint = .TRUE.**, то Система посылает сообщение WM_PAINT процедуре окна немедленно после перемещения окна. Иначе сообщение WM_PAINT помещается в очередь сообщений.

logical(4) function SetWindowPos (hWnd, hWndInsertAfter, x, y, cx, cy, uFlags)

Функция изменяет размер, позицию и порядок Z дочернего окна. Параметры: **hWnd** – дескриптор окна; **hWndInsertAfter** – дескриптор предшествующего окна; **x, y, cx, cy** – координаты и размеры окна; **uFlags** – флаг, задающий стиль отображения окна.

logical(4) function IsWindowVisible (hWnd)

Функция определяет состояние заданного окна. Параметр **hWnd** – дескриптор окна. Если окно видимо на экране (стиль WS_VISIBLE), то возвращается значение **TRUE**, даже если окно полностью закрыто другими окнами.

integer(4) function GetSystemMetrics (nIndex)

Функция передает метрику (размеры элементов дисплея в пикселах) и установки Системы. Аргумент **nIndex** задает, какой именно параметр будет возвращен функцией. Все **nIndex** имеют префикс SM_.

logical(4) function GetMessage (lpMsg, hWnd, wMsgFilterMin, wMsgFilterMax)

Функция извлекает сообщение из очереди и помещает его в структуру **lpMSG**. Параметры: **lpMSG** – структура типа T_MSG, **hWnd** – дескриптор окна, **wMSGFilterMin** и **wMSGFilterMax** – пределы значений обрабатываемых сообщений.

integer(4) function GetWindowLong (hWnd, nIndex)

Функция возвращает информацию о заданном окне. Параметры: **hWnd** – дескриптор окна, **nIndex** – задает, какая именно информация будет возвращена.

integer(4) function SetWindowLong (hWnd, nIndex, dwNewLong)

Функция изменяет атрибут заданного окна. Параметры: **hWnd** – дескриптор окна, **nIndex** – задает, какая именно информация будет изменена, **dwNewLong** – новое значение.

logical(4) function IsWindowEnabled (hWnd)

Функция устанавливает, доступно ли окно с дескриптором **hWnd** клавиатуре и мыши.

integer(4) function ScrollWindowEx (hWnd, dx, dy, prcScroll, prcClip, hrgnUpdate, prcUpdate, flags)

Функция прокручивает содержимое рабочей области окна. Параметры: **hWnd** – дескриптор окна, **dx** и **dy** – величина прокрутки, **prcScroll** – структура типа T_RECT с координатами перемещаемого прямоугольника, **prcClip** – структура типа T_RECT с координатами области, внутри которой проводится прокрутка изображения **prcClip**, **hrgnUpdate** – дескриптор области, испорченной прокруткой, **prcUpdate** – структура типа T_RECT с координатами области, испорченной прокруткой, **flags** – флаги управления с префиксом SW_.

integer(4) function SetScrollPos (hWnd, nBar, nPos, bRedraw)

Функция устанавливает позицию заданной линейки прокрутки. Параметры: **hWnd** – дескриптор окна, **nBar** – флаг линейки прокрутки, **nPos** – новая позиция, **bRedraw** – флаг перерисовки. Возвращает предыдущую позицию.

integer(4) function GetScrollPos (hWnd, nBar)

Функция возвращает текущую позицию линейки прокрутки. Численное значение позиции зависит от установленного диапазона прокрутки. Параметры: **hWnd** – дескриптор окна; **nBar** – флаг, определяющий, какая линейка будет исследоваться.

logical(4) function SetScrollRange (hWnd, nBar, nMinPos, nMaxPos, bRedraw)

Функция устанавливает границы диапазона прокрутки слайдера. Параметры: **hWnd** – дескриптор окна, **nBar** – флаг линейки прокрутки, **nMinPos** и **nMaxPos** – границы диапазона прокрутки, **bRedraw** – флаг перерисовки.

logical(4) function GetScrollRange (hWnd, nBar, lpMinPos, lpMaxPos)

Функция возвращает границы диапазона прокрутки слайдера. Параметры: **hWnd** – дескриптор окна, **nBar** – флаг линейки прокрутки, **lpMinPos**

и **lpMaxPos** – адреса переменных, которые принимают границы диапазона прокрутки.

logical(4) function ShowScrollBar (hWnd, wBar, bShow)

Функция показывает или скрывает заданную линейку прокрутки. Параметры: **hWnd** – дескриптор окна, **wBar** – флаг линейки прокрутки, **bShow** – флаг операции.

logical(4) function EnableScrollBar (hWnd, wSBflags, wArrows)

Функция делает линейку доступной или недоступной. Параметры: **hWnd** – дескриптор окна, **wSBflags** – флаг линейки прокрутки, **wArrows** – флаг блокировки стрелки.

logical(4) function TranslateMessage (lpMsg)

Функция преобразует виртуальные коды клавиш в символьные клавиатурные сообщения. Параметры: **lpMSG** – структура типа T_MSG. Функция генерирует сообщение WM_CHAR только для клавиш, которые отображаются символами ASCII.

logical(4) function DispatchMessage (lpMsg)

Функция посылает сообщение оконной процедуре. Обычно используется для того, чтобы послать сообщение, извлеченное функцией **GetMessage**. Параметр **lpMSG** – структура типа T_MSG.

П-2. Функции оконной процедуры

subroutine PostQuitMessage (nExitCode)

Подпрограмма сообщает Системе о намерении завершить поток. Обычно используется в ответ на сообщение WM_DESTROY. Параметр **nExitCode** задает код завершения прикладной программы. Это значение используется как параметр **wParam** сообщения WM_QUIT. Функция **PostQuitMessage** ставит сообщение WM_QUIT в очередь сообщений немедленно. Когда поток получает сообщение WM_QUIT, он должен прекратить обработку всех сообщений и передать управление Системе.

integer(4) function DefWindowProc (hWnd, Msg, wParam, lParam)

Функция возвращает все необработанные процедурой окна сообщения для обработки по умолчанию. Функция гарантирует, что каждое сообщение будет обязательно обработано. Параметры: **hWnd** – дескриптор окна, **MSG** – собственно сообщение, **wParam** и **lParam** – дополнительная информация, которая зависит от содержания сообщения.

logical(4) function EnumChildWindows (hWndParent, lpEnumFunc, lParam)

Функция перебирает дочерние окна, которые принадлежат заданному родительскому окну, и передает дескриптор функции обработки, определенной прикладной программой. Параметры: **hWndParent** – дескриптор родительского окна, **lpEnumFunc** – адрес функции обработки, **lParam** – определенное программой значение.

integer function RegisterWindowMessage (lpString)

Функция создает новое сообщение окна, которое гарантировано будет уникальным для Системы. Возвращенное значение сообщения можно использовать при вызове функции **SendMessage()** или **PostMessage()**. Параметр **lpString** – С-строка с сообщением, которое будет зарегистрировано. Если сообщение успешно зарегистрировано, то значение возврата – это идентификатор сообщения в диапазоне от #C000 до #FFFF. Иначе значение возврата равно нулю.

Функция обычно используется для создания идентификаторов сообщения при наличии связи между двумя сотрудничающими прикладными программами.

integer(4) function SendMessage (hWnd, Msg, wParam, lParam)

Функция посылает сообщение MSG окну или окнам. Функция обращается к оконной функции заданного окна и не возвращает управления до тех пор, пока сообщение не будет обработано. Параметры: **hWnd** – дескриптор окна, **MSG** – собственно сообщение, **wParam** и **lParam** – дополнительная информация, которая зависит от содержания сообщения. Значение возврата определяется результатом обработки сообщения и зависит от посланного сообщения.

integer(4) function GetCapture()

Функция передает дескриптор окна (если оно существует), которое получает сообщения от мыши. Только одно-единственное окно в данный момент может общаться с мышью и получать сообщения в независимости от того, находится ли курсор внутри него.

integer(4) function SetCapture (hWnd)

Функция устанавливает окно, которому будут адресованы сообщения мыши. Все сообщения направляются окну в независимости от того, находится ли курсор внутри него. Только одно-единственное окно в данный момент может общаться с мышью. Параметр **hWnd** – дескриптор окна. Когда окно больше не нуждается в сообщениях мыши, следует вызвать функцию **ReleaseCapture**.

integer(4) function SetWindowsHookEx (idHook, lpfn, hmod, dwThreadId)

Функция устанавливает режим перехвата некоторых типов сообщений. Процедура перехвата может контролировать события, связанные с конкретным потоком или со всеми потоками в Системе. Параметры: **idHook** – тип перехвата(все типы имеют префикс WH_), **lpfn** – адрес процедуры обработки, **hmod** – дескриптор DLL, содержащей процедуру обработки, **dwThreadId** – идентификатор потока. Возвращает дескриптор программы обработки.

integer(4) function CallNextHookEx (hhk, nCode, wParam, lParam)

Функция передает информацию следующей в списке процедуре обработки. Параметры: **hhk** – дескриптор программы обработки, **nCode** – код, переданный текущей процедуре и переадресованный следующей, **wParam** и **lParam** – содержат информацию, переданную процедуре обработки. Прикладная программа получает **hhk** в результате предыдущего обращения к функции **SetWindowsHookEx**. Возвращает величину, которая будет передана следующей процедурой в список.

logical(4) function ReleaseCapture()

Функция восстанавливает стандартное положение для сообщений мыши.

logical(4) function DestroyWindow (hWnd)

Функция уничтожает данное окно, посылая сообщения WM_DESTROY и WM_NCDESTROY. Одновременно уничтожаются меню окна, очередь сообщений потока, таймеры, удаляет монопольное использование буфера обмена. Параметр **hWnd** – дескриптор окна ***DestroyWindow*** также уничтожает модальные диалоговые окна, созданные функцией **CreateDialog**.

П-3. Функции, обслуживающие меню

logical(4) function AppendMenu (hMenu, uFlags, uIDNewItem, lpNewItem)

Функция добавляет новый пункт в конце заданного меню. Параметры: **hMenu** – дескриптор меню, **uFlags** – флаги, устанавливающие вид и функционирование пунктов меню, **uIDNewItem** – идентификатор пункта, **lpNewItem** – содержание пункта. Прикладная программа должна вызывать функцию **DrawMenuBar** после каждого изменения меню, независимо от того, находится или нет меню в отображаемом окне.

integer(4) function CheckMenuItem (hMenu, uIDCheckItem, uCheck)

Функция проверяет и устанавливает состояние метки ✓ пункта меню. Параметры: **hMenu** – дескриптор меню, **uIDCheckItem** – задает пункт ме-

ню, **uCheck** – флаг операции (MF_CHECKED или MF_UNCHECKED). Возвращает предыдущее состояние пункта меню (MF_CHECKED или MF_UNCHECKED). Если единица меню не существует, то возвращается значение #FFFFFFFF.

integer(4) function CreateMenu()

Функция создает меню. Меню изначально пустое, и для его заполнения используются функции **AppendMenu** и **InsertMenu**.

integer(4) function CreatePopupMenu()

Функция создает всплывающее меню. Меню изначально пустое, и для его заполнения используются функции **AppendMenu** и **InsertMenu**.

logical(4) function DeleteMenu (hMenu, uPosition, uFlags)

Функция удаляет пункт заданного меню. Параметры: **hMenu** – дескриптор меню, **uPosition** – идентификатор пункта меню, **uFlags** – флаг, устанавливающий способ представления **uPosition**. Прикладная программа должна вызывать функцию **DrawMenuBar** после каждого изменения меню, независимо от того, находится или нет меню в отображаемом окне.

logical(4) function DestroyMenu (hMenu)

Функция уничтожает меню и освобождает память. Параметр **hMenu** – дескриптор меню

logical(4) function DrawMenuBar (hWnd)

Функция перерисовывает меню заданного окна. Параметр **hWnd** – дескриптор окна.

logical(4) function EnableMenuItem (hMenu, uIDEnableItem, uEnable)

Функция изменяет состояние пункта меню, делая его активным, неактивным или неопределенным (***grays***). Параметры: **hMenu** – дескриптор меню, **uIDEnableItem** – идентификатор пункта меню, **uEnable** – флаг, устанавливающий способ интерпретации параметра **uIDEnableItem**. Возвращает предыдущее состояние пункта меню (MF_DISABLED, MF_ENABLED или MF_GRAYED). Если элемент меню не существует, возвращает значение #FFFFFFFF.

integer(4) function GetMenu (hWnd)

Функция возвращает дескриптор меню заданного окна. Параметр **hWnd** – дескриптор окна.

integer(4) function GetMenuItemID (hMenu, nPos)

Функция возвращает идентификатор пункта меню, размещенного в заданной позиции. Параметры: **hMenu** – дескриптор меню, **nPos** – позиция пункта меню. Функция возвращает идентификатор. Если идентификатор NULL или если заданный элемент меню вызывает перечень подпунктов, то возвращается значение #FFFFFFFF.

integer(4) function GetMenuItemCount (hMenu)

Функция сообщает общее число пунктов меню. Параметр **hMenu** – дескриптор меню.

logical(4) function InsertMenu (hMenu, uPosition, uFlags, uIDNewItem, lpNewItem)

Функция вставляет новый элемент в меню, перемещая вниз остальные пункты. Параметры: **hMenu** – дескриптор меню; **uPosition** – позиция предшествующего пункта; **uFlags** – флаги, задающие способ интерпретации параметров **uPosition** и **lpNewItem**; **uIDNewItem** – идентификатор нового пункта; **lpNewItem** – содержание нового пункта. Прикладная программа должна вызывать функцию **DrawMenuBar** после каждого изменения.

integer(4) function LoadMenu (hInstance, lpMenuName)

Функция загружает заданное меню. Параметры: **hInstance** – дескриптор приложения, **lpMenuName** – адрес строки с именем ресурса или его дескриптор. Объект, загруженный из ресурсов, освобождается автоматически при завершении программы. Возвращает дескриптор меню.

logical(4) function ModifyMenu (hMnu, uPosition, uFlags, uIDNewItem, lpNewItem)

Функция модифицирует существующий элемент меню. Используется для изменения содержания, вида и поведения пункта меню. Параметры: **hMenu** – дескриптор меню; **uPosition** – позиция предшествующего пункта; **uFlags** – флаги, задающие способ интерпретации параметров **uPosition**; **uIDNewItem** – идентификатор нового пункта; **lpNewItem** – содержание нового пункта. Прикладная программа должна вызывать функцию **DrawMenuBar** после каждого изменения.

logical(4) function SetMenu (hWnd, hMenu)

Функция связывает новое меню с заданным окном. Параметры: **hWnd** – дескриптор окна, **hMenu** – дескриптор меню.

logical(4) function RemoveMenu (hMenu, uPosition, uFlags)

Функция удаляет пункт из заданного меню. Параметры: **hMenu** – дескриптор меню, **uPosition** – идентификатор пункта меню, **uFlags** – флаг, устанавливающий способ представления **uPosition**. Прикладная программа должна вызывать функцию **DrawMenuBar()** после каждого изменения меню, независимо от того, находится или нет меню в отображаемом окне.

logical(4) function TrackPopupMenu (hMenu, uFlags, x, y, nReserved, hWnd, prcRect)

Функция отображает всплывающее меню в заданной позиции. Такое меню может появляться в любой точке экрана. Параметры: **hMenu** – дескриптор меню; **uFlags** – флаги, устанавливающие вид меню; **x, y** – координаты; **nReserved** – зарезервирова; **hWnd** – дескриптор родительского окна; **prcRect** – структура типа T_RECT с координатами границ области допустимого перемещения курсора.

integer(4) function LoadAccelerators (hInstance, lpTableName)

Функция загружает заданную таблицу акселераторов. Параметры: **hInstance** – дескриптор приложения, **lpTableName** – адрес строки с именем ресурса или его дескриптор. Таблицы, загруженные из ресурсов, освобождаются автоматически при завершении программы.

integer(4) function CreateAcceleratorTable (dummy, dummy1)

Функция создает таблицу акселераторов. Параметры: **dummy** – структура типа T_ACCEL, **dummy1** – размерность матрицы **dummy**. Функция возвращает идентификатор таблицы.

logical(4) function DestroyAcceleratorTable (hAccel)

Функция уничтожает таблицу акселераторов **hAccel**, созданных функцией **CreateAcceleratorTable**.

integer(4) function CopyAcceleratorTable (hAccelSrc, lpAccelDst, cAccelEntries)

Функция копирует таблицу акселераторов. Используется для получения данных таблицы акселераторов с заданным идентификатором. Параметры: **hAccelSrc** – идентификатор таблицы, **lpAccelDst** – структура типа T_ACCEL, **cAccelEntries** – размерность матрицы **lpAccelDst**.

integer(4) function TranslateAccelerator (hWnd, hAccTable, lpMsg)

Функция преобразует сообщения клавиш WM_KEYDOWN или WM_SYSKEYDOWN в сообщения WM_COMMAND или WM_SYSCOMMAND и посылает их непосредственно соответствующей процедуре окна. Параметры: **hWnd** – дескриптор окна, **hAccTable** – идентифи-

катор таблицы ускорителей, **lpMSG** – структура типа T_MSG. Если **TranslateAccelerator** возвращает ненулевое значение, то прикладная программа не должна использовать функцию **TranslateMessage**.

П-4. Функции, обслуживающие диалоги

integer(4) function CreateDialog (hInstance, lpTemplateName, hWndParent, lpDialogFunc)

Функция создает немодальное диалоговое окно из ресурса. Параметры: **hInstance** – дескриптор приложения, **lpTemplateName** – имя ресурса диалога или его дескриптор, **hWndParent** – дескриптор родительского окна, **lpDialogFunc** – адрес диалоговой процедуры. Функция возвращает дескриптор диалога.

integer(4) function CreateDialogParam (hInstance, lpTemplateName, hWndParent, lpDialogFunc, dwInitParam)

То же, что и **CreateDialog**, но с передачей дополнительных данных в диалоговую функцию.

integer(4) function DialogBox (hInst, lpszTemplate, hwndOwner, dlgprc)

Функция создает модальное диалоговое окно из ресурса. Параметры: **hInst** – дескриптор приложения, **lpszTemplate** – имя ресурса диалога или его дескриптор; **hwndOwner** – дескриптор родительского окна; **dlgprc** – адрес диалоговой процедуры. Возвращает значение, которое устанавливается параметром функции **EndDialog**.

integer(4) function DialogBoxParam (hInstance, lpTemplateName, hWndParent, lpDialogFunc, dwInitParam)

То же, что и **DialogBox()**, но с передачей некоторого значения в диалоговую функцию.

logical(4) function EndDialog (hDlg, nResult)

Функция уничтожает модальный диалог **hDlg** и возвращает значение **nResult**. Используется только с функцией **DialogBox**.

logical(4) function IsDialogMessage (hDlg, lpMsg)

Функция определяет, адресовано ли сообщение заданному диалоговому окну, и если это так, то обрабатывает его. Параметры: **hDlg** – дескриптор диалогового окна, **lpMSG** – структура типа T_MSG. Используется, как правило, с немодальными диалогами. Производит полную обработку сообщений и поэтому не требует вызова функций **TranslateMessage** и **DispatchMessage**.

integer(4) function MessageBox (hWnd , lpText, lpCaption, uType)

Функция создает, отображает и обслуживает окно сообщений. Параметры: **hWnd** – дескриптор родительского окна, **lpText** – строка с текстом сообщения, **lpCaption** – строка с заголовком окна, **uType** – стиль окна сообщений.

integer(4) function DlgDirList (hDlg, lpPathSpec, nIDListBox, nIDStaticPath, uFileType)

Функция заполняет поле списка именами всех файлов, соответствующих шаблону. Параметры: **hDlg** – дескриптор диалогового окна, **lpPathSpec** – строка с шаблоном для поиска файлов (*[Дисковод:] [каталог \idirectory \u [\u]] [имя файла]*), **nIDListBox** – идентификатор списка, **nIDStaticPath** – статический элемент управления для имени каталога, **uFileType** – атрибуты файлов.

integer(4) function DlgDirListComboBox (hDlg, lpPathSpec, nIDComboBox, nIDStaticPath, uFiletype)

Функция заполняет поле комбинированного списка именами всех файлов, соответствующих шаблону. Параметры: **hDlg** – дескриптор диалогового окна, **lpPathSpec** – строка с шаблоном для поиска файлов (*[Дисковод:] [каталог\idirectory\u [\u]] [имя файла]*), **nIDComboBox** – идентификатор комбинированного списка, **nIDStaticPath** – статический элемент управления для имени каталога, **uFileType** – атрибуты файлов.

logical(4) function GetOpenFileName (dummy)

Функция создает стандартное диалоговое окно, которое дает возможность пользователю выбрать файл для последующего его применения в приложении. Параметр **dummy** – структура типа T_OPENFILENAME.

logical(4) function GetSaveFileName (dummy)

Функция создает стандартное диалоговое окно, которое дает возможность пользователю выбрать имя файла для сохранения в нем данных. Параметр **dummy** – структура типа T_OPENFILENAME.

Integer(2) function GetFileTitle (dummya, dummyb, dummyc)

Функция возвращает имя файла с заданным идентификатором. Параметры: **dummya** – строка, задающая путь, **dummyb** – С-строка с именем файла, **dummyc** – длина строки **dummyb** с учетом пробела завершения.

logical(4) function ChooseColor (dummy)

Функция создает стандартное диалоговое окно, которое дает возможность пользователю выбрать цвет для последующего его применения в приложении. Параметр **dummy** – структура типа T_CHOOSECOLOR.

Integer(4) function FindText (dummy)

Функция создает определенное Системой диалоговое окно, которое дает возможность пользователю найти текст внутри документа. Параметр **dummy** – структура типа T_FINDREPLACE. Функция не выполняет операцию поиска, а просто передает строку, введенную пользователем, которая затем применяется в программе.

Integer(4) function ReplaceText (dummy)

Функция создает стандартное диалоговое окно, которое дает возможность пользователю найти и заменить текст внутри документа. Параметр **dummy** – структура типа T_FINDREPLACE. Функция не выполняет операцию поиска, а просто передает строку, введенную пользователем, которая затем применяется в программе.

logical(4) function ChooseFont (dummy)

Функция создает стандартное диалоговое окно, которое дает возможность пользователю выбрать тип шрифта для последующего его применения в приложении. Параметр **dummy** – структура типа T_CHOOSEFONT.

logical(4) function PrintDlg (dummy)

Функция создает диалоговые окна ***Print*** или ***Print Setup***. Диалоговое окно ***Print*** дает возможность пользователю задать параметры печати, а ***Print Setup*** позволяет задать конфигурацию и инсталлировать принтер. Параметр **dummy** – структура типа T_PRINTDLG.

П-5. Функции, обслуживающие элементы управления диалогом

logical(4) function CheckDlgButton (hDlg, nIDButton, uCheck)

Функция устанавливает и проверяет состояние метки на кнопке управления. Параметры: **hDlg** – дескриптор диалога, **nIDButton** – идентификатор кнопки, **uCheck** – флаг, устанавливающий стиль проверки состояния.

logical(4) function CheckRadioButton (hDlg, nIDFirstButton, nIDLastButton, nIDCheckButton)

Функция добавляет метку на заданную селекторную кнопку в группе и удаляет ее со всех остальных кнопок в группе. Параметры: **hDlg** – деск-

риптор диалога, **nIDFirstButton** – идентификатор первой кнопки в группе, **nIDLastButton** – идентификатор последней кнопки, **nIDCheckButton** – идентификатор кнопки, которая должна быть помечена.

integer(4) function GetDlgItem (hDlg, nIDDlgItem)

Функция возвращает дескриптор элемента управления заданного диалога. Параметры: **hDlg** – идентификатор диалога, **nIDDlgItem** – идентификатор элемента управления.

integer(4) function GetDlgItemInt (hDlg, nIDDlgItem, lpTranslated, bSigned)

Функция преобразует текст заданного окна диалога в целочисленное значение. Параметры: **hDlg** – идентификатор диалога, **nIDDlgItem** – идентификатор элемента управления, **lpTranslated** – возвращаемая переменная логического типа, **bSigned** – флаг, указывающий на необходимость обработки знака минус. Возвращает целочисленное значение.

integer(4) function GetDlgItemText (hDlg, nIDDlgItem, lpString, nMaxCount)

Функция передает заголовок или текст, связанный с элементом управления в диалоговом окне. Параметры: **hDlg** – идентификатор диалога, **nIDDlgItem** – идентификатор элемента управления, **lpString** – текстовой буфер, **nMaxCount** – длина текстовой строки.

integer(4) function IsDlgButtonChecked (hDlg, nIDButton)

Функция определяет состояние метки на кнопке. Если кнопка имеет три состояния, то определяется, в каком из них она находится. Параметры: **hDlg** – идентификатор диалога, **nIDButton** – идентификатор кнопки.

integer(4) function SendDlgItemMessage (hDlg, nIDDlgItem, Msg, wParam, lParam)

Функция посылает сообщение заданному элементу управления диалоговым окном. Параметры: **hDlg** – идентификатор диалога, **nIDDlgItem** – идентификатор элемента управления, **MSG** – собственно сообщение, **wParam** и **lParam** – дополнительная информация, которая зависит от содержания сообщения. Возвращаемое значение зависит от типа посланного сообщения и результата его обработки.

logical(4) function SetDlgItemInt (hDlg, nIDDlgItem, uValue, bSigned)

Функция выводит в окно элемента управления строковое представление целого числа. Параметры: **hDlg** – идентификатор диалога, **nIDDlgItem** – идентификатор элемента управления, **uValue** – целочисленное значение, **bSigned** – флаг, указывающий на необходимость обработки знака минус.

logical(4) function SetDlgItemText (hDlg, nIDDlgItem, lpString)

Функция выводит текст в окно заданного элемента диалога. Параметры: **hDlg** – идентификатор диалога, **nIDDlgItem** – идентификатор элемента управления, **lpString** – строка текста.

П-6. Функции для работы с таймером

integer(4) function SetTimer (hWnd, nIDEvent, uElapse, lpTimerFunc, flag)

Функция создает таймер с определенным значением временного интервала. Параметры: **hWnd** – дескритор окна, которое будет связано с таймером, **nIDEvent** – идентификатор таймера, **uElapse** – шаг таймера в миллисекундах, **lpTimerFunc** – адрес функции, обрабатывающей сообщения таймера WM_TIMER, **flag** – флаг, который не используется.

logical(4) function KillTimer (hWnd, uIDEvent)

Функция уничтожает таймер. Параметры: **hWnd** – дескриптор окна, **uIDEvent** – идентификатор таймера.

П-7. Функции для взаимодействия с реестром

integer(4) function RegCloseKey (hKey)

Функция закрывает ключ с заданным дескриптором **hKey**. Может пройти несколько секунд после закрытия ключа прежде, чем данные будут сброшены из памяти на диск.

integer(4) function RegConnectRegistry (lpMachineName, hKey, phkResult)

Функция устанавливает соединение предопределенного ключа с программой обработки реестра на другом компьютере. Параметры: **lpMachineName** – строка с именем удаленного компьютера; **hKey** – предопределенный ключ, может быть HKEY_LOCAL_MACHINE или HKEY_USERS; **phkResult** – адрес переменной, которая получает дескриптор ключа, идентифицированного предопределенным дискриптором на удаленном компьютере. Нельзя использовать ключи HKEY_CLASSES_ROOT и HKEY_CURRENT_USER.

integer(4) function RegCreateKey (hKey, lpSubKey, phkResult)

Функция создает ключ. Если ключ уже существует в реестре, то функция открывает его. Предусмотрена совместимость с версией 3.1. Новые прикладные программы должны использовать функцию **RegCreateKeyEx**. Параметры: **hKey** – дескриптор ключа, **lpSubKey** – строка с именем ключа, **phkResult** – адрес переменной, которая получает дескриптор ключа. При-

кладная программа может использовать функцию для создания нескольких ключей (*"Subkey1\subkey2\subkey3\subkey4"*).

integer(4) function RegCreateKeyEx (hKey, lpSubKey, Reserved, lpClass, dwOptions, samDesired, lpSecurityAttributes, phkResult, lpdwDisposition)

Функция создает определенный ключ, а если ключ уже существует, открывает его. Параметры: **hKey** – дескриптор ключа, **lpSubKey** – строка с именем ключа, **Reserved** – зарезервирован, **lpClass** – строка с именем класса, **dwOptions** – опции ключа, **samDesired** – маска доступа к ключу, **lpSecurityAttributes** – структура типа T_SECURITY_ATTRIBUTES с атрибутами защиты, **phkResult** – адрес дескриптора ключа, **lpdwDisposition** – адрес переменной, описывающей информацию о процедуре создания ключа. Функция возвращает ERROR_SUCCESS или собщение об ошибке.

integer(4) function RegDeleteKey (hKey, lpSubKey)

Функция удаляет ключ и все его данные. Параметры: **hKey** – дескриптор открытого ключа, **lpSubKey** – строка с именем подключа. Подключ не должен иметь подчиненных ключей. Функция возвращает ERROR_ SUCCESS или сообщение об ошибке.

integer(4) function RegDeleteValue (hKey, lpValueName)

Функция удаляет поименованное значение из заданного ключа. Параметры: **hKey** – дескриптор открытого ключа, **lpValueName** – строка с именем данных. Функция возвращает ERROR_SUCCESS или собщение об ошибке. Ключ должен быть открыт с доступом KEY_SET_VALUE.

integer(4) function RegEnumKey (hKey, dwIndex, lpName, cbName)

Функция перебирает подключи открытого ключа реестра. Предусмотрена совместимость с версией 3.1. Для Win32 прикладные программы должны использовать функцию **RegEnumKeyEx()**. Параметры: **hKey** – дескриптор открытого ключа, **dwIndex** – индекс подключа, **lpName** – строка с именем подключа, **cbName** – длина строки **lpName**. Функция возвращает **ERROR_SUCCESS** или сообщение об ошибке. Ключ должен быть открыт с доступом KEY_ENUMERATE_SUB_KEYS.

integer(4) function RegEnumKeyEx (hKey, dwIndex, lpName, lpcbName, lpReserved, lpClass, lpcbClass, lpftLastWriteTime)

Функция перебирает подключи открытого ключа. В отличие от функции **RegEnumKey** передает имя класса подключа и дату и время последнего изменения. Параметры: **hKey** – дескриптор открытого ключа, **dwIndex** – индекс подключа, **lpName** – строка с именем подключа, **lpcbName** – длина

строки **lpName**, **lpReserved** – зарезервирован, **lpClass** – строка с именем класса, **lpcbClass** – длина строки, **lpftLastWriteTime** – структура типа T_FILETIME. Функция возвращает ERROR_SUCCESS или сообщение об ошибке. Подключ должен быть открыт с доступом KEY_ENUMERATE_SUB_KEYS.

integer(4) function RegEnumValue (hKey, dwIndex, lpValueName, lpcbValueName, lpReserved, lpType, lpData, lpcbData)

Функция перебирает значения для открытого ключа, копирует имя и значение с заданным индексом. Параметры: **hKey** – дескриптор открытого ключа, **dwIndex** – индекс значения, **lpValueName** – строка с именем значения, **lpcbValueName** – длина строки **lpValueName**, **lpReserved** – зарезервирован, **lpType** – формат значений, **lpData** – адрес буфера данных, **lpcbData** – размер буфера. Функция возвращает ERROR_SUCCESS или сообщение об ошибке. Подключ должен быть открыт с доступом KEY_QUERY_VALUE.

integer(4) function RegFlushKey (hKey)

Функция записывает все атрибуты открытого ключа реестра на диск. Параметр **hKey** – дескриптор ключа. Значения будут переписаны из памяти на диск и без вызова **RegFlushKey()**, например при закрытии Системы. Функция может записывать часть или все ключи. Вызов этой функции может пагубно сказаться на эффективности прикладной программы. Вообще функция **RegFlushKey()** должна использоваться только в случае крайней необходимости.

integer(4) function RegLoadKey (hKey, lpSubKey, lpFile)

Функция создает подключ предопределенного ключа HKEY_USER или HKEY_LOCAL_MACHINE и перезагружает в него информацию из заданного файла. Параметры: **hKey** – дескриптор ключа, **lpSubKey** – строка с именем ключа, **lpFile** – строка с именем файла, который должен быть создан функцией **RegSaveKey()**.

integer(4) function RegNotifyChangeKeyValue (hKey, bWatchSubtree, dwNotifyFilter, hEvent, fAsynchronus)

Функция сообщает о внесении изменений в реестр. Параметры: **hKey** – дескриптор ключа, **bWatchSubtree** – флаг, который указывает, сообщить ли об изменениях ключа и всех его подключей или только ключа, **dwNotifyFilter** – устанавливает регистрируемый тип изменения, **hEvent** – идентификатор события, связанный с изменениями, **fAsynchronus** – флаг, устанавливающий способ сообщения о событии.

integer(4) function RegOpenKey (hKey, lpSubKey, phkResult)

Функция открывает ключ. Предусмотрена совместимость с версией 3.1. Для Win32 прикладные программы должны использовать функцию **RegOpenKeyEx()**. Параметры: **hKey** – дескриптор ключа, **lpSubKey** – строка с именем подключа, **phkResult** – дескриптор открытого ключа.

integer(4) function RegOpenKeyEx (hKey, lpSubKey, ulOptions, samDesired, phkResult)

Функция открывает ключ. Параметры: **hKey** – дескриптор ключа, **lpSubKey** – строка с именем подключа, **ulOptions** – зарезервиран, **samDesired** – битовая маска доступа, **phkResult** – дескриптор открытого ключа.

integer(4) function RegQueryInfoKey (hKey, lpClass, lpcbClass, lpReserved, lpcSubKeys, lpcbMaxSubKeyLen, lpcbMaxClassLen, lpcValues, lpcbMaxValueNameLen, lpcbMaxValueLen, lpcbSecurityDescriptor, lpftLastWriteTime)

Функция передает данные о заданном ключе. Параметры: **hKey** – дескриптор ключа, **lpClass** – строка с именем класса, **lpcbClass** – длина строки **lpClass**, **lpReserved** – зарезервирован, **lpcSubKeys** – адрес переменной числа подключей, **lpcbMaxSubKeyLen** – адрес переменной, указывающей максимальную длину имени подключа, **lpcbMaxClassLen** – адрес переменной, указывающей максимальную длину имени класса, **lpcValues** – адрес переменной, указывающей число значений, **lpcbMaxValueNameLen** – адрес переменной, указывающей максимальную длину имени данных, **lpcbMaxValueLen** – адрес переменной, указывающей максимальный размер данных, **lpcbSecurityDescriptor** – адрес переменной, описывающей атрибуты защиты, **lpftLastWriteTime** – структура типа T_FILETIME, указывающая дату последних изменений. Ключ должен быть открыт с доступом KEY_QUERY_VALUE.

integer(4) function RegQueryValue (hKey, lpSubKey, lpValue, lpcbValue)

Функция для заданного ключа в реестре передает данные, связанные с безымянным значением(имя NULL). Предусмотрена совместимость с версией 3.1. Для Win32 прикладные программы должны использовать функцию **RegQueryValueEx()**. Параметры: **hKey** – дескриптор ключа, **lpSubKey** – строка с именем подключа, **lpValue** – строка с именем данных, **lpcbValue** – адрес переменной, указывающей длину строки. Ключ должен быть открыт с доступом KEY_QUERY_VALUE.

integer(4) function RegQueryValueEx (hKey, lpValueName, lpReserved, lpType, lpData, lpcbData)

Функция передает тип и данные значения заданного имени. Параметры: **hKey** – дескриптор ключа, **lpValueName** – строка с именем данных, **lpReserved** – зарезервирован, **lpType** – адрес формата данных, **lpData** – адрес буфера данных, **lpcbData** – размер буфера данных. Ключ должен быть открыт с доступом KEY_QUERY_VALUE.

integer(4) function RegReplaceKey (hKey, lpSubKey, lpNewFile, lpOldFile)

Функция заменяет файл, поддерживающий ключ и все его подключи, так, что при новом запуске они будут иметь значения, сохраненные в новом файле. Параметры: **hKey** – дескриптор ключа, **lpSubKey** – строка с именем подключа, **lpNewFile** – строка с именем нового файла, **lpOldFile** – строка с именем старого файла. Функция возвращает ERROR_SUCCESS или сообщение об ошибке.

integer(4) function RegRestoreKey (hKey, lpFile, dwFlags)

Функция копирует информацию реестра из файла под заданный ключ. Параметры: **hKey** – дескриптор ключа, **lpFile** – строка с именем файла, **dwFlags** – флаг, указывающий, сохранится ли вид ключа после закрытия Системы.

integer(4) function RegSaveKey (hKey, lpFile, lpSecurityAttributes)

Функция сохраняет ключ, все подключи и значения в новом файле. Параметры: **hKey** – дескриптор ключа, **lpFile** – строка с именем файла, **lpSecurityAttributes** – структура типа T_SECURITY_ATTRIBUTES.

integer(4) function RegSetKeySecurity (hKey, SecurityInformation, pSecurityDescriptor)

Функция устанавливает защиту открытого ключа реестра. Параметры: **hKey** – дескриптор ключа, **SecurityInformation** – опция защиты, **pSecurityDescriptor** – структура типа T_SECURITY_DESCRIPTOR

integer(4) function RegSetValue (hKey, lpSubKey, dwType, lpData, cbData)

Функция связывает значение с ключом. Значение должно быть строкой и не может иметь имени. Предусмотрена совместимость с версией 3.1. Для Win32 прикладные программы должны использовать функцию **RegSetValueEx()**. Параметры: **hKey** – дескриптор ключа, **lpSubKey** – строка с именем подключа, **dwType** – тип данных (например, REG_SZ), **lpData** – строка с данными, **cbData** – длина строки данных.

integer(4) function RegSetValueEx (hKey, lpValueName, Reserved, dwType, lpData, cbData)

Функция заносит в буфер значения открытого ключа реестра. Может также устанавливать дополнительное значение и тип информации. Параметры: **hKey** – дескриптор ключа, **lpValueName** – строка с именем данных, **Reserved** – зарезервирован, **dwType** – тип данных, **lpData** – адрес буфера данных, **cbData** – размер буфера данных.

integer(4) function RegUnLoadKey (hKey, lpSubKey)

Функция выгружает определенный ключ и подключи из реестра. Параметры: **hKey** – дескриптор ключа, **lpSubKey** – строка с именем подключа.

П-8. Функции для работы с буфером обмена

logical(4) function ChangeClipboardChain (hWndRemove, hWndNewNext)

Функция удаляет заданное окно из списка просмотра буфера обмена. Параметры: **hWndRemove** – дескриптор окна, которое будет удалено, **hWndNewNext** – дескриптор следующего окна. Окно ***hWndNewNext*** заменяет ***hWndRemove.***

logical(4) function CloseClipboard()

Функция закрывает буфер обмена. Это дает возможность другим окнам обратиться к буферу обмена.

integer(4) function CountClipboardFormats()

Функция возвращает число различных форматов данных в буфере обмена на настоящее время.

logical(4) function EmptyClipboard()

Функция освобождает буфер обмена и ликвидирует дескриптор данных. Функция затем назначает монопольное использование буфера обмена окном, которое в настоящее время имеет открытый буфер обмена. Перед вызовом функции прикладная программа должна открыть буфер обмена, используя функцию **OpenClipboard()**.

integer(4) function EnumClipboardFormats (format)

Функция перебирает форматы данных, которые в настоящее время доступны в буфере обмена. Параметр **format** – идентификатор формата. Форматы хранятся в упорядоченном списке. Поэтому вы должны несколько раз обращаться к функции **EnumClipboardFormats**. После каждого обращения параметр формата определяет разрешенный формат буфера обмена,

а функция возвращает следующий. Буфер обмена должен быть предварительно открыт функцией **OpenClipboard**.

integer(4) function GetClipboardData (uFormat)

Функция возвращает дескриптор глобальной памяти, содержащей данные в заданном формате. Буфер обмена должен быть предварительно открыт. Параметр **uFormat** определяет формат буфера обмена. Для описания форматов буфера обмена используется функция **SetClipboardData**. Прикладная программа может перебирать доступные форматы заранее с помощью функции **EnumClipboardFormats**.

integer(4) function GetClipboardFormatName (format, lpszFormatName, cchMaxCount)

Функция копирует в буфер имя заданного зарегистрированного формата. Параметры: **format** – тип формата, **lpszFormatName** – строка с именем формата, **cchMaxCount** – максимальная длина строки.

integer(4) function GetClipboardOwner()

Функция возвращает дескриптор окна, которое общается с буфером.

integer(4) function GetClipboardViewer()

Функция возвращает дескриптор первого окна просмотра в списке буфера обмена.

integer(4) function GetOpenClipboardWindow()

Функция возвращает дескриптор окна, которое имеет открытый буфер обмена.

integer(4) function GetPriorityClipboardFormat (paFormatPriorityList, cFormats)

Функция возвращает первый наиболее подходящий формат из списка форматов буфера обмена. Параметры: **paFormatPriorityList** – массив целых чисел без знака, идентифицирующих форматы буфера обмена в приоритетном порядке, **cFormats** – размерность массива.

integer(4) function GlobalAlloc (uFlags, dwBytes)

Функция выделяет память и возвращает дескриптор выделенного блока. Параметры: **uFlags** – флаг, задающий свойства блока памяти, **dwBytes** – размер блока в байтах. Флаги имеют префикс GMEM_ (например, GMEM_FIXED). Память, выделенная для буфера обмена, должна быть перемещаемой (**uFlags** = IOR(GMEM_ZEROINIT, GMEM_MOVEABLE) = GHND).

integer(4) function GlobalReAlloc (hMem, dwBytes, uFlags)

Функция изменяет размер или атрибуты заданного глобального объекта памяти. Размер может и увеличиваться и уменьшаться. Параметры: **hMem** – дескриптор блока памяти, **dwBytes** – новый размер блока, **uFlags** – флаг, задающий свойства блока памяти. Возвращает дескриптор обновленного блока.

integer(4) function GlobalSize (hMem)

Функция возвращает размер в байтах глобального объекта памяти.

Параметр **hMem** – дескриптор блока памяти. Размер блока памяти может оказаться больше того, который запрашивался при выделении памяти.

integer(4) function GlobalFlags (hMem)

Функция возвращает информацию о свойствах глобального объекта памяти. Параметр **hMem** – дескриптор блока памяти.

integer(4) function GlobalLock (hMem)

Функция фиксирует глобальный объект памяти и возвращает указатель на первый байт блока памяти. Блок памяти, связанный с блокированным объектом памяти, не может перемещаться или выгружаться. Параметр **hMem** – дескриптор блока памяти. К зафиксированному блоку имеет доступ только одна программа.

integer(4) function GlobalHandle (pMem)

Функция возвращает дескриптор глобального блока памяти с заданным адресом. Параметр **pMem** – адрес блока.

logical(4) function GlobalUnlock (hMem)

Функция разблокирует объект памяти. Функция не действует на объекты памяти с флагом GMEM_FIXED. Параметр **hMem** – дескриптор блока памяти.

integer(4) function GlobalFree (hMem)

Функция освобождает заданный глобальный объект памяти и лишает законной силы дескриптор. Параметр **hMem** – дескриптор блока памяти.

logical(4) function IsClipboardFormatAvailable (format)

Функция определяет, содержит ли буфер обмена данные в определенном формате. Параметр **format** – заданный формат.

logical(4) function OpenClipboard (hWndNewOwner)

Функция открывает буфер обмена для использования и предотвращает изменения содержания буфера обмена другими программами. Параметр

hWndNewOwner – дескриптор окна, которое будет связано с открытым буфером.

integer(4) function SetClipboardViewer (hWndNewViewer)

Функция добавляет новое окно в список буфера обмена. Окна просмотра буфера обмена получают сообщение WM_DRAWCLIPBOARD всякий раз, когда содержание буфера обмена изменяется. Параметр **hWndNew Viewer** – дескриптор нового окна.

integer (4) function SetClipboardData (uFormat, hMem)

Функция записывает данные в буфер обмена в определенном формате. Окно должно быть предварительно связано с буфером обмена, а прикладная программа должна вызвать функцию **OpenClipboard**. Параметры: **uFormat** – формат буфера обмена, **hMem** – дескриптор глобальной памяти. Возвращает новый дескриптор памяти, на которую настроен буфер.

integer(4) function RegisterClipboardFormat (lpszFormat)

Функция регистрирует новый формат буфера обмена для его последующего использования. Параметр **lpszFormat** – строка с текстом нового формата. Возвращает идентификатор формата. Если зарегистрированный формат с данным именем уже существует, то новый формат не регистрируется и возвращается идентификатор существующего формата. Это дает возможность более чем одной прикладной программе копировать и изменять данные, использующие один и тот же формат буфера обмена. Зарегистрированные форматы имеют значения в диапазоне от #C000 до #FFFF.

П-9. Функции для работы с ресурсами

integer(4) function LoadAccelerators (hInstance, lpTableName)

Функция загружает заданную таблицу акселераторов. Параметры: **hInstance** – дескриптор приложения, **lpTableName** – адрес строки с именем ресурса или его дескриптор. Объект, загруженный из ресурсов, освобождается автоматически при завершении программы. Возвращает дескриптор таблицы.

integer(4) function LoadBitmap (hInstance, lpBitmapName)

Функция загружает заданный растровый ресурс. Параметры: **hInstance** – дескриптор приложения, **lpBitmapNam** – адрес растрового имени ресурса или его дескриптор. Объект, загруженный из ресурсов, освобождается автоматически при завершении программы. Возвращает дескриптор растра.

integer(4) function LoadCursor (hInstance, lpCursorName)

Функция загружает заданный ресурс курсора. Параметры: **hInstance** – дескриптор приложения, **lpCursorName** – адрес строки с именем ресурса или его дескриптор. Объект, загруженный из ресурсов, освобождается автоматически при завершении программы. Возвращает дескриптор курсора.

integer(4) function LoadIcon (hInstance, lpIconName)

Функция загружает заданный ресурс пиктограммы. Параметры: **hInstance** – дескриптор приложения, **lpIconName** – адрес строки с именем ресурса или его дескриптор. Объект, загруженный из ресурсов, освобождается автоматически при завершении программы. Возвращает дескриптор иконки.

integer(4) function LoadMenu (hInstance, lpMenuName)

Функция загружает заданное меню. Параметры: **hInstance** – дескриптор приложения, **lpMenuName** – адрес строки с именем ресурса или его дескриптор. Объект, загруженный из ресурсов, освобождается автоматически при завершении программы. Возвращает дескриптор меню.

integer(4) function LoadResource (hModule, hResInfo)

Функция загружает заданный ресурс в глобальную память. Параметры: **hModule** – дескриптор модуля, содержащего ресурс, **hResInfo** – дескриптор ресурса, который должен быть определен с помощью функций **FindResource** или **FindResourceEx**. Система не освобождает загруженные ресурсы автоматически. Прикладные программы должны использовать функцию **FreeResource**, чтобы явно выгрузить из памяти ресурс, когда он больше не нужен. Возвращает дескриптор глобального блока памяти, который содержит данные, связанные с ресурсом.

integer(4) function SizeofResource (hModule, hResInfo)

Функция возвращает размер заданого ресурса в байтах. Параметры: **hModule** – дескриптор модуля, содержащего ресурс, **hResInfo** – дескриптор ресурса, который должен быть определен с помощью функций **FindResource** или **FindResourceEx**. Возвращенное значение может быть большее, чем фактический размер ресурса, из-за выравнивания.

integer(4) function LoadString (hInstance, uID, lpBuffer, nBufferMax)

Функция загружает заданный ресурс строк. Параметры: **hInstance** – дескриптор приложения, **uID** – идентификатор строки, **lpBuffer** – текстовая строка, **nBufferMax** – длина строки. Объект, загруженный из ресурсов, ос-

вобождается автоматически при завершении программы. Возвращает дескриптор числа загруженных символов.

integer(4) function FindResource (hModule, lpName, lpType)

Функция определяет расположение ресурса. Параметры: **hModule** – дескриптор модуля, **lpName** – адрес строки с именем, **lpType** – тип ресурса. Все типы имеют префикс RT_. Возвращает дескриптор блока памяти с информацией о ресурсе. Чтобы получить дескриптор ресурса, следует передать полученный дескриптор функции **LoadResource**. Прикладная программа может использовать функцию, чтобы найти ресурс любого типа, но она должна обращаться только к двоичным данным ресурса для последующего обращения к функции **LockResource**.

П-10. Макросы

integer(2) function HiWord (param)

Функция восстанавливает старшее слово из данного значения с 32 битами.

integer(2) function LoWord (param)

Функция восстанавливает младшее слово из данного значения с 32 битами.

integer(4) function RGB (red, green, blue)

Функция формирует из базовых цветов (RGB) цвет, основанный на параметрах и цветных возможностях устройства вывода. Параметы: **red**, **green**, **blue** – байтовые аргументы, задающие базовые цвета. Интенсивность для каждого параметра находится в диапазоне от 0 до 255.

BYTE function GetRedValue(param)

Функция определяет содержание красного в цвете **param**.

BYTE function GetGreenValue(param)

Функция определяет содержание зеленого в цвете **param**.

BYTE function GetBlueValue(param)

Функция определяет содержание синего в цвете **param**.

BYTE function HiByte(param)

Функция определяет содержание старшего байта параметра **integer(2) param**.

BYTE function LoByte (param)

Функция определяет содержание младшего байта параметра **integer(2) param**.

integer(4) function MakeLong (wLow, wHigh)

Функция возвращает значение **integer(4)**, образованное из двух величин типа **integer(2)**. Параметры: **wLow** – младшее слово, **wHigh** – старшее слово.

integer(4) function MakeLParam (wLow, wHigh)

Функция возвращает значение **integer(4)**, образованное из двух величин типа **integer(2)**. Параметры: **wLow** – младшее слово, **wHigh** – старшее слово.

integer(4) function MakeLResult (wLow, wHigh)

Функция возвращает значение **integer(4)**, образованное из двух величин типа **integer(2)**. Параметры: **wLow** – младшее слово, **wHigh** – старшее слово.

subroutine MakePointS (dword, ret_val)

Подпрограмма преобразует координаты в указатель на структуру типа T_POINT. Параметры: **dword = MakeLong (x, y)**, **ret_val** – указатель на структуру типа T_POINT.

integer(2) function MakeWord (bLow, bHigh)

Функция образует значение типа **integer(2)** из пары байтовых параметров. Параметры **bLow** и **bHigh** – аргументы типа BYTE.

integer(4) function MakeWparam (bLow, bHigh)

Функция возвращает значение **integer(4)**, образованное из двух величин типа **integer(2)**. Параметры: **wLow** – младшее слово, **wHigh** – старшее слово.

П-11. Графические функции

logical(4) function Arc (dummy0, dummy1, dummy2, dummy3, dummy4, dummy5, dummy6, dummy7, dummy8)

Функция Arc рисует дугу. Параметры: **dummy0** – дескриптор контекста устройства; **dummy1**, **dummy2**, **dummy3** и **dummy4** – координаты описанного прямоугольника; **dummy5**, **dummy6**, **dummy7** и **dummy8** – координаты начальной и конечной точек дуги. Дуга рисуется текущим пером, всегда против часовой стрелки.

integer(4) function BeginPaint (hWnd, lpPaint)

Функция подготавливает заданное окно к перерисовке и заполняет структуру типа T_PAINTSTRUCT информацией о параметрах перерисов-

ки. Параметры: **hWnd** – дескриптор окна, **lpPaint** – структуру типа T_PAINTSTRUCT. Возвращает дескриптор контекста устройства данного окна. Функция автоматически устанавливает область отсечения. Область модификации устанавливается функцией **InvalidateRect** или **InvalidateRgn** и Системой. Прикладная программа должна вызвать **BeginPaint** только в ответ на сообщение WM_PAINT. Каждое обращение к **BeginPaint** должно иметь соответствующее обращение к функции **EndPaint**.

logical(4) function BitBlt (dummy0, dummy1, dummy2, dummy3, dummy4, dummy5, dummy6, dummy7, dummy8)

Функция выполняет побитовое копирование цветных растровых изображений из исходного контекста устройства в контекст устройства адресата. Параметры: **dummy0** – дескриптор контекста устройства адресата; **dummy1** и **dummy2** – координаты верхнего левого угла прямоугольника, в который будет выводиться изображение; **dummy3** и **dummy4** – ширина и высота прямоугольника, в который будет выводиться изображение; **dummy5** – дескриптор исходного контекста устройства; **dummy6** и **dummy7** – координаты левого верхнего угла исходного изображения; **dummy8** – способ вывода изображения.

logical(4) function Chord (dummy0, dummy1, dummy2, dummy3, dummy4, dummy5, dummy6, dummy7, dummy8)

Функция рисует сегмент текущим пером и заполняет его, используя текущую кисть. Параметры: **dummy0** – дескриптор контекста устройства; **dummy1**, **dummy2**, **dummy3** и **dummy4** – координаты описанного прямоугольника; **dummy5**, **dummy6**, **dummy7** и **dummy8** – координаты начальной и конечной точек дуги эллипса.

integer function CreateCompatibleDC (dummy0)

Функция создает контекст устройства в памяти, совместимый с заданным устройством. Параметр **dummy0** – контекст устройства памяти. Если вы больше не нуждаетесь в контексте устройства памяти, удалите его, вызовая функцию **DeleteDC**.

integer function CreateFont (dummy0, dummy1, dummy2, dummy3, dummy4, dummy5, dummy6, dummy7, dummy8, dummy9, dummy10, dummy11, dummy12, dummy13)

Функция создает логический шрифт, который имеет специфические характеристики и может впоследствии быть выбран как рабочий шрифт для любого устройства. Параметры: **dummy0** – высота шрифта, **dummy1** – ширина шрифта, **dummy2** – угол наклона текста, **dummy3** – угол наклона

символов, **dummy4** – насыщенность шрифта, **dummy5** – курсив, **dummy6** – подчеркивание, **dummy7** – зачеркивание, **dummy8** – множество символов шрифта, **dummy9** – точность отображения, **dummy10** – отсечение, **dummy11** – качество, **dummy12** – тип и семейство шрифтов, **dummy13** – строка с именем шрифта. Созданный шрифт должен быть уничтожен функцией **DeleteObject**.

integer function CreatePen (dummy0, dummy1, dummy2)

Функция создает логическое перо, которое имеет определенный стиль, ширину и цвет. Перо может впоследствии быть выбрано в контекст устройства и использоваться для рисования линий и кривых. Параметры: **dummy0** – стиль пера, **dummy1** – ширина пера, **dummy2** – цвет пера. Все стили пера имеют префикс PS_ (напрмер, PS_SOLID). Возвращает дескриптор пера.

integer function CreateSolidBrush (dummy0)

Функция создает логическую сплошную кисть. Параметр **dummy0** – цвет кисти. Возвращает дескриптор кисти. После того как прикладная программа создала кисть, она может выбирать ее, вызывая функцию **SelectObject**.

logical(4) function Ellipse (dummy0, dummy1, dummy2, dummy3, dummy4)

Функция рисует эллипс текущим пером и заполняет его, используя текущую кисть. Центр эллипса совмещен с центром заданного прямоугольника. Параметры: **dummy0** – дескриптор контекста устройства; **dummy1**, **dummy2**, **dummy3** и **dummy4** – координаты описанного прямоугольника. Текущая позиция не используется и не изменяется.

logical(4) function EndPaint (hWnd, lpPaint)

Функция заканчивает процесс перерисовки данного окна. Эта функция требуется для каждого обращения к функции **BeginPaint**, но только после того, как процесс полностью завершен. Параметры: **hWnd** – дескриптор окна, **lpPaint** – структура типа T_PAINTSTRUCT.

integer(4) function FillRect (hDC, lprc, hbr)

Функция заполняет прямоугольник, используя заданную кисть. Заполняется область от левой и верхней рамок, включая их, до правой и нижней рамок. Параметры: **hDC** – дескриптор контекста устройства, **lprc** – структура типа T_RECT с координатами, **hbr** – дескриптор кисти.

integer(4) function GetDC (hWnd)

Функция возвращает дескриптор контекста устройства для рабочей области заданного окна. Контекст устройства может использоваться в последующих функциях графического интерфейса (GDI). Параметр **hWnd** – дескриптор окна. По окончании сеанса рисования контекст должен быть освобожден с помощью функции **ReleaseDC**.

integer function GetROP2 (dummy0)

Функция возвращает режим наложения цветов для заданного контекста устройства. Параметр **hDC** – дескриптор контекста устройства.

integer(4) function GetWindowDC (hWnd)

Функция возвращает контекст устройства для всего окна, включая название окна, меню и линейки прокрутки. Параметр **hWnd** – дескриптор окна. По окончании сеанса рисования контекст должен быть освобожден с помощью функции **ReleaseDC**.

logical(4) function InvalidateRect (hWnd, lpRect, bErase)

Функция добавляет прямоугольник к области модификации заданного окна. Область модификации представляет часть рабочей области окна, которая должна быть повторно выведена. Параметры: **hWnd** – дескриптор окна, **lpRect** – структура типа T_RECT с координатами прямоугольника, **bErase** – режим перерисовки фона.

logical(4) function LineTo (dummy0, dummy1, dummy2)

Функция рисует линию из текущей позиции до заданой точки, но исключая ее. Параметры: **dummy0** – дескриптор контекста устройства, **dummy1** и **dummy2** – координаты конечной точки. Для рисования используется текущее перо. После прорисовки текущая позиция заменяется на конечную.

logical(4) function MoveToEx (dummy0, dummy1, dummy2, dummy3)

Функция устанавливает текущую позицию в заданную точку и возвращает предыдущую. Параметры: **dummy0** – дескриптор контекста устройства, **dummy1** и **dummy2** – координаты **x** и **y** новой точки, **dummy3** – структура типа T_POINT со старыми координатами.

logical(4) function PatBlt (dummy0, dummy1, dummy2, dummy3, dummy4, dummy5)

Функция заполняет заданный прямоугольник, используя текущую кисть заданного контекста устройства. Параметры: **dummy0** – дескриптор контекста устройства; **dummy1**, **dummy2**, **dummy3** и **dummy4** – координа-

ты и размеры прямоугольника; **dummy5** – способ использования кисти (например, PATCOPY – заполнение текущей кистью).

logical(4) function Pie (dummy0, dummy1, dummy2, dummy3, dummy4, dummy5, dummy6, dummy7, dummy8)

Функция рисует сектор эллипса текущим пером и заполняет его, используя текущую кисть. Параметры: **dummy0** – дескриптор контекста устройства; **dummy1**, **dummy2**, **dummy3** и **dummy4** – координаты описанного прямоугольника; **dummy5**, **dummy6**, **dummy7** и **dummy8** – координаты начальной и конечной точек дуги эллипса.

logical(4) function Polyline (dummy0, dummy1, dummy2)

Функция рисует ломаную линию, соединяя точки, заданные в массиве. Параметры: **dummy0** – дескриптор контекста устройства, **dummy1** – структура типа T_POINT с координатами, **dummy2** – число точек. Для рисования используется текущее перо. В отличие от функции **PolylineTo** функция **Polyline()** не модифицирует текущую позицию.

logical(4) function PolyTextOut (dummy0, dummy1, dummy2)

Функция выводит несколько строк, используя шрифт и текстовые цвета, выбранные в определенном контексте устройства. Параметры: **dummy0** – дескриптор контекста устройства, **dummy1** – массив структур типа T_POLYTEXT с отдельными строками, **dummy2** – число строк.

logical(4) functionRectangle (dummy0, dummy1, dummy2, dummy3, dummy4)

Функция рисует прямоугольник, используя текущее перо, и заполняет его, используя текущую кисть. Параметры: **dummy0** – дескриптор контекста устройства; **dummy1**, **dummy2**, **dummy3** и **dummy4** – координаты прямоугольника. Текущая позиция не используется и не изменяется.

logical(4) function RoundRect (dummy0, dummy1, dummy2, dummy3, dummy4, dummy5, dummy6)

Функция рисует текущим пером прямоугольник со скругленными углами и заполняет его, используя текущую кисть. Параметры: **dummy0** – дескриптор контекста устройства; **dummy1**, **dummy2**, **dummy3** и **dummy4** – координаты описанного прямоугольника; **dummy5** и **dummy6** – ширина и высота закругления. Текущая позиция не используется и не изменяется.

Integer(4) function SelectObject (dummy0, dummy1)

Функция выбирает объект в текущем контексте устройства. Новый объект заменяет предыдущий объект. Параметры: **dummy0** – контекст

устройства, **dummy1** – дескриптор объекта(растра, кисти, шрифта, пера). Возвращает дескриптор замененного объекта.

integer function SetROP2 (dummy0, dummy1)

Функция устанавливает текущий режим наложения цветов. Параметры: **dummy0** – дескриптор контекста устройства, **dummy1** – режим наложения (**AND**, **OR**, или **XOR**). Все режимы имеют префикс R2_. Возвращает прежнее значение режима.

logical(4) function StretchBlt (dummy0, dummy1, dummy2, dummy3, dummy4, dummy5, dummy6, dummy7, dummy8, dummy9, dummy10)

Функция копирует растровое цветное изображение из исходного прямоугольника в прямоугольник адресата. При необходимости функция изменяет размеры исходного изображения так, чтобы разместить в прямоугольнике адресата. Параметры: **dummy0** – дескриптор контекста устройства адресата; **dummy1** и **dummy2** – координаты верхнего левого угла прямоугольника, в который будет выводиться изображение; **dummy3** и **dummy4** – ширина и высота прямоугольника, в который будет выводится изображение; **dummy5** – дескриптор исходного контекста устройства; **dummy6** и **dummy7** – координаты левого верхнего угла исходного изображения; **dummy8** и **dummy9** – ширина и высота исходного изображения; **dummy10** – способ вывода изображения. Функция растягивает или сжимает исходное изображение в памяти и затем копирует результат в прямоугольник адресата.

logical(4) function TextOut (dummy0, dummy1, dummy2, dummy3, dummy4)

Функция выводит в заданную точку на экране текстовую строку, используя выбранный шрифт. Параметры: **dummy0** – дескриптор контекста устройства; **dummy1** и **dummy2** – координаты начальной позиции; **dummy3** – строка с текстом; **dummy4** – длина строки. Интерпретация начальной точки зависит от текущей установки.

П-12. Функции многодокументного интерфейса

integer(4) function DefFrameProc (hWnd, hWndMDIClient, uMsg, wParam, lParam)

Функция обеспечивает обработку по умолчанию для любых сообщений окна, которые не обработала оконная процедура многодокументного интерфейса (MDI). Все сообщения окна, которые явно не обработаны процедурой окна, должны быть переданы функции **DefFrameProc**, а не функции **DefWindowProc**. Параметры: **hWnd** – дескриптор обрамляющего (главно-

го) окна, **hWndMDIClient** – дескриптор рабочего окна, **uMSG** – сообщение, **wParam** и **lParam** – дополнительные параметры.

integer(4) function DefMDIChildProc (hWnd, uMsg, wParam, lParam)

Функция обеспечивает обработку значения по умолчанию для любого сообщения окна, которые не обрабатывает оконная процедура многодокументного интерфейса (MDI) дочернего окна. Сообщение окна, не обработанное процедурой окна, должно быть передано функции **DefMDIChild Proc()**, а не функции **DefWindowProc()**. Параметры: **hWnd** – дескриптор дочернего окна, **uMSG** – сообщение, **wParam** и **lParam** – дополнительные параметры. Функция воспринимает в качестве родительского окна окно с дескриптором класса **MDICLIENT**.

logical(4) function TranslateMDISysAccel (hWndClient, lpMsg)

Функция преобразует в команды меню сообщения клавиш – ускорителей многодокументного интерфейса. Функция транслирует WM_KEYUP- и WM_KEYDOWN-сообщения в WM_SYSCOMMAND-сообщения и посылает их соответствующим MDI-дочерним окнам. Параметры: **hWndClient** – дескриптор рабочего окна, **lpMSG** – структура типа T_MSG с сообщением.

integer(4) function CreateMDIWindow (lpClassName, lpWindowName, dwStyle, x, y, nWidth, nHeight, hWndParent, hInstance, lParam)

Функция создает дочернее окно– многодокументный интерфейс (MDI). Параметры: **lpClassName** – строка с именем класса, **lpWindowName** – строка с именем окна, **dwStyle** – стиль окна, **x** и **y** – координаты окна, **nWidth** и **nHeight** – размеры окна, **hWndParent** – дескриптор родительского окна, **hInstance** – дескриптор приложения, **lParam** – значение, определяемое программистом. Возвращает дескриптор окна. Использование функции подобно посылке сообщения WM_MDICREATE. Система может поддерживать не более 16 364 дескрипторов окна.

П-13. Функции многопотоковых приложений

logical(4) function CreateProcess (lpApplicationName, lpCommandLine, lpProcessAttributes, lpThreadAttributes, bInheritHandles, dwCreationFlags, lpEnvironment, lpCurrentDirectory, lpStartupInfo, lpProcessInformation)

Функция создает новый процесс, который выполняет заданный исполняемый файл. Параметры: **lpApplicationName** – строка с именем выполняемой программы, **lpCommandLine** – строка с аргументами командной строки, **lpProcessAttributes** и **lpThreadAttributes** – атрибуты доступа(для

Windows NT), **bInheritHandles** – устанавливает наследование дескрипторов, **dwCreationFlags** – флаг, задающий особенности создания нового процесса, **lpEnvironment** – задает параметры среды для нового процесса, **lpCurrentDirectory** – строка, задающая каталог, **lpStartupInfo** – структура типа T_STARTUPINFO с информацией об окне дочернего процесса, **lpProcessInformation** – структура типа T_PROCESS_INFORMATION со значениями идентификаторов процесса.

integer(4) function CreateThread (lpThreadAttributes, dwStackSize, lpStartAddres, lpParameter, dwCreationFlags, lpThreadId)

Функция CreateThread создает поток. Параметры: **lpThreadAttributes** – структура типа T_SECURITY_ATTRIBUTES с атрибутами доступа, **dwStackSize** – размер стека, **lpStartAddress** – адрес потока функции(точки входа), **lpParameter** – параметр для потока, **dwCreationFlags** – флаг состояния потока, **lpThreadId** – идентификатор потока. Возвращает дескриптор потока.

subroutine ExitProcess (uExitCode)

Подпрограмма завершает процесс и все его потоки. Эта функция обеспечивает штатное закрытие процесса. Параметр **uExitCode** – код завершения.

logical(4) function TerminateProcess (hProcess, uExitCode)

Функция завершает определенный процесс и все его потоки. Эта функция управляет процессом извне. Функция обеспечивает немедленное и безусловное завершение процесса и используется только в критических обстоятельствах. Библиотеки DLL, присоединенные к процессу, не получают сообщения о завершении процесса. Параметры: **hProcess** – дескриптор процесса, **uExitCode** – код завершения.

subroutine ExitThread (dwExitCode)

Подпрограмма завершает поток. Обеспечивает штатный выход из потока. Параметр **dwExitCode** – код завершения.

logical(4) function TerminateThread (hThread, dwExitCode)

Функция закрывает поток. Стек не освобождается, а библиотеки DLL, присоединенные к потоку, не оповещаются о его завершении. Это опасная функция, и должна она использоваться только в критических случаях, если точно известно, что именно поток сделает в результате вызова этой функции. Параметры: **hThread** – дескриптор процесса, **dwExitCode** – код завершения.

integer(4) function CreateEvent (lpEventAttributes, bManualReset, bInitialState, lpName)

Функция создает именованное или неименованное событие (объект синхронизации). Параметры: **lpEventAttributes** – структура типа T_SECURITY_ATTRIBUTES с атрибутами доступа, **bManualReset** – тип события, **bInitialState** – начальное состояние, **lpName** – строка с именем события. Возвращает дескриптор объекта.

integer(4) function CreateSemaphore (lpSemaphoreAttributes, lInitialCount, lMaximumCount, lpName)

Функция создает именованный или неименованый объект синхронизации – семафор. Параметры: **lpSemaphoreAttributes** – структура типа T_SECURITY_ATTRIBUTES с атрибутами доступа; **lInitialCount** – начальное число потоков, которые имеют доступ к семафору; **lMaximum Count** – максимальное число потоков, имеющих доступ к объекту; **lpName** – строка с именем события. Возвращает дескриптор объекта.

logical(4) function ReleaseSemaphore (hSemaphore, lReleaseCount, lpPreviousCount)

Функция освобождает семафор для использования его другими потоками. Обычно используется для ограничения потоков, использующих объект. Параметры: **hSemaphore** – дескриптор семафора, **lReleaseCount** – величина сдвига счетчика, **lpPreviousCount** – адрес предыдущего значения счетчика.

integer(4) function WaitForSingleObject (hHandle, dwMilliseconds)

Функция обеспечивает режим ожидания объекта синхронизации (события или семафора) в течение заданного отрезка времени. Параметры: **hHandle** – дескриптор объекта, **dwMilliseconds** – время ожидания в миллисекундах. Возвращает значение, зависящее от события, в течение заданного интервала времени.

integer(4) function OpenSemaphore (dwDesiredAccess, bInheritHandle, lpName)

Функция возвращает дескриптор заданного семафора. Параметры: **dwDesiredAccess** – флаг уровня доступа к семафору, **bInheritHandle** – устанавливает наследование дескрипторв, **lpName** – строка с именем семафора.

integer(4) function OpenEvent (dwDesiredAccess, bInheritHandle, lpName)

Функция возвращает дескриптор события. Параметры: **dwDesired Access** – флаг уровня доступа к событию, **bInheritHandle** – устанавливает наследование дескрипторов, **lpName** – строка с именем события.

integer(4) function GetCurrentThreadId()

Функция возвращает идентификатор потока.

П-14. Функции для работы с файлами

integer(4) function GetFileSize (hFile, lpFileSizeHigh)

Функция возвращает младшее слово размера заданного файла в байтах. Параметры: **hFile** – дескриптор файла, **lpFileSizeHigh** – адрес старшего слова для размера. Если число, равное размеру, полностью размещается в переменной типа **integer(4)**, то **lpFileSizeHigh** = **NULL**. Получение размеров файла этой функцией удобнее, чем средствами Фортрана.

integer(4) function OpenFile (lpFileName, lpReOpenBuff, uStyle)

Функция создает, открывает или удаляет файл. Параметры: **lpFileName** – имя файла, **lpReOpenBuff** – структура типа T_OFSTRUCT с информацией, **uStyle** – тип операции. Возвращает дескриптор файла.

Заключение

Вот мы и добрались с вами до последней страницы. Если ваша работа с книгой шла так, как я это предполагал, то вы приобрели определенную легкость в общении со средствами создания приложений, а также определенный опыт программирования. Более того, теперь в вашем распоряжении есть шаблон приложения с обширным арсеналом элементов управления. Все это фундамент ваших будущих успехов.

Полагаю, что вы поняли, сколь обширны возможности интерфейса API. С моей же точки зрения, как автора, они так обширны и разнообразны, что трудно решить, какие именно средства API надлежит прежде всего описать.

Кроме того, то, что вы прочли, – это лишь видимая часть моей работы. Значительное время ушло на создание шаблона приложения. Все его фрагменты были реально созданы и испытаны. В процессе этой работы пришлось прочитать то самое важное, что написано в любой книге между строк. Чтобы не затягивать дело с изданием, я просто решил остановиться на 12-м уроке.

Смею надеяться, что уж коль вы смогли одолеть все премудрости этой книги, то теперь сможете самостоятельно освоить то, что осталось. Хотя должен вас предупредить, что осталось еще немало.

Что еще вы могли бы все же попробовать?

Мы с вами совершенно не касались графических услуг, которые предоставляют API-функции. Без чрезмерных усилий вы сможете использовать в приложении буфер обмена (***Clipboard***). Если проработать [8, 9], то можно попытаться освоить многозадачный режим работы вашего приложения. Несколько сложнее создать многодокументный интерфейс (MDI), который широко используется в большинстве приложений для Windows. При надлежащем упорстве вы можете обратиться к аппаратным средствам ПК.

В приложении приведено краткое описание некоторых API-функций. Это поможет вам самостоятельно постичь премудрости оконного интерфейса.

Наконец более современные продукты фирмы COMPAG предоставляют разработчикам приложений дополнительные удобства и можно попытаться их освоить.

Перечень того, что еще ждет вас, можно было бы расширить. Однако наилучший выход – написать продолжение этой книги. Если обстоятельства позволят, я попытаюсь это сделать.

Желаю вам успехов.

Литература

1. Горелик А. М., Ушкова В. Л. Фортран сегодня и завтра. – М.: Наука, 1990.
2. Меткалф М., Рид Дж. Описание языка программирования Фортран-90: Пер. с англ. – М.: Мир, 1995.
3. Фролов Ф. В., Фролов Г. В. Программирование для Windows-95 и Windows NT. – М.: Диалог-МИФИ, 1996. – (Библиотека системного программиста; Т. 24). – 288 с.
4. Бартеньев О. В. Современный Фортран. – М.: Диалог-МИФИ, 1998. – 397 с.
5. Бартеньев О. В. Фортран для студентов. – М.: Диалог-МИФИ, 1999. – 400 с.
6. Бартеньев О. В. Visual Fortran: новые возможности. – М.: Диалог-МИФИ, 1999. – 301 с.
7. Березин Б. Н., Березин С. Б. Начальный курс C и C++. – М.: Диалог-МИФИ, 1999. – 288 с.
8. Шилд Г. Программирование на C и C++ для Windows-95. – Киев: Торгово-издательское бюро BHV, 1996. – 399 с.
9. Румянцев П. В. Азбука программирования в Win32 API. – М.: Радио и связь, 1998. – 272 с.

Содержание