Министерство образования Республики Беларусь

Учреждение образования
«Гомельский государственный технический
университет имени П. О. Сухого»

Институт повышения квалификации и переподготовки

Кафедра «Профессиональная переподготовка»

Е. И. Гридина

СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

ПОСОБИЕ
по одноименной дисциплине для слушателей специальности 1-40 01 73 «Программное обеспечение информационных систем» заочной формы обучения

Гомель 2017

УДК 004.65(075.8)
ББК 32.973-018.2я73
Г83

Рекомендовано кафедрой «Профессиональная переподготовка»
ИПКиП ГГТУ им. П. О. Сухого
(*протокол № 1 от 20.09.2016 г.*)

Рецензент: канд. техн. наук, доц. каф. «Информатика» УО «Гомельский государственный технический университет имени П. О. Сухого» *Т. А. Трохова*

Гридина, Е. И.

Г83 Системное программирование : пособие по одноим. дисциплине для слушателей специальности 1-40 01 73 «Программное обеспечение информационных систем» заоч. формы обучения / Е. И. Гридина. – Гомель : ГГТУ им. П. О. Сухого, 2017. – 69 с. – Систем. требования: PC не ниже Intel Celeron 300 МГц ; 32 Mb RAM ; свободное место на HDD 16 Mb ; Windows 98 и выше ; Adobe Acrobat Reader. – Режим доступа: https://elib.gstu.by. – Загл. с титул. экрана.

Данное издание представляет собой пособие по ознакомлению с основами системного программирования, рассмотрены графический интерфейс Windows-приложения, элементы управления, растровая графика, процессы и потоки, DLL-библиотеки. Пособие написано доступным языком и содержит описание сигнатур используемых функций, составляющих библиотеку Win32 API и позволяющих получить доступ к системным ресурсам компьютера.

Издание адресовано слушателям ИПКиП специальности 1-40 01 73 «Программное обеспечение информационных систем».

УДК 004.65(075.8)
ББК 32.973-018.2я73

Содержание

Современные инструментальные средства разработки системно-ориентированных приложений.

В операционной системе Windows между приложением и совокупностью системных вызовов (системных сервисов в терминологии Microsoft) расположен дополнительный абстрактный слой – программный интерфейс Win32 API. За счет этого Win32-приложение может работать практически во всех версиях Windows, несмотря на то, что сами системные вызовы в различных версиях системы различны и не документированы.

Для создания Windows-приложений на C++ можно использовать среду разработки Microsoft Visual Studio 2010/2013/2015. Express.

Исчерпывающая информация по программному интерфейсу Win32 API содержится в справочной документации на Win32 API. Эту документацию можно просмотреть на сайте http://msdn.microsoft.com или на компакт-дисках MSDN (Microsoft Developer Network Library). MSDN является программой технической поддержки разработчиков.

Прикладной программный интерфейс

API-интерфейс Win32 (также известный как Windows API) – это платформа на основе C для создания приложений Windows.

Программный интерфейс приложения (API) предоставляет услуги, используемые всеми приложениями Windows. Используя возможности API можно разработать приложения с графическим пользовательским интерфейсом; системы доступа к таким ресурсам, как память и устройства, отображать графику и форматированный текст; включать аудио, видео, сети, безопасность.

Механизм сообщений в операционных системах

Каждое окно, создаваемое Windows, имеет свой собственный цикл сообщений и свою функцию обработки сообщений (оконную процедуру), предназначенную для обработки сообщений именно этого окна. Когда главный цикл сообщений приложения вызывает DispatchMessage(), Windows передает сообщение оконной процедуре приложения. Оконная процедура – это функция, обычно называемая WndProc(), которая

обрабатывает сообщения данного окна. Именно в этот момент программам решает, что делать с информацией, содержащейся в структуре MSG обрабатываемого сообщения.

Вот так определяется оконная процедура:

LRESULT CALLBACK WndProc(HWND hWnd, UINT message, PARAM wParam, LPARAM lParam);

Аргументы функции WndProc() представляют собой первые четыре члена структуры MSG сообщения, которое в данный момент обрабатывается. Эти параметры были извлечены и посланы оконной процедуре благодаря функции DispatchMessage(). Все сообщения приложению передаются из цикла соощений главной оконной процедуре приложения.

Оконная процедура обрабатывает сообщения, поступающие окну. Очень часто эти сообщения передают окну информацию о том, что пользователь осуществил ввод с помощью клавиатуры или мыши. Таким образом, например, кнопки “узнают” о том, что они нажаты. Другие сообщения говорят окну о том, что необходимо изменить размер окна или о том, что поверхность окна необходимо перерисовать.

Когда программа для Windows начинает выполняться, Windows строит для программы очередь сообщений. В этой очереди хранятся сообщения для любых типов окон, которые могли бы быть созданы программой. Небольшая часть программы, которая называется циклом обработки сообщений, выбирает эти сообщения из очереди и перенаправляет их соответствующей оконной процедуре. Другие сообщения отправляются непосредственно оконной процедуре, минуя очередь сообщений.

Основной цикл сообщений выглядит так:

```
while (GetMessage(&msg,0,0,0))
{
  TranslateMessage(&msg);
   DispatchMessage(&msg);
}
```

Цикл сообщений – это обыкновенный цикл while(), продолжающийся до тех пор, пока программа не получит сообщение о завершении. Внутри цикла while() на каждой итерации вызывается Windows API – функция GetMessage(). Прототип функции GetMessage() объявлен в файле WINDOWS.H и выглядит так:

BOOL GetMessage(MSG FAR* lpmsg, HWND hwnd, UINT uMsgFilterMin, UINT uMsgFilterMax).

Функция GetMessage() извлекает следующее сообщение из очереди и хранит его для дальнейшего использования программой в особой структуре MSG. Цикл сообщений продолжает опрашивать очередь до тех пор, пока не получит сообщение WM_QUIT (0x0012).

API-функции TranslateMessage() и DispatchMessage() принимают указатель на структуру MSG в качестве единственного аргумента. TranslateMessage() осуществляет трансляцию ввода с клавиатуры. DispatchMessage() посылает сообщение процедуре обработки сообщений в окне – оконной процедуре.

Сообщения, посылаемые окну, могут быть синхронными и асинхронными.

Синхронные сообщения – это сообщения, которые Windows помещает в очередь сообщений приложения. В цикле очередное синхронное сообщение выбирается из очереди функцией GetMessage, затем передается Windows посредством функции DispatchMessage, после чего Windows отправляет синхронное сообщение оконной процедуре для последующей обработки.

В отличие от синхронных сообщений асинхронные сообщения передаются непосредственно оконной процедуре для немедленной обработки, минуя очередь сообщений.

К синхронным сообщениям относятся сообщения о событиях пользовательского ввода, например, клавиатурные сообщения и сообщения мыши. Кроме этого синхронными являются сообщения от таймера (WM_TIMER), сообщение о необходимости перерисовки клиентской области окна (WM_PAINT) и сообщение о выходе из программы (WM_QUIT). Остальные сообщения, как правило, являются асинхронными. Во многих случаях асинхронные сообщения являются результатом обработки синхронных сообщений. Например, когда WinMain вызывает функцию CreateWindow, Windows создает окно и для этого отправляет оконной процедуре асинхронное сообщение WM_CREATE. Когда WinMain вызывает ShowWindow, Windows отправляет оконной процедуре асинхронные сообщения WM_SIZE и WM_SHOWWINDOW. Когда WinMain вызывает UpdateWindow, Windows отправляет оконной процедуре асинхронное сообщение WM_PAINT.

Посылка сообщений

Существует множество задач, для решения которых необходимо уметь «вручную» посылать сообщения окнам, не дожидаясь их от

операционной системы. Для этой цели используются функции API SendMessage и PostMessage.

```
LRESULT SendMessage(
HWND hWnd,
// дескриптор окна, которому отправляется сообщение
UINT Msg, // идентификатор сообщения
WPARAM wParam, // дополнительная информация о сообщении
LPARAM lParam  // дополнительная информация о сообщении
)

BOOL PostMessage(
HWND hWnd,
// дескриптор окна, которому отправляется сообщение
UINT Msg, // идентификатор сообщения
WPARAM wParam, // дополнительная информация о сообщении
LPARAM lParam  // дополнительная информация о сообщении
);
```

Как видно, обе функции имеют одинаковую сигнатуру, а их сходство состоит в том, что они предназначены для посылки сообщения в окно некоторого приложения. Однако вышеприведенные функции имеют определённые отличия.

Функция SendMessage вызывает оконную процедуру определенного окна и не завершает свою работу до тех пор, пока оконная процедура не обработает сообщение. Иначе говоря, выполнение программы не будет продолжено, пока сообщение не будет обработано. Оконная процедура, которой отправляется сообщение, может быть той же самой оконной процедурой, другой оконной процедурой той же программы или даже оконной процедурой другого приложения. Таким образом, функция SendMessage посылает асинхронное сообщение указанному окну.

Функция PostMessage посылает синхронное сообщение указанному окну. В отличие от SendMessage, функция PostMessage не вызывает явно оконную процедуру, а всего лишь помещает сообщение в очередь сообщений. Выполнение самой функции на этом завершается, и работа приложения может быть продолжена.

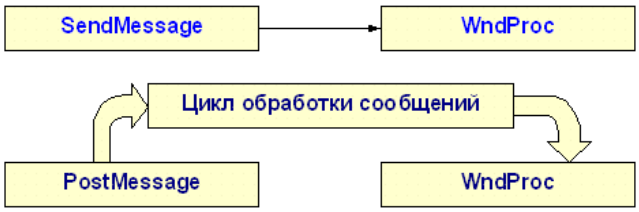

Рисунок 1 – Цикл обработки сообщений

Следует отметить, если первый параметр вышерассмотренных функций имеет значение HWND_BROADCAST, то сообщение посылается всем окнам верхнего уровня, существующим в настоящий момент в системе.

Организация графического пользовательского интерфейса в операционных системах

Окно приложения обычно содержит заголовок, меню, рамку и иногда полосы прокрутки. Окна диалога – это дополнительные окна. Больше того, в окне диалога всегда имеется еще несколько окон, называемых “дочерними”. Эти дочерние окна имеют вид кнопок, переключателей, флажков, полей текстовго ввода или редактирования, списков и полос прокрутки.

Окно – это объект. Код – это оконная процедура. Данные – это информация, хранимая оконной процедурой, и информация, хранимая системой Windows для каждой окна и каждого окна и каждого класса окна, которые имеются в системе.

Пользователь рассматривает окна на экране в качестве объектов и непосредственно взаимодействует с этими объектами, нажимая кнопки и переключатели, передвигая бегунок на полосах прокрутки. Положение программиста аналогично положению пользователя. Окно получает от пользователя информацию в виде оконных “сообщений”. Кроме этого окно обменивается сообщениями с другими окнами.

В обычной Dos-программе, написанной на Си, функция main() является начальной точкой входа в программу. Программы для Windows также имеют точку входа, и Windows ищет функцию WinMain() именно как начальную точку входа приложения в период выполнения. Фунция WinMain() имеет следующий прототип:

int FAR PASCAL WinMain (HINSTANCE hInst, HINSTANCE hPrevInst, LPSTR lpCmdLine, int CmdShow)

Существуют разные формы этого прототипа в Си-программах для Windows. Эквивалентный прототип функции WinMain() может выглядеть так:

int APIENTRY WinMain (HINSTANCE hInst, HINSTANCE hPrevInst, LPSTR lpCmdLine, int CmdShow)

Внутренняя структура этих прототипов одинакова.

Действия, которые вы программируете в функции WinMain(), могут быть настолько простыми или же настолько сложными, насколько позволяет ваше умение, но в любом случае эта функция должна выполнить следующее:

- инициализировать приложение;
- инициализировать и создать окна приложения;
- войти в цикл сообщений приложения.

Диалоговые окна

Диалоговые окна, или окна диалога (dialog box), реализуют одну из важнейших составляющих программирования Windows-приложений. Обычно диалоговые окна используются для получения от пользователя дополнительной информации, а также для вывода результатов работы приложения.

Диалоговое окно имеет вид всплывающего окна с одним или несколькими элементами управления (controls), которые являются для него дочерними окнами. Используя элементы управления, пользователь вводит текст, выбирает указанные опции (флажки, переключатели, элементы списка) и нажимает кнопки, вызывающие различные действия приложения.

От обычных окон диалоговые окна отличаются тем, что они создаются на базе предопределенного в Windows класса диалоговых окон. Оконная процедура этого класса, спрятанная в недрах Windows, обеспечивает обработку сообщений, поступающих в диалоговое окно, а также задает специфическое поведение элементов управления диалогового окна. Например, она управляет передачей фокуса ввода от одного элемента другому или переносит фокус ввода между группами элементов при нажатии клавиши Tab. Эту невидимую для программиста оконную процедуру иногда называют менеджером диалогового окна (dialog box manager).

Менеджер диалогового окна передает многие сообщения в функцию, определенную в вашем приложении, которая называется процедурой диалогового окна, или просто диалоговой процедурой (dialog procedure).

Эта процедура похожа на обычную оконную процедуру, но все же имеет некоторые важные особенности.

Второе отличие диалоговых окон от обычных окон состоит в том, что они всегда связаны с шаблоном диалога, содержащим размеры окна, состав и расположение элементов управления. Шаблон диалогового окна можно определить двумя способами: а) в файле описания ресурсов, используя редактор диалоговых окон, б) создавая шаблон в памяти в процессе работы приложения.

Типы диалоговых окон

Диалоговые окна бывают модальные (modal) и немодальные (modeless). Они различаются по влиянию на дальнейшее выполнение программы.

Когда в программе вызывается модальное диалоговое окно, оно ожидает выполнения некоторого действия со стороны пользователя, прежде чем программа сможет продолжить свое выполнение. Пользователь не может переключиться между диалоговым окном и другими окнами программы. Он должен явно закрыть диалоговое окно, что обычно делается щелчком на кнопке ОК или Cancel. Однако пользователь может переключаться в другие программы, не закрыв диалоговое окно.

Существует также специальный вид модальных диалоговых окон – системные модальные (system modal) окна, которые не позволяют переключаться даже в другие программы. Они сообщают о серьезных проблемах, и пользователь должен закрыть системное модальное окно, чтобы продолжить работу в Windows.

Немодальное диалоговое окно не приостанавливает выполнение программы. Оно может получать и терять фокус ввода. Это значит, что пользователь может свободно переключаться между диалоговым окном и другими окнами программы. Окна этого типа предпочтительней использовать в тех случаях, когда они содержат элементы управления, которые должны быть в любой момент доступны пользователю.

Элементы управления в диалоговом окне

Основную функциональную нагрузку в диалоговом окне выполняют элементы управления. Все версии Windows поддерживают так называемые базовые элементы управления.

Обычно элементы управления определяются в шаблоне диалогового окна на языке описания шаблона диалога. В случае создания шаблона с

помощью редактора диалоговых окон эти определения элементов генерируются автоматически.

Одним из атрибутов описания элемента управления в шаблоне диалога является идентификатор элемента управления.

Каждый элемент управления, описанный в шаблоне диалога, реализуется Windows в виде окна соответствующего класса. Это окно является дочерним окном по отношению к диалоговому окну. Как всякое окно, оно идентифицируется своим дескриптором типа HWND. Вместо термина «дескриптор окна элемента управления» в документации (MSDN) обычно используется более короткий термин «дескриптор элемента управления».

Если элемент управления определен в шаблоне диалога, то программисту известен только его идентификатор. В то же время многие функции, работающие с элементом управления, принимают в качестве параметра его дескриптор. Для получения дескриптора элемента управления по его идентификатору используется функция GetDlgltem.

```
HWND GetDlgltem (
HWND hDlg. // дескриптор диалогового окна
int nIDDlgltem // идентификатор элемента управления
);
```

В случае успешного завершения функция возвращает дескриптор элемента управления, в случае ошибки — значение NULL

Иногда возникает необходимость обратного преобразования, чтобы по дескриптору элемента управления определить его идентификатор. Такую проблему решает вызов функции GetDlgCtrLTD.

```
int GetDlgCtrllD (
HWND hwndCtl // дескриптор элемента управления
);
```

Элементы управления обычно определяются в шаблоне диалогового окна. Существует, однако, и альтернативный способ создания и размещения элемента управления при помощи функции CreateWindiow, первому параметру которой передается имя предопределенного оконного класса.

Элементы управления могут быть разрешенными (enabled) или запрещенными (disabled). По умолчанию все элементы управления имеют статус разрешенных элементов. Запрещенные элементы выводятся на

экран серым цветом и не воспринимают пользовательский ввод с клавиатуры или от мыши. Изменение статуса элементов управления осуществляется при помощи функции EnableWindow.

Создание и обработка диалогового окна

Создание диалогового окна и работа с ним требуют выполнения следующей последовательности действий:

– определение шаблона диалогового окна;

– определение диалоговой процедуры;

– вызов функции создания диалогового окна;

– обмен данными между диалоговой процедурой и вызывающей функцией окна.

Реализация некоторых шагов различается для модальных и немодальных диалоговых окон.

Диалоговая процедура

Диалоговая процедура DlgProc во многом напоминает оконную процедуру. Она должна иметь спецификатор CALLBACK, поскольку вызывается операционной системой. Имя функции может быть произвольным, однако сложилась традиция завершать это имя суффиксом DlgProc.

Диалоговая процедура принимает такой же набор параметров, что и обычная оконная процедура. Но некоторые различия между диалоговой процедурой и оконной процедурой все же есть:

– оконная процедура возвращает значение типа LRESULT, а диалоговая процедура — значение типа B00L;

– если оконная процедура не обрабатывает какое-то сообщение, то она вызывает DefWindowProc. Если диалоговая процедура не обрабатывает какое-то сообщение, то она возвращает значение FALSE. Если же сообщение обрабатывается диалоговой процедурой, то она возвращает значение TRUE;

– диалоговая процедура не обрабатывает сообщение WM_CREATE. Вместо этого она выполняет инициализацию при обработке специального сообщения WM_INITDIAL0G;

– диалоговая процедура обычно не обрабатывает сообщение WM_PAINT, так как все функции диалогового окна реализуются элементами управления.

Сообщение WM_INITDIALOG является первым сообщением, которое получает диалоговая процедура. Если после обработки этого сообщения процедура возвращает значение TRUE, то Windows помещает фокус ввода на первое дочернее окно элемента управления, которое имеет стиль WS_TABST0P. В то же время при обработке сообщения WM_INITDIALOG диалоговая процедура может использовать функцию SetFocus для того, чтобы установить фокус на какой-то другой элемент управления. Но тогда она должна вернуть значение FALSE.

Блок обработки сообщения WM_INITDIALOG является самым удобным местом для инициализации элементов управления, если в этом есть необходимость.

Основным сообщением, обрабатываемым в диалоговой процедуре, является WM_COMMAND. Напомним, что если источником сообщения WM_COMMAND является элемент управления, то младшее слово параметра wParam содержит идентификатор элемента управления, старшее слово wParam содержит код уведомления, а параметр lParam — дескриптор элемента управления.

Функция EndDialog закрывает модальное диалоговое окно.

```
BOOL EndDialog (
HWNO hDlg, // дескриптор диалогового окна
INT_PTR nResult // значение, возвращаемое из функции DialogBox
);
```

Второй параметр функции EndDialog задает значение, которое передается функции DialogBox для использования в качестве кода возврата из функции DialogBox. Чаще всего функции EndDialog передается в параметре nResult значение TRUE при обработке команды IDOK и FALSE — при обработке команды IDCANCEL.

Немодальный диалог

Немодальные диалоговые окна позволяют пользователю переключаться между диалоговым окном и окном, в котором оно было создано. Окна этого типа больше напоминают обычные всплывающие окна, которые могут создаваться программой.

Немодальные диалоговые окна лучше использовать, когда пользователь работает с ними и с основным окном одновременно. Пожалуй, самые распространенные немодальные окна — это окна инструментов поиска Find и Replace, отображаемые программами

обработки текстов. Пользователю может потребоваться оставить такой диалог в активном состоянии, если нужно выполнить несколько операций поиска и замены. При отображении этого окна можно также редактировать документ, в котором выполняется поиск или замена.

Иногда немодальное диалоговое окно создается в момент старта приложения и может оставаться на экране до окончания работы приложения.

Различия между модальными и немодальными окнами диалога

Модальные диалоговые окна создаются при помощи функции DialogBox. Эта функция возвращает управление только после закрытия диалогового окна. Немодальные диалоговые окна создаются с помощью функции CreateDiaLog. Она принимает такие же параметры, что и функция DialogBox:

hDlgModeless = CreateDialog (hlnst. MAKEINTRESOURCE(IDD_DLG), hWnd, DlgProc);

Функции передаются дескриптор экземпляра приложения, идентификатор шаблона диалога, дескриптор родительского окна и адрес диалоговой процедуры.

Различие состоит в том, что функция CreateDiaLog сразу возвращает дескриптор диалогового окна. Как правило, этот дескриптор хранится в глобальной переменной.

Определяя свойства шаблона диалогового окна, обязательно установите флажок Visible на вкладке More Styles окна Dialog Properties. Если этот флажок сброшен, то для появления окна на экране потребуется после вызова функции CreateDiaLog дополнительно вызвать функцию ShowWindow: ShowWindow(hDlgModeless. SW_SHOW);

При сброшенном флажке Visible и отсутствии указанного вызова функции ShowWindow немодальный диалог вообще не появится на экране. Модальное диалоговое окно менее притязательно: оно появляется на экране и при сброшенном флажке Visible.

В отличие от сообщений для модальных диалоговых окон, направляемых системой непосредственно менеджеру диалогового окна, сообщения для немодального диалогового окна проходят через очередь сообщений программы. Поэтому цикл обработки сообщений в теле функции WinMain должен иметь вид.

```
while (GetMessage(&msg. NULL. 0. 0)) {
if (!IsDialogMessage(hModelessDlg. &msg)) {
```

```
        TranslateMessage(&msg);
        DispatchMessage(&msg);
        }
}
```

Функция IsDialogMessage определяет, относится ли сообщение, сохраненное в переменной msg, к диалоговому окну hModelessDlg. Если да, то функция обрабатывает это сообщение, отправляя его диалоговой процедуре и возвращая значение TRUE. В ином случае функция ничего не делает с сообщением и возвращает значение FALSE.

Сообщения, переданные на обработку диалоговой процедуре, не должны обрабатываться функциями TranslateMessage и DispatchMessage. Это обеспечивает условный оператор if, анализирующий код возврата функции IsDialogMessage.

Заметим, что если немодальное диалоговое окно еще не создано, то дескриптор hModelessDlg должен быть равен нулю и в этом случае функция IsDialogMessage тоже возвращает значение FALSE.

Для закрытия немодального диалогового окна вместо функции End Dialog вызывается функция DestroyWindow. Если диалог закрывается, а приложение продолжает работать, то рекомендуется дескриптор hModelessDlg установить в нулевое значение.

Если немодальное диалоговое окно имеет заголовок, в котором размещается кнопка закрытия окна, то пользователь по привычке может попытаться закрыть диалог с помощью данной кнопки. Менеджер немодального диалогового окна не обрабатывает сообщение WM_CLOSE. Поэтому реакция приложения будет зависеть от решения программиста. Если вы считаете, что окно можно закрыть, то в диалоговую процедуру нужно добавить следующий код:

```
case WM_CLOSE:
DestroyWindow(hDlg);
hModelessDlg = 0:
break;
```

Это же сообщение будет обрабатываться в случае выбора команды Close (Alt+F4) в системном меню диалогового окна.

Класс окна в операционных системах, предопределенные классы, получение и изменение данных окна и класса

Структура WNDCLASS используется для описания всех окон приложений.

```
typedef struct WNDCLASS
{
UINT          style;          //стиль класса окна
WNDPROC       lpfnWndProc;    //оконная процедура
INT           cbClsExtra;     //дополнительные байты класса
INT           cbWndExtra;     //дополнительные байты окна
HANDLE        hInstance;      //дескриптор экземпляра окна
HICON         hIcon;          //дескриптор пиктограммы
HCURSOR       hCursor;        //дескриптор курсора
HBRUSH        hbrBackground;  //фон(цвет)
LPCTSTR       lpszMenuName;   //ресурс меню
LPCTSTR       lpszClassName;  //имя класса окна
}WNDCLASS;
```

lpszClassName – указатель на строку, содердащую имя класса. Поскольку определенный в приложении класс доступен всем приложениям, имя класса не должно повторяться в разных приложениях.

hInstance – манипулятор копии, создающей класс окна.

lpfnWndProc – указатель на функцию поддержки окна.

style - определяет свойства окна.

hbrBackground – определяет фон окна.

hCursor – определяет курсор, используемый в данном окне по умолчанию.

hIcon – определяет пиктограмму, которая будет отображаться при переводе окна в неактивное состояние.

lpszMenuName – указатель на имя меню окна, определенное в файле ресурсов.

cbClsExtra – определяет число байт, которое необходимо дополнительно запросить у Windows под эту структуру.

cbWndExtra – определяет число байт, которое необходимо дополнительно запросить у Windows для размещения всех структур, создаваемых совместно с данным классом.

Ресурсы операционных систем, их создание и использование

Операционная система Windows по сравнению с операционными системами типа MS-DOS обладает серьезными преимуществами и для

пользователей, и для программистов. Среди этих преимуществ обычно выделяют:

- графический интерфейс пользователя;
- многозадачность и многопоточность;
- управление памятью;
- независимость от аппаратных средств.

Графический интерфейс пользователя GUI (Graphical User Interface) дает возможность пользователям работать с приложениями максимально удобным способом. Стандартизация графического интерфейса имеет очень большое значение для пользователя, потому что одинаковый интерфейс экономит его время и упрощает изучение новых приложений. С точки зрения программиста, стандартный вид интерфейса обеспечивается использованием подпрограмм, встроенных непосредственно в Windows, что также приводит к существенной экономии времени при написании новых программ.

Многозадачные операционные системы позволяют пользователю одновременно работать с несколькими приложениями или несколькими копиями одного приложения. Многозадачность осуществляется в Windows при помощи процессов и потоков. Любое приложение Windows после запуска реализуется как процесс. Процесс можно представить как совокупность программного кода и выделенных для его исполнения системных ресурсов. При инициализации процесса система всегда создает первичный (основной) поток, который исполняет код программы, манипулируя данными в адресном пространстве процесса.

Память – это один из важнейших разделяемых ресурсов в операционной системе. Если одновременно запущены несколько приложений, то они должны разделять память, не выходя за пределы выделенного адресного пространства. Так как одни программы запускаются, а другие завершаются, то память фрагментируется. Система должна уметь объединять свободное пространство памяти, перемещая блоки кода и данных. Операционная система Windows обеспечивает достаточно большую гибкость в управлении памятью. Если объем доступной памяти меньше объема исполняемого файла, то система может загружать исполняемый файл по частям, удаляя из памяти отработавшие фрагменты. Если пользователь запустил несколько копий, которые также называют отдельными экземплярами приложения, то система размещает в памяти только одну копию исполняемого кода, которая используется этими экземплярами совместно. Программы, запущенные в Windows, могут использовать также функции из библиотек динамической компоновки – DLL (dynamic link libraries). Windows поддерживает

механизм связи программ во время их работы с функциями из DLL. Даже сама операционная система Windows, по существу, является набором динамически подключаемых библиотек. Эти библиотеки содержат набор функций WinAPI, позволяющих программисту создавать приложения, работающие под управлением Windows.

Еще одним преимуществом Windows является независимость от используемой платформы. У программ, написанных для Windows, нет прямого доступа к аппаратной части таких устройств отображения информации, как, например, экран и принтер. Вместо этого они вызывают функции графической подсистемы WinAPI, называемой графическим интерфейсом устройства (Graphics Device Interface, GDI). Функции GDI реализуют основные графические команды при помощи обращения к программным драйверам соответствующих аппаратных устройств. Одна и та же команда (например, LineTo – нарисовать линию) может иметь различную реализацию в разных драйверах. Эта реализация скрыта от программиста, использующего WinAPI, что упрощает разработку приложений. Таким образом, приложения, написанные с использованием WinAPI, будут работать с любым типом дисплея и любым типом принтера, для которых имеется в наличии драйвер Windows. То же самое относится и к устройствам ввода данных – клавиатуре, манипулятору «мышь» и т.д. Такая независимость Windows от аппаратных средств достигается благодаря указанию требований, которым должна удовлетворять аппаратура, в совокупности с SDK (Software Development Kit – набор разработки программ) и/или DDK (Driver Development Kit – набор разработки драйверов устройств). Разработчики нового оборудования поставляют его вместе с программными драйверами, которые обязаны удовлетворять этим требованиям.

Организация интерфейса на основе меню

Работа с меню в Windows – одно из самых простых и понятных мест. Для создания меню вам просто нужно:

– задать структуру меню в файле ресурсов, последовательно определив пункты меню в виде текстовых строк;

– каждому пункту меню поставить в соответствие уникальный идентификатор;

– указать имя меню в структуре класса окна.

Когда пользователь выбирает пункт меню, Windows передает приложению сообщение WM_COMMAND, содержащее идентификатор выбранного пункта, так что после определения структуры меню в файле

ресурсов нужно оформить собственно текст программы – ввести обработку сообщения WM_COMMAND.

Для создания меню можно использовать три метода.

Во-первых, можно описать шаблон меню в файле ресурсов приложения. Этот способ больше всего подходит для создания статических меню, не меняющихся или меняющихся не очень сильно в процессе работы приложения.

Во-вторых, можно создать меню “с нуля” при помощи специальных функций программного интерфейса Windows. Этот способ хорош для приложений, меняющих внешний вид меню, когда вы не можете создать заранее подходящий шаблон. Разумеется, второй метод пригоден и для создания статических меню.

В-третьих, можно подготовить шаблон меню непосредственно в оперативной памяти и создать меню на базе этого шаблона.

Определение меню приложения, включенное в файл ресурсов, может быть создано как вручную, так и при помощи редактора меню, входящего в состав компилятора C++.

Для этого необходимо при создании проекта выбрать пункт меню Insert -> Resource (Ctrl+R) и в предложенном списке ресурсов выбрать необходимый (для создания меню – Menu) и нажать New. Далее рисуем необходимое нам меню. Строка меню выводится на экране непосредственно под строкой заголовка. Эта строка иногда называется главным меню (main menu) или меню верхнего уровня (top-level menu) программы. Выбор элемента главного меню обычно приводит к вызову другого меню, появляющегося под главным, и которое обычно называют всплывающим меню (popup menu) или подменю (submenu). Вы также можете определить несколько уровней вложенности всплывающего меню (pop_up), т.е. определенный пункт всплывающего меню может вызвать появление другого всплывающего меню.

Пункты всплывающих меню могут быть помечены (checked), при этом слева от текста элемента меню Windows ставится специальная метка или “галочка” . Галочки позволяют пользователю узнать о том, какие опции программы выбраны из этого меню. Эти опции могут быть взаимоисключающими. Пункты главного меню помечены быть не могут.

Пункты меню в главном и всплывающих меню могут быть “разрешены” (enabled), “запрещены” (disabled) или “недоступны” (grayed). Слова “активно” (active) и “неактивно” (inactive) иногда используются, как синонимы слов “разрешено” и “запрещено”. При выборе элемента separator в всплывающем меню рисуется горизонтальная черта. Эта черта часто используется для разделения групп, связанных по

смыслу и назначению опций. Опции grayed и inactive нельзя использовать вместе. При выборе опции help – данный пункт меню, а также все последующие появляются на экране, выравненные по правому краю.

Во многих приложениях Windows в описании ресурсов имеется только одно меню. Ссылка в программе на это меню имеет место в определении класса окна: wndclass.lpszMenuName = MAKEINTRESOURSE(IDR_MENU1), где IDR_MENU1 – имя созданного вами меню, которое определено в подключаемом файле (в нашем случае resourse.h), как #define IDR_MENU1 101, а MAKEINTRESOURSE позволяет преобразовать число к типу в строку для использования в качестве имени ресурса.

Значения идентификаторов в инструкциях MENUITEM – это числа, которые Windows посылает оконной процедуре в сообщениях меню. Значение идентификатора должно быть уникальным в рамках меню. Вместо чисел может оказаться удобнее использовать идентификаторы, определенные в заголовочном файле. По договоренности эти идентификаторы начинаются с символов IDM (идентификатор меню).

Когда пользователь выбирает пункт меню, Windows посылает оконной процедуре несколько различных сообщений. Большинство из этих сообщений могут игнорироваться программой, и просто передаваться DefWindowProc.

Самым важным сообщением меню является WM_COMMAND. Это сообщение показывает, что пользователь выбрал разрешенный пункт меню окна. Если оказывается, что для меню и дочерних окон управления используются одни и те же идентификаторы, то различить их можно с помощью параметра lParam, который для пункта меню будет равен 0.

Обработка сообщений меню

Операционная система Windows посылает сообщение WM_COMMAND при каждом выборе пункта меню, определяющего команду. При этом LOWORD(wParam) содержит идентификатор пункта меню, а HIWORD(wParam) и lParam содержат нулевые значения.

Чаще всего WM_COMMAND – единственное сообщение, обрабатываемое приложением, которое может поступить от меню. При выборе пунктов системного меню вместо указанного сообщения отправляется сообщение WM_SYSCOMMAND.

Иногда в программе может потребоваться обработка сообщений WM_INITMENU и WM_INITMENUPOPUP. Они отправляются непосредственно перед активизацией главного меню или выпадающего

меню. Эти сообщения позволяют приложению изменить меню перед тем, как оно будет отображено на экране.

При навигации по меню система отправляет также сообщение WM_MENUSELECT. Оно более универсально по сравнению с WM_COMMAND, так как инициируется даже тогда, когда выделен недоступный или запрещенный пункт меню. Это сообщение может использоваться для формирования контекстной справки меню, которая отображается в строке состояния приложения.

Для изменения статуса пунктов меню применяется функция API EnableMenuItem.

```
BOOL EnableMenuItem(
HMENU hMenu, // дескриптор меню
UINT uIDEnableItem,
// идентификатор или позиция пункта меню
UINT uEnable
// интерпретация второго параметра и выполняемое действие
);
```

Пункт меню, для которого применяется функция, задается вторым параметром uIDEnableItem. Этому параметру передается либо идентификатор пункта меню, либо позиция пункта меню. Выбор варианта интерпретации задается в третьем параметре. Третий параметр uEnable задается как побитовая операция объединения двух флагов. Первый флаг может содержать одно из следующих значений:

– MF_BYCOMMAND – в этом случае второй параметр должен содержать идентификатор пункта меню;

– MF_BYPOSITION – в этом случае второй параметр должен содержать относительную позицию пункта меню с отсчетом от нуля.

Второй флаг может принимать одно из следующих значений:

– MF_ENABLED – пункт меню разрешён;

– MF_DISABLED – пункт меню запрещён;

– MF_GRAYED – пункт меню недоступен.

Следует отметить, если в результате применения функции изменяется статус пункта главного меню, то следует обязательно вызвать функцию API DrawMenuBar для повторного отображения изменившейся полосы меню.

```
BOOL DrawMenuBar(
HWND hWnd // дескриптор главного окна приложения
```

);

В коде приложения может использоваться функция API CheckMenuItem, позволяющая установить или снять отметку на пункте меню.

```
DWORD CheckMenuItem(
HMENU hMenu,  // дескриптор меню
UINT uIDCheckItem, // идентификатор или позиция пункта меню
UINT uCheck
// интерпретация второго параметра и выполняемое действие
);
```

Второму параметру функции uIDCheckItem передается либо идентификатор пункта меню, либо позиция пункта меню. Выбор варианта интерпретации задается в третьем параметре. Третий параметр uCheck задается как побитовая операция объединения двух флагов. Первый флаг может содержать одно из следующих значений:

– MF_BYCOMMAND – в этом случае второй параметр должен содержать идентификатор пункта меню;

– MF_BYPOSITION – в этом случае второй параметр uIDCheckItem должен содержать относительную позицию пункта меню с отсчетом от нуля.

Второй флаг может принимать одно из следующих значений:

– MF_CHECKED – поместить отметку слева от имени пункта меню;

– MF_UNCHECKED – снять отметку слева от имени пункта меню.

В функциях CheckMenuItem и EnableMenuItem следует использовать флаг MF_BYCOMMAND, передавая второму параметру идентификатор пункта меню. Это предотвратит возможные проблемы в процессе сопровождения программы, если потребуется модификация меню, изменяющая относительные позиции пунктов меню.

Необходимо отметить, что функцию CheckMenuItem можно применять только для пунктов подменю. Пункты главного меню не могут быть помечены.

Сообщение WM_MENUSELECT удобно использовать для формирования контекстной справки меню, которая отображается в строке состояния приложения. Для отображения текста контекстной справки в строке состояния может быть использован ресурс таблицы строк. Таблица строк представляет собой список строк, связанных с уникальными целочисленными идентификаторами.

Для удобства формирования контекстной справки меню каждой строке сопоставлется уже существующий идентификатор одного из пунктов меню. Это позволяет при обработке сообщения WM_MENUSELECT загрузить из ресурсов ту строку, которая соответствует выделенному пункту меню. После этого загруженная строка выводится в строку состояния, тем самым, обеспечивая контекстную справку для выделенного пункта меню.

Для загрузки строки из ресурса таблицы строк предназначена функция API LoadString.

```
int LoadString(
HINSTANCE hInstance,
// дескриптор приложения, содержащего таблицу строк
UINT uID,
// идентификатор строки, которая должна быть загружена
LPTSTR lpBuffer,
// указатель на буфер, в который будет записана строка
int nBufferMax // размер буфера
);
```

Динамическое создание меню

Способ создания меню с помощью средств интегрированной среды разработки приложений не является единственным. Альтернативным подходом является программное (динамическое) создание меню.

Для создания и модификации меню используются следующие функции API.

Функция API CreateMenu предназначена для создания главного меню приложения: HMENU CreateMenu(VOID);

Меню, созданное этой функцией, изначально будет пустым, т.е. не будет содержать ни одного пункта. Заполнить меню пунктами можно с помощью функций API AppendMenu или InsertMenu.

Функция API CreatePopupMenu предназначена для создания всплывающего меню приложения: HMENU CreatePopupMenu(VOID);

Всплывающее меню, созданное этой функцией, также изначально будет пустым. Заполнить меню пунктами можно с помощью уже упомянутых функций API AppendMenu или InsertMenu.

Функция API AppendMenu применяется для добавления пунктов к концу меню.

```
BOOL AppendMenu(
HMENU hMenu,
// дескриптор меню, к которому добавляется новый пункт
UINT uFlags,
// внешний вид и правило поведения добавляемого пункта меню
UINT_PTR uIDNewItem,
// идентификатор для нового пункта меню или
// дескриптор выпадающего меню
LPCTSTR lpNewItem
// содержимое нового пункта меню – зависит от второго
// параметра
);
```

Второй параметр может иметь одно или несколько значений (флагов), приведенных ниже. В последнем случае флаги объединяются с помощью побитовой операции «ИЛИ».

– MF_POPUP – создает всплывающее меню. В этом случае третий параметр функции содержит дескриптор всплывающего меню.

– MF_CHECKED – помещает отметку рядом с пунктом меню.

– MF_DEFAULT – пункт меню установлен в качестве применяемого по умолчанию. Имя этого пункта выделяется жирным шрифтом. В этом случае при открытии подменю двойным щелчком мыши, Windows автоматически выполнит команду по умолчанию, закрыв при этом подменю.

– MF_ENABLED – делает пункт меню доступным для выбора.

– MF_DISABLED – делает пункт меню недоступным для выбора.

– MF_GRAYED – выделяет серым цветом пункт меню и запрещает его выбор.

– MF_UNCHECKED – пункт меню не имеет отметки.

– MF_SEPARATOR – указывает, что пункт меню является разделителем (сепаратором).

– MF_STRING – отображает пункт меню с использованием текстовой строки, которая задается в четвертом параметре.

Функция API InsertMenu позволяет вставить новый пункт в меню.

```
BOOL InsertMenu(
HMENU hMenu,
// дескриптор меню, которое модифицируется
UINT uPosition,
// идентификатор или позиция пункта меню, перед которым
```

```
// происходит вставка нового пункта
UINT uFlags,
// интерпретация второго параметра, а также внешний вид и
//правило поведения нового пункта меню
PTR uIDNewItem,
// идентификатор для нового пункта меню или
// дескриптор выпадающего меню
LPCTSTR lpNewItem
// содержимое нового пункта меню – зависит от третьего
// параметра
);
```

Второму параметру функции uPosition передается либо идентификатор пункта меню, либо позиция пункта меню. Выбор варианта интерпретации задается в третьем параметре:

– MF_BYCOMMAND – в этом случае второй параметр должен содержать идентификатор пункта меню, перед которым происходит вставка нового пункта;

– MF_BYPOSITION – в этом случае второй параметр должен содержать относительную позицию пункта меню с отсчетом от нуля.

Кроме того, в третьем параметре можно дополнительно указать один или несколько флагов, приведенных выше при рассмотрении функции AppendMenu.

Функция API ModifyMenu применяется для изменения существующего пункта меню.

```
BOOL ModifyMenu(
HMENU hMenu, // дескриптор меню, которое модифицируется
UINT uPosition,
// идентификатор или позиция пункта меню, который будет
// изменен
UINT uFlags,
// интерпретация второго параметра, а также внешний вид и
//правило поведения изменяемого пункта меню
PTR uIDNewItem,
// идентификатор для изменяемого пункта меню или
// дескриптор выпадающего меню
LPCTSTR lpNewItem
// новое содержимое изменяемого пункта меню – зависит
// от третьего параметра
```

);

Назначение параметров функции ModifyMenu аналогично функции InsertMenu.

Функция API DeleteMenu применяется для удаления отдельного пункта из указанного меню.

```
BOOL DeleteMenu(
HMENU hMenu, // дескриптор меню, которое модифицируется
UINT uPosition,
// идентификатор или позиция пункта меню, который будет
// удален
UINT uFlags // интерпретация второго параметра
);
```

Следует отметить, что если удаляемым пунктом является выпадающее меню, то оно уничтожается и высвобождается память.

После вызова функций AppendMenu, InsertMenu или DeleteMenu с целью модификации главного меню следует обязательно вызвать рассмотренную ранее функцию API DrawMenuBar для повторного отображения изменившейся полосы меню.

Функция API DestroyMenu предназначена для уничтожения меню и высвобождения памяти, занимаемой меню.

```
BOOL DestroyMenu(
HMENU hMenu // дескриптор уничтожаемого меню
);
```

Функция API GetMenuString получает строку текста указанного пункта меню.

```
int GetMenuString(
HMENU hMenu, // дескриптор меню
UINT uIDItem,
// идентификатор или позиция пункта меню, текст которого
// необходимо получить
LPTSTR lpString,
// указатель на строковый буфер, в который будет
// записана строка текста
int nMaxCount,
// максимальное число символов, которое должно быть
```

```
// записано в буфер
UINT uFlag // интерпретация второго параметра
);
```

Контекстное меню

Контекстное меню – это меню, которое появляется в любой части окна приложения при щелчке правой кнопкой мыши. При этом содержание контекстного меню изменяется в зависимости от «контекста» – места, где произведен щелчок правой кнопкой мыши. Обычно контекстное меню содержит пункты, которые дублируют команды основного меню, но сгруппированные иначе, чтобы максимально облегчить пользователю работу с приложением.

Создание контекстного меню выполняется аналогично созданию главного меню приложения. Для этого используется два подхода:

- определение шаблона меню в файле описания ресурсов;
- программное (динамическое) создание меню.

Создавая шаблон контекстного меню с помощью редактора меню, необходимо определить нулевой пункт меню нулевого уровня как подменю, имеющее какое-нибудь условное имя. Этот пункт нигде не будет отображаться. Он необходим только для получения дескриптора подменю с помощью функции API GetSubMenu.

После определения контекстного меню в файле описания ресурсов необходимо его загрузить с помощью функции API LoadMenu. Данная функция вернет дескриптор фиктивного меню нулевого уровня, которое не должно отображаться на экране.

Содержанием контекстного меню является нулевой пункт указанного меню, поэтому окончательное значение дескриптора контекстного меню определяется вызовом функции GetSubMenu. Полученный дескриптор контекстного меню затем передается функции TrackPopupMenu, которая выводит всплывающее контекстное меню на экран.

```
BOOL TrackPopupMenu(
HMENU hMenu, // дескриптор контекстного меню
UINT uFlags,
//флаги, определяющие позиционирование и другие опции меню
int x,
// горизонтальное расположение контекстного меню в экранных
// координатах
```

```
int y,
// вертикальное расположение контекстного меню в экранных
// координатах
int nReserved,
// зарезервированное значение; должно быть равно 0
HWND hWnd,
// дескриптор окна, которому принадлежит контекстное меню
HWND prcRect // параметр игнорируется
);
```

Описанные выше действия иллюстрируются следующим фрагментом кода:

```
// Загрузим меню из ресурсов приложения
HMENU hMenu = LoadMenu(GetModuleHandle(NULL), MAKEINTRESOURCE(IDR_MENU1));
// Получим дескриптор подменю
HMENU hSubMenu = GetSubMenu(hMenu, 0);
// Отобразим контекстное меню, левый верхний угол которого привязывается
// к точке текущего положения курсора мыши
TrackPopupMenu(hSubMenu, TPM_LEFTALIGN, xPos, yPos, 0, hDialog, NULL);
```

Как известно, при щелчке правой кнопкой мыши Windows отправляет оконной процедуре приложения сообщение WM_RBUTTONDOWN, содержащее в LPARAM клиентские координаты курсора мыши в момент щелчка. Помимо этого сообщения в оконную процедуру приложения приходит сообщение WM_CONTEXTMENU, содержащее в LPARAM экранные координаты курсора мыши (в младшей части LPARAM находится координата X положения курсора мыши, а в старшей части - координата Y), а в WPARAM дескриптор окна, в котором произведен щелчок.

Учитывая, что рассмотренная выше функция отображения контекстного меню TrackPopupMenu принимает экранные координаты позиции расположения меню, то было бы удобно предусмотреть в приложении обработчик сообщения WM_CONTEXTMENU вместо WM_RBUTTONDOWN для организации вызова контекстного меню. В этом случае не нужно выполнять преобразование клиентских координат в экранные координаты.

Функция API CheckMenuRadioItem отмечает выбранный пункт меню и снимает отметку со всех других опций меню в указанной группе.

```
BOOL CheckMenuRadioItem(
HMENU hmenu,
// дескриптор меню, которое содержит группу Radio-элементов
UINT idFirst,// первый пункт меню в группе
UINT idLast, // последний пункт меню в группе
UINT idCheck, // пункт меню, который должен быть отмечен
UINT uFlags // интерпретация второго, третьего и четввертого параметров
);
```

Последний параметр может принимать одно из следующих значений:

– MF_BYCOMMAND – в этом случае второй, третий и четвертый параметры должны содержать идентификатор пункта меню;

– MF_BYPOSITION – в этом случае второй, третий и четвертый параметры должны содержать относительную позицию пункта меню с отсчетом от нуля.

Акселераторы

Обычно для многих пунктов меню определены клавиатурные комбинации (акселераторы), предоставляющие альтернативный способ вызова команд меню.

Чтобы добавить в приложение обработку акселераторов, необходимо выполнить следующую последовательность действий:

– модифицировать определение ресурса меню, добавив к имени каждого дублируемого пункта информацию о быстрой клавише;

– определить таблицу акселераторов в файле описания ресурсов;

– обеспечить загрузку таблицы акселераторов в память приложения;

– модифицировать цикл обработки сообщений в функции WinMain.

Для того чтобы определить таблицу акселераторов в файле описания ресурсов необходимо воспользоваться редактором таблиц акселераторов. Для этого необходимо активизировать вкладку Resource View, в которой с помощью контекстного меню вызвать диалог добавления ресурса Add -> Resource…

В этом диалоговом окне необходимо выбрать из списка Accelerator и нажать кнопку New, в результате чего будет открыто окно редактора таблицы акселераторов.

По умолчанию редактор присваивает таблице акселераторов имя IDR_ACCELERATOR1. Впоследствии этот идентификатор можно изменить на другой идентификатор, отражающий семантику ресурса.

Поскольку акселераторы ставятся в соответствие командам меню, то они должны иметь те же идентификаторы, что и элементы меню, которым они соответствуют. Данное требование не является обязательным при реализации акселераторов, однако если оно не будет соблюдаться, то идея акселераторов как средства быстрого вызова команд меню потеряет смысл.

Во время работы приложения для загрузки таблицы акселераторов в память и получения ее дескриптора используется функция API LoadAccelerators.

```
HACCEL LoadAccelerators(
HINSTANCE hInstance, // дескриптор приложения
LPCTSTR lpTableName
// указатель на строку, содержащую имя таблицы
// акселераторов
);
```

Для обработки акселераторов приложение должно перехватывать сообщения клавиатуры, анализировать их коды и в случае совпадения с кодом, определенным в таблице акселераторов, направлять соответствующее сообщение в оконную процедуру главного окна. С этой целью используется функция API TranslateAccelerator.

```
int TranslateAccelerator (
HWND hWnd, // дескриптор окна-получателя сообщений
HACCEL hAccTable,// дескриптор таблицы акселераторов
LPMSG lpMsg // указатель на структуру MSG
);
```

Функция TranslateAccelerator преобразует сообщение WM_KEYDOWN или WM_SYSKEYDOWN в сообщение WM_COMMAND или WM_SYSCOMMAND, если таблица акселераторов содержит соответствующий код виртуальной клавиши (с учетом состояния клавиш <Ctrl>, <Alt> и <Shift>).

Сформированное сообщение, содержащее идентификатор акселератора в младшем слове параметра wParam, отправляется непосредственно в оконную процедуру, минуя очередь сообщений. Возврат из функции TranslateAccelerator происходит только после того, как оконная процедура обработает посланное сообщение.

Если функция TranslateAccelerator возвращает ненулевое значение, это значит, что преобразование комбинации клавиш и обработка отправленного сообщения завершились успешно. В этом случае приложение не должно повторно обрабатывать ту же самую комбинацию клавиш при помощи функций TranslateMessage и DispatchMessage.

Обработка пользовательского ввода в операционных системах. Организация вывода

В Win32 единицей работы компьютера является поток − ход выполнения программы в рамках процесса (в контексте процесса). Поток выполняет программный код, принадлежащий процессу. Процесс – это экземпляр выполняемой программы (но не ход ее выполнения). Он не является динамическим объектом и включает виртуальное адресное пространство, код и данные, файлы, синхронизирующие объекты, динамические библиотеки.

Каждое приложение создает, по меньшей мере, один первичный поток, но может создать и много потоков.

Любое приложение Windows представлено на экране дисплея как минимум одним окном с набором стандартных элементов управления.

Различают следующие типы окон:

– перекрывающие (overlapped window);

– всплывающие (pop–up window);

– дочерние (child window);

– слоистые (layered window) – особые окна, которые позволяют

– улучшить визуальный эффект, включая прозрачность.

Перекрывающие окна создаются функцией CreateWindowEx( ) со стилем WS_OVERLAPPEDWINDOW. Этот стиль определяет наличие заголовка, системного меню, кнопок минимизации и максимизации, кнопки закрытия окна и «толстой» рамки, позволяющей изменять размеры окна.

Перекрывающие окна предназначены для главных окон приложений и могут иметь меню.

Всплывающие окна создаются функцией CreateWindowEx( ) со стилем WS_POPUP и предназначены для окон диалогов, окон сообщений

и других окон временного использования, которые могут находиться вне главного окна приложения. Для того чтобы временное окно имело заголовок, рамку и системное меню, необходимо при его создании использовать комбинацию стилей WS_POPUPWINDOW | WS_CAPTION.

Дочерние окна создаются функцией CreateWindowEx( ) со стилем WS_CHILD и обычно используются для разделения клиентской области родительского окна на отдельные функциональные области. Дочерние окна могут иметь заголовок, системное меню, кнопки минимизации и максимизации, рамку и полосы прокрутки, но не могут иметь меню. Дочерние окна всегда находятся в пределах клиентской области родительского окна, т.е. их координаты всегда отсчитываются от левого верхнего угла родительского окна. Родительское окно может быть перекрывающим, всплывающим или даже другим дочерним окном.

Виды приложений:

– SDI (Single Document Interface) – приложение с одно–документным интерфейсом;

– MDI (Multiple Document Interface) – приложение с многодокументным интерфейсом;

– диалоговое приложение (Based Dialog) − содержит только диалоговое окно с элементами управления, не имеет главного окна, а значит, не имеет меню.

Элементы управления

1 Элемент управления «кнопка»

1.1 Обычная кнопка (Button)

Создать кнопку на форме диалога можно двумя способами:

– с помощью средств интегрированной среды разработки Microsoft Visual Studio;

– посредством вызова функции CreateWindowEx.

При первом способе необходимо определить кнопку в шаблоне диалогового окна на языке описания шаблона диалога. Это произойдёт автоматически, если активизировать окно Toolbox (<Ctrl><Alt><X>) и «перетащить» кнопку на форму диалога.

После размещения кнопки на форме диалога ей назначается идентификатор (например, IDC_BUTTON1), который впоследствии можно изменить на идентификатор, отражающий семантику ресурса.

Свойства элемента управления Button:

– свойство Caption содержит текстовую строку, которая будет отображаться внутри ограничивающего прямоугольника элемента Button. При этом в строке могут использоваться управляющие символы \t (табуляция) и \n (перевод строки).

– свойство Multiline позволяет располагать текст в несколько строк;

– свойства Horizontal Alignment и Vertical Alignment позволяют выбрать вариант выравнивания текста внутри ограничивающего прямоугольника;

– свойство Flat позволяет создать плоскую кнопку;

– свойство Default Button назначает кнопке атрибут «применяемая по умолчанию»;

– свойства Icon или Bitmap позволяют указать, что вместо текста на кнопке будет отображаться пиктограмма или растровый образ;

– при истинном значении свойства Notify кнопка будет отправлять уведомление BN_CLICKED родительскому окну (диалогу).

При воздействии на элемент управления диалога в диалоговую процедуру DlgProc поступает сообщение WM_COMMAND, в котором LOWORD(wParam) содержит идентификатор элемента управления, HIWORD(wParam) содержит код уведомления, а lParam – дескриптор элемента управления.

Если кнопка имеет фокус ввода, то текст на кнопке обводится штриховой линией, а нажатие и отпускание клавиши пробела имеет тот же эффект, что и щелчок мышью по кнопке. Существует программный способ перевода фокуса ввода на элемент управления. Для этой цели служит функция API SetFocus.

```
HWND SetFocus(
HWND hWnd
// дескриптор окна, приобретающего клавиатурный ввод
);
```

Для получения дескриптора окна (элемента управления), обладающего фокусом ввода используется функция API GetFocus: HWND GetFocus(VOID).

Альтернативный способ создания кнопки – использование функции CreateWindowEx. В этом случае во втором параметре функции передается имя предопределенного оконного класса – BUTTON.

При создании кнопок используются следующие стили:

– стиль WS_CHILD позволяет создать кнопку как дочернее окно диалога;

– стиль WS_VISIBLE управляет видимостью кнопки;

– стиль BS_BITMAP указывает на то, что на кнопке должен быть рисунок (растровый битовый образ) вместо текста;

– стиль WS_DISABLED указывает на то, что кнопка будет запрещённой. Напомнить слушателям, что запрещённые элементы выводятся на экран серым цветом и не воспринимают пользовательский ввод с клавиатуры или от мыши.

1.2 Флажок (Check Box)

Данный элемент управления обычно применяется для установки или сброса определённых опций, независимых друг от друга. Флажок действует как двухпозиционный переключатель. Один щелчок вызывает появление контрольной отметки (галочки), а другой щелчок приводит к ее исчезновению.

Создать Check Box на форме диалога можно двумя способами:

– с помощью средств интегрированной среды разработки Microsoft Visual Studio;

– посредством вызова функции CreateWindowEx.

При первом способе необходимо определить Check Box в шаблоне диалогового окна на языке описания шаблона диалога. Это произойдёт автоматически, если активизировать окно Toolbox (<Ctrl><Alt><X>) и «перетащить» Check Box на форму диалога.

После размещения флажка на форме диалога ему назначается идентификатор (например, IDC_CHECK1), который впоследствии можно изменить на идентификатор, отражающий семантику ресурса.

Следует отметить, что элемент управления Check Box обладает тем же набором свойств, что и Button, а также располагает дополнительными свойствами, характерными только для него:

– свойство Auto позволяет элементу управления отслеживать все щелчки мышью, и при этом элемент управления сам включает или выключает контрольную отметку. Если же отключить свойство Auto, то управление флажком полностью возлагается на приложение;

– свойство Tri-state используется для создания флажка, имеющего три состояния. Кроме состояний «установлен» и «сброшен» добавляется «неопределенное состояние», в котором флажок отображен в серой гамме. Серый цвет показывает пользователю, что выбор флажка не определен или не имеет отношения к текущей операции;

– свойство Push-like изменяет внешний вид флажка так, что он выглядит как нажимаемая кнопка. Вместо установки галочки эта кнопка

переходит в нажатое состояние и остается в нем до следующего щелчка мышью.

Для того, чтобы перевести Check Box в некоторое состояние, ему необходимо отправить сообщение BM_SETCHECK, передав в WPARAM одно из следующих значений:

– BST_CHECKED – установить отметку;

– BST_UNCHECKED – снять отметку;

– BST_INDETERMINATE – установить неопределенное состояние.

Существует альтернативный способ программной инициализации состояния элемента управления Check Box. Для этого используется функция API CheckDlgButton.

```
BOOL CheckDlgButton(
HWND hDlg,
// дескриптор диалога, содержащего кнопку (флажок)
int nIDButton,
// идентификатор элемента управления (флажка)
UINT uCheck
// состояние флажка – одно из вышеперечисленных значений
);
```

Для получения состояния флажка следует ему послать сообщение BM_GETCHECK. В этом случае SendMessage вернёт одно из вышеперечисленных значений.

Альтернативным способом получения состояния флажка является вызов функции API IsDlgButtonChecked.

```
UINT IsDlgButtonChecked(
HWND hDlg, // дескриптор диалога, содержащего кнопку (флажок)
int nIDButton // идентификатор элемента управления (флажка)
).
```

1.3 Переключатель (Radio Button)

Данный элемент управления обычно применяется для представления в окне множества взаимоисключающих опций, из которых можно выбрать только одну. В отличие от флажков, повторный щелчок на переключателе не меняет его состояние.

Создать Radio Button на форме диалога можно двумя способами:

– с помощью средств интегрированной среды разработки Microsoft Visual Studio;

– посредством вызова функции CreateWindowEx.

При первом способе необходимо определить Radio Button в шаблоне диалогового окна на языке описания шаблона диалога. Это произойдёт автоматически, если активизировать окно Toolbox (<Ctrl><Alt><X>) и «перетащить» Radio Button на форму диалога.

После размещения переключателя на форме диалога ему назначается идентификатор (например, IDC_RADIO1), который впоследствии можно изменить на идентификатор, отражающий семантику ресурса.

Следует отметить, что элемент управления Radio Button обладает тем же набором свойств, что и Check Box, но, кроме того, имеет важное свойство Group. Для первого переключателя в группе связанных взаимоисключающих переключателей нужно обязательно установить значение свойства Group, равным True. Все последующие переключатели (в файле описания ресурсов) со значением свойства Group, равным False, считаются принадлежащими к этой группе. Если в последовательности описаний элементов управления встречается переключатель со значением свойства Group, равным True, считается, что он начинает новую группу элементов Radio Button.

Для того, чтобы перевести Radio Button в некоторое состояние, ему необходимо отправить сообщение BM_SETCHECK, передав в WPARAM одно из следующих значений:

– BST_CHECKED – установить отметку;
– BST_UNCHECKED – снять отметку;
– BST_INDETERMINATE – установить неопределенное состояние.

Существует альтернативный способ выбора переключателя. Для этого используется функция API CheckRadioButton.

```
BOOL CheckRadioButton(
HWND hDlg,
// дескриптор диалога, содержащего кнопку (переключатель)
int nIDFirstButton,
// идентификатор первого переключателя в группе
int nIDLastButton,
// идентификатор последнего переключателя в группе
int nIDCheckButton
// идентификатор выбираемого переключателя
);
```

Данная функция помечает указанный переключатель в группе, удаляя отметку со всех других переключателей этой же группы. Другими

словами, функция CheckRadioButton посылает сообщение BM_SETCHECK каждому переключателю указанной группы. При этом выбираемому переключателю в WPARAM передается BST_CHECKED, а остальным – BST_UNCHECKED.

Для получения состояния переключателя следует ему послать сообщение BM_GETCHECK. В этом случае SendMessage (либо SendDlgItemMessage) вернёт одно из вышеперечисленных значений.

Альтернативным способом получения состояния переключателя является вызов функции API IsDlgButtonChecked, рассмотренной ранее.

Для посылки сообщений элементам управления используется функция API SendDlgItemMessage.

```
LRESULT SendDlgItemMessage(
HWND hDlg,
// дескриптор диалога, содержащего элемент управления
int nIDDlgItem,
// идентификатор дочернего окна (элемента управления),
// которому отправляется сообщение
UINT Msg, // идентификатор сообщения
WPARAM wParam, // дополнительная информация о сообщении
LPARAM lParam  // дополнительная информация о сообщении
);
```

2 Элемент управления «текстовое поле ввода» (Edit Control)

На практике применяются различные текстовые поля ввода в самом широком спектре: от небольшого однострочного поля ввода до многострочного элемента управления с автоматическим переносом строк, как в программе Microsoft Notepad. Создать текстовое поле ввода на форме диалога можно двумя способами:

– с помощью средств интегрированной среды разработки Microsoft Visual Studio;

– посредством вызова функции CreateWindowEx.

При первом способе необходимо определить Edit Control в шаблоне диалогового окна на языке описания шаблона диалога. Это произойдёт автоматически, если активизировать окно Toolbox (<Ctrl><Alt><X>) и «перетащить» текстовое поле ввода на форму диалога.

После размещения текстового поля ввода на форме диалога ему назначается идентификатор (например, IDC_EDIT1), который

впоследствии можно изменить на идентификатор, отражающий семантику ресурса.

По умолчанию Edit Control является однострочным, с автоматической горизонтальной прокруткой (свойство Auto HScroll), с объемной рамкой (свойство Border) и выравниванием текста по левой границе окна (свойство Align Text со значением Left).

Если установить истинным значение свойства Multiline, то текстовое поле ввода будет работать в многострочном режиме. При этом станут доступными для использования свойства Horizontal Scroll, Vertical Scroll, Auto VScroll.

Значение True свойства Number разрешает вводить в Edit Control только цифры. Если же окно редактирования предназначено для ввода пароля, то следует установить истинным значение свойства Password, и тогда вместо вводимых символов в поле ввода будут отображаться символы звездочки.

Свойство No Hide Selection позволяет элементу управления всегда показывать выделенную подстроку, даже если Edit Control теряет фокус.

Свойство Want Return используется для многострочного окна редактирования. Если для этого свойства установлено значение True, то нажатие клавиши <Enter> приводит к вводу символа возврата каретки.

Взаимоисключающие свойства Uppercase и Lowercase преобразуют вводимые символы. В первом случае происходит конвертирование в символы верхнего регистра, а во втором – в символы нижнего регистра.

Истинное значение свойства Read Only устанавливает для Edit Control режим «только для чтения», не позволяющий пользователю вводить или редактировать текст.

Альтернативный способ создания текстового поля ввода - использование функции CreateWindowEx. В этом случае во втором параметре функции передается имя предопределенного оконного класса – EDIT.

Сообщение WM_GETTEXTLENGTH, которое отправляется текстовому полю ввода с целью определения длины введённого текста.

Для изменения стилей элементов управления используется функция API SetWindowLong.

```
LONG SetWindowLong(
HWND hWnd, // дескриптор окна
int nIndex, // индекс значения, которое нужно изменить
LONG dwNewLong // новое значение
);
```

Функция API GetWindowLong позволяет получить текущие стили окна (элемента управления).

```
LONG GetWindowLong(
HWND hWnd, // дескриптор окна
int nIndex // индекс значения, которое нужно выбрать
);
```

3 Элемент управления «список» (List Box)

Элемент управления List Box представляет собой список элементов, из которых пользователь может выбрать один или более. Элементы списка могут быть представлены строками, растровыми образами или комбинацией текста и изображения. Если размеры окна не позволяют показать все элементы списка, то List Box создает полосу прокрутки. Типичный пример использования списка – список файлов и папок в диалоговом окне «Открыть» приложения «Блокнот».

Создать список на форме диалога можно двумя способами:

– с помощью средств интегрированной среды разработки Microsoft Visual Studio;

– посредством вызова функции CreateWindowEx.

При первом способе необходимо определить List Box в шаблоне диалогового окна на языке описания шаблона диалога. Это произойдёт автоматически, если активизировать окно Toolbox (<Ctrl><Alt><X>) и «перетащить» список на форму диалога.

После размещения списка на форме диалога ему назначается идентификатор (например, IDC_LIST1), который впоследствии можно изменить на идентификатор, отражающий семантику ресурса.

Следует отметить, что различают списки с единичным выбором, в которых пользователь может выбрать только один пункт списка, и списки с множественным выбором, допускающие выбор более одного пункта списка. Система обычно отображает выбранный элемент в списке, инвертируя цвет текста и цвет фона для символов.

Свойства элемента управления List Box.

По умолчанию создается список с единичным выбором (свойство Selection со значением Single), имеющий рамку (свойство Border со значением True), выполняющий автоматическую сортировку элементов списка (истинное значение свойства Sort) и обеспечивающий появление

вертикальной полосы прокрутки в случае необходимости (свойство Vertical Scroll со значением True).

Свойство Multicolumn позволяет располагать список в несколько столбцов. В этом случае List Box может прокручиваться по горизонтали. Но для этого свойство Horizontal Scroll должно иметь значение True.

4 Элемент управления «комбинированный список» (Combo Box)

Комбинированный список (Combo Box) является комбинацией списка и текстового поля ввода. Типичный пример — окно «Configuration Manager» интегрированной среды разработки приложений Microsoft Visual Studio, обладающее комбинированным списком для выбора конфигурации проекта (Debug или Release).

Создать Combo Box на форме диалога можно двумя способами:

– с помощью средств интегрированной среды разработки Microsoft Visual Studio;

– посредством вызова функции CreateWindowEx.

При первом способе необходимо определить Combo Box в шаблоне диалогового окна на языке описания шаблона диалога. Это произойдёт автоматически, если активизировать окно Toolbox (<Ctrl><Alt><X>) и «перетащить» комбинированный список на форму диалога.

После размещения списка на форме диалога ему назначается идентификатор (например, IDC_COMBO1), который впоследствии можно изменить на идентификатор, отражающий семантику ресурса.

Следует отметить, что многие свойства Combo Box аналогичны свойствам элемента управления List Box. Но есть и некоторые различия. В частности, стиль комбинированного списка задаётся свойством Type.

5 Общие элементы управления

Помимо базовых элементов управления (Button, Edit Control, Static и т.д.), которые поддерживались самыми ранними версиями Windows, в системе используется библиотека элементов управления общего пользования (common control library). Общие элементы управления, включенные в эту библиотеку, дополняют базовые элементы управления и позволяют придать приложениям более совершенный вид. К общим элементам управления относятся панель инструментов (Toolbar), окно подсказки (Tooltip), индикатор (Progress Bar), счётчик (Spin Control), строка состояния (Status Bar) и другие. Библиотека элементов управления

общего пользования реализована в виде динамически загружаемой библиотеки comctl32.dll.

Большинство элементов управления общего пользования реализовано в виде окна соответствующего предопределенного класса, и, следовательно, элементы управления могут быть созданы вызовом функции CreateWindowEx.

Разница между базовыми элементами управления и общими элементами управления состоит в типе посылаемых уведомительных сообщений. Базовые элементы управления посылают сообщения WM_COMMAND, а общие элементы управления почти всегда посылают сообщения WM_NOTIFY.

Чтобы использовать в приложении какой-либо элемент управления общего пользования, сначала нужно вызвать функцию API InitCommonControlsEx, которая регистрирует оконные классы элементов управления.

```
BOOL InitCommonControlsEx(
LPINITCOMMONCONTROLSEX lpInitCtrls
);
```

В единственном параметре lpInitCtrls передается адрес структурной переменной типа INITCOMMONCONTROLSEX, содержащей информацию о том, какие классы элементов управления должны быть зарегистрированы.

```
typedef struct tagINITCOMMONCONTROLSEX {
DWORD dwSize; // размер структуры в байтах
DWORD dwICC; // флаги загрузки классов из DLL
} INITCOMMONCONTROLSEX, *LPINITCOMMONCONTROLSEX;
```

Необходимо отметить, что библиотека элементов управления общего пользования реализована в виде динамически загружаемой библиотеки comctl32.dll. Для того чтобы использовать в приложении функцию InitCommonControlsEx необходимо подключить заголовочный файл commctrl.h, в котором она описана. Помимо этого следует указать компоновщику расположение библиотечного файла comctl32.lib, содержащего ссылку на DLL и перечень находящихся в ней функций.

5.1 Элемент управления «индикатор процесса» (Progress Control)

Элемент управления Progress Control обычно используется в приложениях для отображения процесса выполнения некоторой длительной операции.

Создать индикатор процесса на форме диалога можно двумя способами:

– с помощью средств интегрированной среды разработки Microsoft Visual Studio;

– посредством вызова функции CreateWindowEx.

При первом способе необходимо определить Progress Control в шаблоне диалогового окна на языке описания шаблона диалога. Для этого следует активизировать окно Toolbox и «перетащить» индикатор на форму диалога.

После размещения индикатора на форме диалога ему назначается идентификатор (например, IDC_PROGRESS1), который впоследствии можно изменить на идентификатор, отражающий семантику ресурса.

По умолчанию Progress Control заполняется по горизонтали, т.е. слева направо. Для вертикального заполнения индикатора (снизу вверх) необходимо свойству Vertical указать значение True.

При ложном значении свойства Smooth (по умолчанию) индикатор процесса заполняется отдельными маленькими прямоугольниками.

Если же необходимо использовать сплошное заполнение окна индикатора процесса, то свойству Smooth нужно указать значение True.

5.2 Общий элемент управления «регулятор» (Slider Control)

Элемент управления Slider Control (ползунок или регулятор), который ранее назывался Trackbar, представляет собой окно с линейкой и перемещаемым по ней ползунком. Примером этого элемента управления является «Регулятор громкости» операционной системы Windows. Подобный регулятор дает возможность пользователю выбирать дискретные значения в заданном диапазоне.

Создать Slider Control на форме диалога можно двумя способами:

– с помощью средств интегрированной среды разработки Microsoft Visual Studio;

– посредством вызова функции CreateWindowEx.

При первом способе необходимо определить Slider Control в шаблоне диалогового окна на языке описания шаблона диалога. Для

этого следует активизировать окно Toolbox и «перетащить» регулятор на форму диалога.

После размещения регулятора на форме диалога ему назначается идентификатор (например, IDC_SLIDER1), который впоследствии можно изменить на идентификатор, отражающий семантику ресурса.

Некоторые свойства элемента управления Slider Control:

– свойство Orientation позволяет выбрать ориентацию элемента управления: Horizontal (по умолчанию) или Vertical;

– свойство Point предназначено для выбора стиля размещения меток: Both (по умолчанию метки размещаются с обеих сторон), Top/Left или Bottom/Right;

– свойство Tick Marks задаёт наличие или отсутствие меток;

– свойство Auto Ticks определяет, будут ли метки для всех значений в заданном диапазоне значений или только для начального и конечного положений ползунка;

– свойство Border определяет наличие рамки у элемента управления;

– свойство Tooltips включает поддержку всплывающих подсказок, отображающих текущую позицию ползунка.

5.3 Общий элемент управления «счётчик» (Spin Control)

Элемент управления Spin Control (счётчик) реализован как две кнопки со стрелками, с помощью которых можно увеличивать, или уменьшать некоторое числовое значение. Значение, связанное со счётчиком, называется его текущей позицией.

Кроме того, счётчик можно ассоциировать с другим элементом управления, называемым приятельским окном (buddy window). Чаще всего таким окном является текстовое поле. Комбинацию счётчика с текстовым полем называют также полем с прокруткой. Поле с прокруткой воспринимается пользователем как единый элемент управления. Содержимое текстового поля в таком элементе отображает текущую позицию счётчика.

Счётчик можно создать несколькими способами:

– при помощи редактора диалоговых окон на этапе визуального проектирования формы диалога;

– посредством вызова функции CreateWindowEx;

– посредством вызова функции CreateUpDownControl.

При первом способе необходимо определить Spin Control в шаблоне диалогового окна на языке описания шаблона диалога. Для этого

следует активизировать окно Toolbox и «перетащить» счётчик на форму диалога.

После размещения спина на форме диалога ему назначается идентификатор (например, IDC_SPIN1), который впоследствии можно изменить на идентификатор, отражающий семантику ресурса.

Некоторые свойства элемента управления Spin Control:

– свойство Orientation предназначено для выбора ориентации элемента управления: Vertical (по умолчанию) или Horizontal;

– выпадающий список Alignment позволяет выбрать один из трех стилей размещения счётчика:

а) Unattached – счётчик располагается рядом с «приятелем»;

б) Left - счётчик располагается в левой части «приятеля», уменьшая тем самым его клиентскую область;

в) Right - счётчик располагается в правой части «приятеля», уменьшая тем самым его клиентскую область.

– свойство Auto Buddy обеспечивает автоматический выбор в качестве «приятельского окна» ближайшего предыдущего элемента управления в файле описания ресурсов;

– свойство Set Buddy Integer вместе с предыдущим свойством определяет синхронную работу счётчика и «приятеля»: любое изменение позиции счётчика сразу отображается в ассоциированном окне. Аналогично при вводе в «приятельское окно» допустимого целого числа устанавливается новая позиция счётчика;

– свойство No Thousands позволяет вставлять пробел после каждых трех цифр в изображении десятичного числа;

– свойство Wrap определяет работу счётчика – при истинном значении этого свойства счётчик работает как циклический, т.е. после максимального значения текущим становится минимальное, и наоборот;

– свойство Arrow Keys поддерживает управление счётчиком с помощью клавиш управления курсором.

Таким образом, при создании поля с прокруткой необходимо поместить счётчик на форму диалога сразу вслед за размещением «приятеля» (текстового поля). Кроме того, свойствам Auto Buddy и Set Buddy Integer следует установить значение True.

Существует ещё один программный способ создания счётчика – вызов функции API CreateUpDownControl, которая создаёт счётчик, определяет его минимальную, максимальную и текущую позиции, а также приятельское окно.

HWND CreateUpDownControl(

DWORD dwStyle, // стили элемента управления
int x, // клиентская координата X левого верхнего угла
int y, // клиентская координата Y левого верхнего угла
int cx, // ширина элемента управления
int cy, // высота элемента управления
HWND hParent, // дескриптор родительского окна
int nID, // идентификатор элемента управления
HINSTANCE hInst, // дескриптор приложения
HWND hBuddy, // дескриптор «приятеля»
int nUpper, // верхняя граница
int nLower, // нижняя граница
int nPos // текущая позиция
).

Графические подсистемы

Графика в 32-битной Windows реализуется, в основном, функциями, экспортируемыми из GDI32.DLL, динамически подключаемой библиотеки. Эти модули обращаются к различным функциям драйверов отображения – .DRV файлу для видеомониторов и, возможно, к одному или нескольким .DRV файлам драйверов принтеров или плоттеров. Видеодрайвер осуществляет доступ к аппаратуре видеомонитора или преобразует команды GDI в коды или команды, воспринимаемые различными принтерами. Разные видеоадаптеры и принтеры требуют различных файлов драйверов.

Поскольку к IBM PC совместимым компьютерам может быть подключено множество различных устройств отображения, одной из основных задач GDI является поддержка аппаратно-независимой графики. Программы для Windows должны нормально работать на любых графических устройствах отображения, которые поддерживаются Windows. GDI выполняет эту задачу, предоставляя возможность отделить ваши программы от конкретных характеристик различных устройств отображения.

Все графические устройства отображения делятся на две больших группы: растровые устройства и векторные устройства. Большинство устройств, подключаемых к PC – растровые устройства, т. к. они представляют графические образы как шаблон точек. Эта группа включает видеоадаптеры, матричные принтеры и лазерные принтеры. Группа векторных устройств, отображающих графические образы с использованием линий, в основном, состоит из плоттеров.

Большинство традиционных графических программ работает исключительно с векторами. Это значит, что программа, использующая один из таких графических языков, абстрагируется от особенностей оборудования.

Устройство отображения оперирует пикселями для создания графических образов, тогда как программа при связи с интерфейсом не использует понятие пикселя. Несмотря на то, что Windows GDI – это высокоуровневая векторная система рисования, она также может применяться и для относительно низкоуровневых манипуляций с пикселями.

С этой точки зрения, Windows GDI это такой же традиционный графический язык, как С – язык программирования. Язык С – хорошо известен своей высокой степенью переносимости относительно разных операционных систем и сред. С также хорошо известен тем, что он дает программисту возможность выполнять низкоуровневые системные функции, что часто недоступно в других языках программирования высокого уровня.

Также как С иногда называют "ассемблером высокого уровня", так можно считать, что GDI – это высокоуровневый интерфейс для аппаратных средств графики.

Как уже было сказано выше, по умолчанию Windows использует систему координат, основанную на пикселях.

Большинство традиционных графических языков используют "виртуальные" системы координат, с границами 0 и 32767 для горизонтальной и вертикальной осей. Хотя некоторые графические языки не дают вам возможности использовать пиксельную систему координат, Windows GDI позволяет применять обе системы (также как дополнительные координатные системы на базе физических единиц измерения).

Большинство программ для Windows вызывают функцию UpdateWindow при инициализации в WinMain, сразу перед входом в цикл обработки сообщений. Windows использует эту возможность для асинхронной отправки в оконную процедуру первого сообщения WM_PAINT. Это сообщение информирует оконную процедуру о том, что рабочая область готова к рисованию. После этого оконная процедура должна быть готова в любое время обработать дополнительные сообщения WM_PAINT и даже перерисовать, при необходимости, всю рабочую область окна. Оконная процедура получает сообщение WM_PAINT при возникновении одной из следующих ситуаций:

– предварительно скрытая область окна открылась, когда пользователь передвинул окно или выполнил какие-то действия, в результате которых окно вновь стало видимым;

– пользователь изменил размера окна (если в стиле класса окна установлены биты CS_HREDRAW и CS_HVREDRAW);

– в программе для прокрутки части рабочей области используются функции ScrollWindow или ScrollDC;

– для генерации сообщения WM_PAINT в программе используются функции InvalidateRect или InvalidateRgn.

В некоторых случаях, когда часть рабочей области временно закрывается, Windows пытается сначала ее сохранить, а затем восстановить. Это не всегда возможно. В некоторых случаях Windows может послать синхронное сообщение WM_PAINT. Обычно это происходит, когда:

– Windows удаляет диалоговое окно или окно сообщения, которое перекрывало часть окна программы;

– раскрывается пункт горизонтального меню и затем удаляется с экрана.

В некоторых случаях Windows всегда сохраняет перекрываемую область, а потом восстанавливает ее. Происходит это всегда, когда:

– курсор мыши перемещается по рабочей области;

– иконку перемещают по рабочей области.

Если программе необходимо перерисовать свою рабочую область, она может заставить Windows сгенерировать сообщение WM_PAINT. Это может показаться не самым простым способом вывода информации на экран, но структура вашей программы от этого только улучшится.

В Windows имеется несколько функций GDI для вывода строк текста в рабочей области окна. Функция TextOut выводит на экран строку символов TextOut(hdc, x, y, psString, iLength);

Параметр psString — это указатель на строку символов, а iLength – длина строки символов. Параметры x и y определяют начальную позицию строки символов в рамках рабочей области. Параметр hdc – это "описатель контекста устройства" (handle to a device context), являющийся важной частью GDI. Практически каждой функции GDI в качестве первого параметра необходим этот описатель.

Обработка сообщения WM_PAINT почти всегда начинается с вызова функции BeginPaint.

```
hdc = BeginPaint(hwnd, &ps);
```

и заканчивается вызовом функции EndPaint.

EndPaint(hwnd, &ps);

В обеих функциях первый параметр – это описатель окна программы, а второй – это указатель на структуру типа PAINTSTRUCT. В структуре PAINTSTRUCT содержится некоторая информация, которую оконная процедура может использовать для рисования в рабочей области.

При обработке вызова BeginPaint, Windows обновляет фон рабочей области, если он еще не обновлен. Обновление фона осуществляется с помощью кисти, заданной в поле hbrBackground структуры WNDCLASSEX, которая использовалась при регистрации класса окна. В случае нашей программы HELLOWIN подготовлена белая кисть и это означает, что Windows обновит фон окна, закрасив его белым цветом. Вызов BeginPaint делает всю рабочую область действительной (не требующей перерисовки) и возвращает описатель контекста устройства. Контекст устройства описывает физическое устройство вывода информации (например, дисплей) и его драйвер. Описатель контекста устройства необходим вам для вывода в рабочую область окна текста и графики. Используя описатель контекста устройства, возвращаемого функцией BeginPaint, вы не сможете рисовать вне рабочей области, даже не пытайтесь. Функция EndPaint освобождает описатель контекста устройства, после чего его значение нельзя использовать.

Если оконная процедура не обрабатывает сообщения WM_PAINT (что бывает крайне редко), они должны передаваться в DefWindowProc. Функция DefWindowProc просто по очереди вызывает BeginPaint и EndPaint и, таким образом, рабочая область устанавливается в действительное состояние, т. е. состояние, не требующее перерисовки.

После того, как WndProc вызвала BeginPaint, она вызывает GetClientRect: GetClientRect(hwnd, &rect);

Первый параметр – это описатель окна программы. Второй параметр – это указатель на переменную rect, для которой в WndProc задан тип RECT.

RECT – это структура "прямоугольник" (rectangle), определенная в заголовочных файлах Windows. Она имеет четыре поля типа LONG, имена полей: left, top, right и bottom. GetClientRect помещает в эти четыре поля размер рабочей области окна. Поля left и top всегда устанавливаются в 0. В полях right и bottom устанавливается ширина и высота рабочей области в пикселях.

WndProc никак не использует структуру RECT, за исключением передачи указателя на нее в качестве четвертого параметра функции DrawText: DrawText(hdc, "Hello, Windows!", -1, &rect, DT_SINGLELINE | DT_CENTER | DT_VCENTER);

DrawText рисует текст. Поскольку эта функция что-то рисует, то первый параметр – это описатель контекста устройства, возвращенный функцией BeginPaint. Вторым параметром является рисуемый текст, а третий параметр установлен в – 1, чтобы показать, что строка текста заканчивается нулевым символом.

Последний параметр – это набор флагов, значения которых задано в заголовочных файлах Windows. Флаги показывают, что текст следует выводить в одну строку, по центру относительно горизонтали и вертикали и внутри прямоугольной области, размер которой задан четвертым параметром. Вызов этой функции приводит, таким образом, к появлению строки "Hello, Windows!" в центре рабочей области.

Когда рабочая область становится недействительной (как это происходит при изменении размеров окна), WndProc получает новое сообщение WM_PAINT. Новый размер окна WndProc получает, вызвав функцию GetClientRect, и снова рисует текст в центре окна.

В основном, функции GDI могут быть разбиты на несколько крупных групп. Это группы не имеют четких границ и частично перекрываются.

Функции, которые получают (или создают) и освобождают (или уничтожают) контекст устройства.

Функции GetDC и ReleaseDC позволяют сделать это внутри обработчиков сообщений, отличных от WM_PAINT. Функции BeginPaint и EndPaint (хотя технически — это часть подсистемы USER в Windows) используются в теле обработчика сообщения WM_PAINT для рисования.

Функции, которые получают информацию о контексте устройства. Например, функция GetTextMetrics, используемая для получения информации о размерах выбранного в контексте устройства шрифта.

Функции рисования. Существуют различные функции GDI позволяющие рисовать линии, залитые области, растровые образы.

Функции, которые устанавливают и получают атрибуты контекста устройства. Атрибут контекста устройства определяет различные особенности работы функции рисования. Например, вы используете функцию SetTextColor для задания цвета любого текста, выводимого с использованием функции TextOut.

Функции, которые работают с объектами GDI. Например, функции CreatePen, CreatePenIndirect, ExtCreatePen. Эти функции возвращают описатель логического карандаша.

Типы графических объектов, выводимых на экран или принтер, которые могут быть разделены на несколько категорий, часто называют "примитивами". Это:

– прямые (отрезки) и кривые. Прямые – основа любой векторной графической системы. GDI поддерживает прямые линии, прямоугольники, эллипсы (включая окружности), дуги, являющиеся частью кривой эллипса, и сплайны Безье. Все они будут рассмотрены в этой главе. Более сложные кривые могут быть изображены как ломаные линии, которые состоят из очень коротких прямых, определяющих кривые. Линии рисуются с использованием карандаша, выбранного в контексте устройства;

– закрашенные области. Если набор прямых и кривых линий ограничивает со всех сторон некоторую область, то она может быть закрашена с использованием объекта GDI "кисть", выбранного в контексте устройства. Эта кисть может быть сплошной, штриховой (состоящей из горизонтальных, вертикальных или диагональных штрихов) или шаблонной, заполняющей область горизонтально и вертикально;

– битовые шаблоны (растровые шаблоны, растровые образы). Битовые шаблоны — это двумерный массив битов, соответствующий пикселям устройства отображения. Это базовый инструмент в растровой графике. Битовые образы используются, в основном, для отображения сложных (часто из реального мира) изображений на дисплее или принтере. Битовые образы также используются для отображения маленьких картинок, таких как значки, курсоры мыши, кнопки панели инструментов программ, которые нужно быстро нарисовать. GDI поддерживает два типа битовых образов или шаблонов — старые, но все еще используемые, аппаратно-зависимые, являющиеся объектами GDI, и новые (начиная с версии Windows 3.0) аппаратно-независимые (Device Independent Bitmap, DIB), которые могут быть сохранены в файлах на диске;

– текст. Текст отличается от других математических объектов компьютерной графики. Типов текста бесконечно много. Это известно из многолетней истории типографского дела, которое многие считают искусством. Поэтому поддержка текста часто наиболее сложная часть в системах компьютерной графики, и, вместе с тем, наиболее важная. Структуры данных, используемые для описания объектов GDI — шрифтов, а также для получения информации о них — самые большие

среди других структур данных в Windows. Начиная с версии Windows 3.1, GDI поддерживает шрифты TrueType, основанные на закрашенных контурах, которыми могут манипулировать другие функции GDI. Windows из соображений совместимости и экономии памяти по-прежнему поддерживает старые шрифты, основанные на битовых массивах (такие как системный шрифт по умолчанию).

Другие аспекты GDI не так легко классифицируются. Это:

– режимы масштабирования и преобразования. Хотя, по умолчанию, вывод задается в пикселях, существуют и другие возможности. Режимы масштабирования GDI позволяют вам рисовать, задавая размеры в дюймах (иногда, в долях дюйма), в миллиметрах, или других удобных вам единицах измерения;

– метафайлы. Метафайл – это набор вызовов команд GDI, сохраненный в двоичном виде. Метафайлы, в основном, используются для передачи изображений векторной графики через буфер обмена (clipboard);

– регионы. Регион – это сложная область, состоящая из любых фигур, и обычно задаваемая как булева комбинация простых регионов. Регионы, как правило, хранятся внутри GDI как ряды скан-линий, независимо от любой комбинации отрезков, которые могут быть использованы для задания регионов;

– пути. Путь – это набор отрезков и кривых, хранящихся внутри GDI. Они могут использоваться для рисования, закрашивания и при отсечении. Пути могут быть преобразованы в регионы;

– отсечение. Рисование может быть ограничено некоторой областью рабочего пространства окна. Это и называется отсечением, область отсечения может быть прямоугольной или любой другой, какую вы можете описать математически как набор коротких отрезков. Отсечение, как правило, задается регионом или путем;

– палитры. Использование привычных палитр обычно ограничено способностью дисплея показывать не более 256 цветов. Windows резервирует только 20 из этих цветов для использования системой. Вы можете изменять другие 236 цветов для точного отображения красок предметов реального мира как битовые образы;

– печать. Несмотря на то, что эта глава посвящена отображению на экране дисплея, все, чему вы научитесь здесь, относится и к принтерам.

Объекты ядра и их использование в приложении

Процесс (process) представляет собой объект, обладающий собственным независимым виртуальным адресным пространством, в

котором могут размещаться код и данные, защищенные от других процессов. В свою очередь, внутри каждого процесса могут независимо выполняться одна или несколько потоков (threads). Поток, выполняющийся внутри процесса, может сама создавать новые потоки и новые независимые процессы, а также управлять взаимодействием объектов между собой и их синхронизацией.

Создавая процессы и управляя ими, приложения могут организовывать параллельное выполнение нескольких задач, обеспечивающих обработку файлов, проведение вычислений или связь с другими системами в сети. Допускается даже использование нескольких процессоров с целью ускорения обработки данных.

Внутри каждого процесса могут выполняться одна или несколько потоков, и именно поток является базовой единицей выполнения в Windows. Выполнение потоков планируется системой на основе обычных факторов: наличие таких ресурсов, как CPU и физическая память, приоритеты, равнодоступность ресурсов и так далее. Начиная с версии NT4, в Windows поддерживается симметричная многопроцессорная обработка (Symmetric Multiprocessing, SMP), позволяющая распределять выполнение потоков между отдельными процессорами, установленными в системе.

С точки зрения программиста каждому процессу принадлежат ресурсы, представленные следующими компонентами:

– одна или несколько потоков;

– виртуальное адресное пространство, отличное от адресных пространств других процессов, если не считать областей памяти, распределенных явным образом для совместного использования (разделения) несколькими процессами. Заметьте, что разделяемые отображенные файлы совместно используют физическую память, тогда как разделяющие их процессы используют различные виртуальные адресные пространства;

– один или несколько сегментов кода, включая код DLL;

– один или несколько сегментов данных, содержащих глобальные переменные;

– строки, содержащие информацию об окружении, например, информацию о текущем пути доступа к файлам;

– куча процесса;

– различного рода ресурсы, например, дескрипторы открытых файлов и другие кучи.

Поток разделяет вместе с процессом код, глобальные переменные, строки окружения и другие ресурсы. Каждый поток планируется независимо от других и располагает следующими элементами:

– стек, используемый для вызова процедур, прерываний и обработчиков исключений, а также хранения автоматических переменных;

– локальные области хранения потока (Thread Local Storage, SLT) – массивы указателей, используя которые каждый поток может создавать собственную уникальную информационную среду;

– аргумент в стеке, получаемый от создающего потока, который обычно является уникальным для каждого потока;

– структура контекста, поддерживаемая ядром системы и содержащая значения машинных регистров.

Синхронизация

Кроме проблем, связанных с использованием общих глобальных переменных, у потоков могут быть также проблемы доступа к общим системным ресурсам.

Потоки должны взаимодействовать друг с другом в двух основных случаях:

– совместно используя разделяемый ресурс (чтобы не разрушить его);

– когда нужно уведомить другие потоки о завершении каких-либо операций.

Примитив синхронизации – это объект, который помогает управлять многопоточным приложением. В Windows доступны следующие основные типы примитивов синхронизации:

– атомарные операции API-уровня;
– критические секции;
– события;
– ожидаемые таймеры;
– семафоры;
– мьютексы.

Каждый из указанных типов примитивов синхронизации применяется в определенных ситуациях.

Атомарный доступ и семейство Interlocked-функций

Большая часть синхронизации потоков связана с атомарным доступом (atomic access) – монопольным захватом ресурса обращающимся к нему потоком. Win32 API предоставляет несколько функций для реализации взаимно блокированных операций. Все Interlocked-функций работают корректно только при условии, что их аргументы выровнены по границе двойного слова (DWORD).

Функция Interlockedlncrement, имеющая прототип

LONG InterlockedIncrement(LPLONG lpAddend);

Иинкрементирует 32-разрядную переменную, адрес которой задается параметром LpAddend. Функция возвращает новое значение указанной переменной.

Функция InterlockedDecrement определена аналогично функции Interlockedlncrement, но она декрементирует 32-разрядную переменную.

Пара функций

LONG InterlockedExchange(LPLONG IpTarget. LONG Value):

PVOID InterlockedExchangePointer(PVOID* ppvTarget. PVOID pvValue):

монопольно заменяет текущее значение переменной типа LONG, адрес которой передается в первом параметре, значением, передаваемым во втором параметре. В 32-разрядном приложении обе функции работают с 32-разрядными значениями. В 64-разрядной программе первая функция оперирует 32-разрядными значениями, а вторая – 64-разрядными. Обе функции возвращают исходное значение переменной.

Следующая функция добавляет к значению переменной, адрес которой передается в первом параметре, значение, передаваемое во втором параметре:

LONG InterlockedExchangeAdd(LPLONG IpAddend. LONG Increment);

Еще две функции выполняют операцию сравнения и присваивания по результату сравнения:

LONG InterlockedCompareExchangetLPLONG IpDestination. LONG Exchange, LONG Comparand);

PVOID InterlockedCompareExchangePointer(PVOID* ppvDestination, PVOID pvExchange. PVOID pvComparand);

Если значение переменной, адрес которой передается в первом параметре, совпадает со значением, передаваемым в третьем параметре, то оно заменяется значением, передаваемым во втором параметре. В 32-разрядном приложении обе функции работают с 32-разрядными значениями. В 64-разрядной программе первая функция оперирует 32-разрядными значениями, а вторая – 64-разрядными. Обе функции возвращают исходное значение переменной, заданной первым параметром.

Критические секции

Критическая секция (critical section) – это небольшой участок кода, который должен использоваться только одним потоком одновременно. Если в одно время несколько потоков попытаются получить доступ к критическому участку, то контроль над ним будет предоставлен только одному из потоков, а все остальные будут переведены в состояние ожидания до тех пор, пока участок не освободится.

Для использования критической секции необходимо определить переменную типа CRITICAL_SECTION:

```
CRITICAL_SECTION CS;
```

Поскольку эта переменная должна находиться в области видимости для каждого использующего ее потока, обычно она объявляется как глобальная. Эту переменную следует инициализировать до ее первого применения с помощью функции InitializeCriticalSection:

```
InitializeCriticalSection(&cs);
```

Чтобы завладеть критическим участком, поток должен вызвать функцию EnterCriticalSection:

```
EnterCriticalSection(&cs);
```

Если критический участок не используется в данный момент другим потоком, он обозначается системой как свободный, и поток немедленно продолжает выполнение. Если критический участок уже используется, то поток блокируется до тех пор, пока участок не будет освобожден.

После вызова EnterCriticalSection следуют инструкции, принадлежащие критическому участку.

Конец критического участка обозначается вызовом функции LeaveCriticalSection:

```
LeaveCriticalSection(&cs);
```

Как только поток получает контроль над критическим участком, доступ других потоков к этому участку блокируется. При этом очень важно, чтобы время выполнения критического участка было минимальным. Это позволит добиться наилучших результатов работы приложения.

Если критический участок больше не нужен, используемые им ресурсы освобождаются вызовом функции DeleteCriticalSection.

Кроме перечисленных функций Windows предоставляет функцию TryEnterCriticalSection, которая позволяет осуществить попытку захватить критический участок:

```
BOOL bAcquired = TryEnterCriticalSectiот(&cs);
if (bAcquired) {
// Запись в защищенные переменные
}
else {
// Контроль над критическим участком недопустим
}
```

Если критический участок доступен, то поток становится его «владельцем» и осуществляет запись в защищенные переменные. Если участок недоступен, то поток не блокируется, как в случае применения функции EnterCriticalSection, а занимается другой работой, за что отвечает ветвь else.

Работая с критическими секциями или применяя Interlocked-функции, программа не переключается в режим ядра. Поэтому эти виды синхронизации называют синхронизацией в пользовательском режиме. Она отличается высокой скоростью реализации. Главным недостатком такой синхронизации является невозможность ее применения для потоков, принадлежащих разным процессам.

Описываемые ниже примитивы синхронизации основаны на использовании объектов ядра. Но сначала мы должны рассмотреть так называемые wait-функции.

Wait-функции

Многие объекты ядра могут находиться либо в свободном (signaled state), либо в занятом состоянии (nonsignaled state). К таким объектам относятся:

- процессы;
- потоки;
- задания;
- файлы;
- консольный ввод;
- уведомления об изменении файлов;
- события;
- ожидаемые таймеры;
- семафоры;
- мьютексы.

Wait-функции позволяют потоку в любой момент приостановиться и ждать освобождения какого-либо объекта ядра. Важным свойством функций этого семейства является то, что они не тратят процессорное время, пока ждут освобождения объекта или наступления тайм-аута.

Функция WaitForSingleObject

Функция имеет следующий прототип:

DWORD WaitForSingleObject(HANDLE hHandle. DWORD dwMillisecond);

Когда поток вызывает эту функцию, параметр hHandle идентифицирует объект ядра, поддерживающий состояния «свободен-занят». Параметр dwMillisecond задает тайм-аут (time out) — интервал времени, спустя которое функция возвращает управление, даже если объект остается в занятом состоянии. Если параметр dwMillisecond имеет нулевое значение, то функция только проверяет состояние объекта и возвращает управление немедленно. Константа INFINITE в качестве значения dwMillisecond задает бесконечное значение тайм-аута.

Возвращаемым значением функции чаще всего является одна из следующих констант:

– WAIT_OBJECT_0 – контролируемый объект ядра перешел в свободное состояние;

– WAIT_TIMEOUT – истек интервал тайм-аута, а контролируемый объект ядра остается в занятом состоянии;

– WAIT_FAILED – функция завершилась с ошибкой. Для получения дополнительной информации об ошибке нужно использовать функцию GetLastError.

Вот пример обработки кода возврата функции WaitForSingleObject:

```
DWORD dw - WaitForSingleObject(hProcess, 5000):
switch (dw) {
case WAIT_OBJECT_0:
// код обработки для завершившегося процесса
break;
case WAIT_TIMEOUT:
// код обработки, если процесс не завершился в течение 5000 мс
break;
case WAIT_FAILED:
// функция завершилась с ошибкой,
break;
}
```

Будьте осторожны, используя константу INFINITE в качестве второго параметра функции WaitForSingleObject. Если объект hHandle так и не перейдет в свободное состояние, то вызывающий поток никогда не проснется. Единственное «утешение» заключается в том, что тратить драгоценное процессорное время он при этом не будет.

Функция WaitForMultipleObjects

Функция имеет следующий прототип:

```
DWORD WaitForMultipleObjects(DWORD nCount, CONST HANDLE* lpHandles, BOOL fWaitAll. DWORD dwMi11iseconds);
```

Она работает так же, как и функция WaitForSingleObject, но при этом позволяет ждать освобождения сразу нескольких объектов или какого-то одного объекта из заданного списка.

Параметр nCount определяет количество интересующих вас объектов ядра. Его значение должно быть в пределах от 1 до MAXIMUM_WAIT_OBJECTS. В заголовочных файлах Windows эта

константа имеет значение 64. Параметр lpHandles содержит указатель на массив дескрипторов объектов ядра. В массиве могут содержаться дескрипторы объектов разных типов.

Функция WaitForMultipleObjects приостанавливает поток и заставляет его ждать освобождения либо всех заданных объектов ядра, либо одного из них. Параметр fWaitAll определяет поведение функции. Значение TRUE задает режим ожидания освобождения всех указанных объектов, а FALSE – только одного из них. В последнем случае код возврата функции содержит информацию о том, какой именно объект освободился.

Возвращаемое функцией значение сообщает, почему возобновилось выполнение вызвавшего ее потока. Значения WAIT_TIMEOUT и WAIT_FAILED интерпретируются по аналогии с функцией WaitForSingleObject. Если параметр fWaitAll равен TRUE и все объекты перешли в свободное состояние, то функция возвращает значение WAIT_OBJECT_0. Если же fWaitAll имеет значение FALSE, то функция возвращает управление, как только освобождается любой из объектов. При этом ее код возврата лежит в интервале от WAIT_OBJECT_0 до WAIT_OBJECT_0 + nCount - 1. Иными словами, если из кода возврата вычесть константу WAIT_OBJECT_0, то вы получите индекс освободившегося объекта в массиве lpHandles.

События

Событие (event) – самая простая разновидность объектов ядра. Оно содержит счетчик количества пользователей и две булевы переменные. Одна переменная указывает тип данного объекта-события, а другая – его состояние.

События просто уведомляют об окончании какой-либо операции. Объекты-события бывают двух типов: со сбросом вручную (manual-reset events) или с автосбросом (auto-reset events). Первые события позволяют возобновить выполнение сразу нескольких ждущих потоков, а вторые — только одного потока.

Объект ядра «событие» создается функцией CreateEvent, имеющей следующий прототип:

```
HANDLE CreateEvent(
LPSECURITY_ATTRIBUTES eventAttributes, // атрибуты доступа
BOOL bManualReset, // тип сброса
BOOL blnitialState, // начальное состояние
```

LPCTSTR pszName // имя объекта
);

Параметр bManualReset определяет тип объекта-события. Значение TRUE создает событие со сбросом вручную, а значение FALSE – событие с автосбросом.

Параметр blnitialState определяет начальное состояние события – свободное (TRUE) или занятое (FALSE).

Параметр pszName содержит указатель на С-строку, в которой указывается имя объекта. Если pszName имеет значение NULL, то создается неименованный объект.

Например, следующий вызов функции CreateEvent из процесса А создает событие с автосбросом и именем EventName:

HANDLE hEvent = CreateEvent(NULL, FALSE, FALSE, "EventName");

На самом деле все обстоит немножко сложнее. При таком вызове система проверяет, не существует ли уже объект ядра с таким именем. Если подобный объект существует, то ядро проверяет тип этого объекта. Допустим, что некий процесс В уже создал ранее объект-событие с таким же именем EventName. В этом случае система проверяет права доступа процесса А к этому объекту. Если с правами доступа все в порядке, то в таблице дескрипторов процесса А создается новая запись с дескриптором hEvent. То есть процесс А получает свой дескриптор уже существующего именованного объекта-события, а счетчик пользователей этого объекта увеличивается на единицу.

Предположим, что имеет место другая ситуация, когда некий процесс В уже создал ранее объект ядра с таким же именем, но другого типа, например семафор. Тогда функция CreateEvent вернет значение NULL, а если после этого вызвать функцию GetLastError, то она вернет код ошибки, равный 6 (ERROR_INVALID_HANDLE).

Наконец, самая простая ситуация – когда объект ядра с таким именем (EventName) в момент вызова функции CreateEvent не существует. Тогда действительно будет создан новый объект ядра «событие» с именем EventName.

Так что будьте аккуратны с именованием объектов ядра и учитывайте, что пространство имен объектов ядра является общим для объектов всех типов.

Объекты-события могут разделяться разными процессами. Допустим, что некий процесс A создал новое событие hEvent с именем EventName. Потоки из других процессов могут получить доступ к этому объекту несколькими способами:

– вызовом функции CreateEvent с передачей в параметре pszName такого же имени (эта ситуация только что рассматривалась);

– наследованием дескриптора;

– применением функции DuplicateHandle;

– вызовом функции OpenEvent с передачей в параметре pszName такого же имени.

Функция OpenEvent имеет следующий прототип:

```
HANDLE OpenEvent(
DWORD dwDesiredAccess, // требуемый доступ
BOOL blnheritHandle, // опция наследования
LPCTSTR pszName // имя объекта
);
```

Параметр dwDesiredAccess определяет требуемый доступ к объекту-событию. Его значением может быть любая комбинация следующих флагов:

– EVENT_ALL_ACCESS – все возможные флаги доступа к объекту-событию;

– EVENT_MODIFY_STATE – разрешено использование дескриптора объекта в функциях SetEvent и ResetEvent;

– SYNCHRONIZE – разрешено использование дескриптора объекта в любых wait-функциях.

Функция OpenEvent завершится с ошибкой, если атрибуты доступа, заданные для объекта с именем pszName при вызове функции CreateEvent, не будут разрешать доступ, указанный параметром dwDesiredAccess.

Параметр blnheritHandle определяет, разрешено ли наследование данного объекта при создании дочерних процессов. Последний параметр pszName указывает на имя существующего объекта-события.

Если функция завершается успешно, то возвращаемое значение содержит дескриптор существующего объекта-события. Если возникла ошибка, то функция возвращает значение NULL. При помощи функции GetLastError можно уточнить причину возникновения ошибки.

Создав событие или получив дескриптор существующего события, вы можете управлять его состоянием с помощью функции BOOL

SetEvent(HANDLE hEvent), которая переводит объект hEvent в свободное состояние, или с помощью функции BOOL ResetEvent(HANDLE hEvent), которая переводит его в занятое состояние.

Для событий с автосбросом действует следующее правило. Когда ожидание потоком освобождения события успешно завершается, этот объект автоматически сбрасывается в занятое состояние. Для событий со сбросом вручную автоматического сбрасывания не происходит, поэтому для возврата в занятое состояние необходимо вызвать функцию ResetEvent.

Семафоры

Объекты ядра «семафор» (semaphore) используются для учета ресурсов. Кроме счетчика числа пользователей семафор содержит два 32-битных значения со знаком. Одно из них определяет максимальное количество ресурсов, контролируемых семафором, а другое используется как счетчик текущего числа ресурсов.

Объект ядра «семафор» создается вызовом функции CreateSemaphore.

```
HANDLE CreateSemaphore (
LPSECURITY_ATTRIBUTES semaphoreAttributes.
// атрибуты доступа
LONG llnitialCount, // текущее количество доступных ресурсов
LONG lMaximumCount, // максимальное количество ресурсов
LPCTSTR pszName // имя объекта
);
```

Например, в результате следующего вызова:

HANDLE hSem = CreateSemaphore(NULL, 0. 5, "MySemaphore");

будет создан именованный объект-семафор с максимальным числом ресурсов, равным пяти, и изначально нулевым количеством доступных ресурсов.

Поток может увеличить значение счетчика текущего числа доступных ресурсов на величину LReleaseCount, вызывая функцию ReleaseSemaphore.

```
BOOL ReleaseSemaphore(
HANDLE hSemaphore. // дескриптор семафора
LONG IReleaseCount.
```

```
// приращение количества доступных ресурсов
LPLONG lpPreviousCount
// предыдущее значение счетчика ресурсов
);
```

По адресу, указанному в последнем параметре, функция записывает предыдущее значение счетчика ресурсов. Если оно вас не интересует, то просто передайте в параметре lpPreviousCount значение NULL.

Как обычно, любой процесс может получить свой («процессозависимый») дескриптор существующего объекта «семафор», вызвав функцию OpenSemaphore:

```
HANDLE OpenSemaphore (
DWORD dwDesiredAccess, // требуемый доступ
BOOL blnheritHandle, // параметр наследования
LPCTSTR pszName // имя объекта
);
```

Параметр dwDesiredAccess определяет требуемый уровень доступа к объекту-семафору. Его значением может быть любая комбинация следующих флагов:

– SEMAPHORE_ALL_ACCESS – Все возможные флаги доступа к объекту-семафору;

– SEMAPHORE_MODIFY_STATE – Разрешено использование дескриптора объекта в функции ReleaseSemaphore;

– SYNCHRONIZE – Разрешено использование дескриптора объекта в любых wait-функциях.

Если функция завершается успешно, то возвращаемое значение содержит дескриптор существующего объекта-семафора. Если возникла ошибка, функция возвращает значение NULL. При помощи функции GetLastError можно уточнить причину возникновения ошибки.

Поток получает доступ к ресурсу, вызывая одну из wait-функций и передавая ей дескриптор семафора, охраняющего этот ресурс. Wait-функция проверяет у семафора значение счетчика текущего числа ресурсов. Если оно больше нуля и семафор свободен, то значение счетчика уменьшается на единицу, а вызывающий поток продолжает выполнение. Очень важно, что эта операция выполняется для семафора на уровне атомарного доступа. Иначе говоря, пока wait-функция не вернет управление, операционная система не позволит прервать эту операцию никакому другому потоку.

Если wait-функция определяет, что счетчик текущего числа ресурсов равен нулю (семафор занят), то система переводит вызывающий поток в состояние ожидания. Когда другой поток увеличит значение этого счетчика, система вспомнит о ждущем потоке и снова начнет выделять ему процессорное время. А он, захватив ресурс, уменьшит значение счетчика на единицу.

Мьютексы

Объекты ядра «мьютексы» (mutex) гарантируют потокам взаимоисключающий доступ к единственному ресурсу. Кроме счетчика пользователей мьютекс содержит счетчик рекурсии и переменную, в которой запоминается идентификатор потока-владельца. Мьютексы ведут себя подобно критическим секциям. Однако если критические секции являются объектами пользовательского режима, то мьютексы — это объекты ядра. Поэтому они позволяют синхронизировать доступ к ресурсу нескольких потоков из разных процессов.

Для использования объекта-мьютекса один из процессов должен сначала создать его вызовом функции CreateMutex.

```
HANDLE CreateMutex (
LPSECURITY_ATTRIBUTES mutexAttributes,
// атрибуты доступа
BOOL blnitialOwner, // флаг наличия потока-владельца
LPCTSTR pszName // имя объекта
);
```

Параметр blnitialOwner определяет начальное состояние мьютекса. Если он имеет значение FALSE, то объект-мьютекс не принадлежит ни одному из потоков и поэтому находится в свободном состоянии. При этом его идентификатор потока-владельца и счетчик рекурсии равны нулю. Если же в этом параметре передается значение TRUE, то идентификатор потока-владельца в мьютексе приравнивается идентификатору вызывающего потока, а счетчик рекурсии получает единичное значение. Если идентификатор потока-владельца не равен нулю, мьютекс считается занятым.

Разумеется, любой процесс может получить свой («процессозависимый») дескриптор существующего объекта «мьютекс», вызвав функцию OpenMutex:

HANDLE OpenMutex(DWORD dwDesiredAccess, BOOL blnheritHandle, LPCTSTR pszName);

Значением параметра dwDesiredAccess может быть любая комбинация флагов:

– MUTEX_ALL_ACCESS – все возможные флаги доступа к объекту-мьютексу;

– SYNCHRONIZE – разрешено использование дескриптора объекта в любых wait-функциях, а также при вызове ReleaseMutex .

Если функция завершается успешно, то возвращаемое значение содержит дескриптор существующего объекта-мьютекса. В случае возникновения ошибки функция возвращает значение NULL.

Для получения доступа к разделяемому ресурсу поток вызывает одну из wait-функций и передает ей дескриптор мьютекса, охраняющего этот ресурс. Wait-функция проверяет у мьютекса идентификатор потока-владельца. Если идентификатор равен нулю, то ресурс свободен и вызывающий поток может продолжить выполнение. В этом случае перед возвратом из wait-функции идентификатор потока-владельца в мьютексе становится равным идентификатору вызывающего потока, а счетчику рекурсии присваивается единичное значение.

Если wait-функция определяет, что идентификатор потока-владельца не равен нулю, то вызывающий поток переходит в состояние ожидания. Система запоминает это и, когда идентификатор потока-владельца обнуляется (а перед этим обнуляется и счетчик рекурсии), записывает в него идентификатор ожидающего потока, а счетчику рекурсии присваивает единичное значение. После этого ожидающий поток вновь становится планируемым. Все проверки и изменения состояния объекта-мьютекса выполняются на уровне атомарного доступа.

Как только разделяемый ресурс перестает быть нужным, поток, владеющий мьютексом, должен его освободить, вызвав функцию ReleaseMutex. Функция ReleaseMutex уменьшает значение счетчика рекурсии в мьютексе на единицу. Если счетчик рекурсии становится равным нулю, то обнуляется также идентификатор потока-владельца.

Система допускает рекурсивный многократный захват мьютекса одним потоком. Это происходит, если, уже владея мьютексом и не освободив его, поток вызывает wait-функцию с дескриптором этого же мьютекса. В этом случае счетчик рекурсии в мьютексе каждый раз увеличивается на единицу. Для освобождения мьютекса количество вызовов потоком функции ReleaseMutex должно совпадать с количеством предыдущих захватов мьютекса.

Разработка и использование динамически загружаемых библиотек

DLL (англ. Dynamic-link library – библотека динамической компоновки) – понятие операционных систем Microsoft Windows и IBM OS/2; динамическая библиотека, позволяющая многократное применение различными программными приложениями. К DLL относятся также элементы управления ActiveX и драйверы.

Неявное связывание (Implicit linking) – самый распространенный в настоящее время способ загрузки DLL-библиотек. Суть его заключается в том, что компоновщик при построении исполняемого exe-файла не включает код функций в тело программы, а создает раздел импорта, где перечисляются символические имена функций и переменных для каждой из DLL-библиотек. Для этого к проекту необходимо подключить соответствующий lib-файл. При запуске программы загрузчик операционной системы анализирует раздел импорта, загружает все необходимые DLL-библиотеки и, что важно, проецирует их на адресное пространство загружаемого приложения. Причем, если в загружаемой DLL-библиотеке существует свой раздел импорта, то загружаются и те dll-файлы, которые там указаны. Однако если библиотека один раз уже загружена, то второй раз она не загружается, а строится ссылка на уже существующий экземпляр.

Поиск DLL-библиотек осуществляется по стандартной схеме:

- в папке, откуда запущено приложение;
- в текущей папке;
- в системной папке Windows\system32;
- в папке Windows;
- в папках, которые перечислены в переменной окружения PATH.

Если библиотека не найдена, загрузка приложения прекращается и выводится сообщение об ошибке.

Явная загрузка и связывание требуемой DLL в период выполнения приложения.

Иначе говоря, его поток явно загружает DLL в адресное пространство процесса, получает виртуальный адрес необходимой DLL-функции и вызывает ее по этому адресу. Изящество такого подхода в том, что все происходит в уже выполняемом приложении.

При явном подключении DLL программист должен сам позаботиться о загрузке библиотеки перед ее использованием.

HMODULE WINAPI LoadLibrary (LPCTSTR lpLibFileName);

Так как DLL загружается в память, после работы с ней необходимо ее выгрузить.

BOOL FreeLibrary(HMODULE hModule   );

После того как библиотека загружена, адрес любой из содержащихся в ней функций можно получить с помощью GetProcAddress, которой необходимо передать дескриптор библиотеки и имя функции. Затем функцию из DLL можно вызывать, как обычно.

FARPROC GetProcAddress (HMODULE hModule, LPCSTR lpProcName);

GetModuleHandle получает указатель на DLL.

HMODULE GetModuleHandle(LPCTSTR lpModuleName // Имя модуля);

Эта функция возвращает указатель на модуль, если он загружен.

Функция GetModuleFileName получает полный путь и имя DLL.

DWORD GetModuleFileName(HMODULE hModule, LPTSTR lpFilename, DWORD nSize);

Механизмы управления виртуальной и динамически распределяемой памятью

Необходимым условием для того, чтобы программа могла выполняться, является ее нахождение в оперативной памяти. Только в этом случае процессор может извлекать команды из памяти и интерпретировать их, выполняя заданные действия. Объем оперативной памяти, который имеется в компьютере, существенно сказывается на характере протекания вычислительного процесса. Он ограничивает число одновременно выполняющихся программ и размеры их виртуальных адресных пространств. В некоторых случаях, когда все задачи мультипрограммной смеси являются вычислительными (то есть выполняют относительно мало операций ввода-вывода, разгружающих

центральный процессор), для хорошей загрузки процессора может оказаться достаточным всего 3-5 задач. Однако если вычислительная система загружена выполнением интерактивных задач, то для эффективного использования процессора может потребоваться уже несколько десятков, а то и сотен задач.

Большое количество задач, необходимое для высокой загрузки процессора, требует большого объема оперативной памяти. В условиях, когда для обеспечения приемлемого уровня мультипрограммирования имеющейся оперативной памяти недостаточно, был предложен метод организации вычислительного процесса, при котором образы некоторых процессов целиком или частично временно выгружаются на диск.

В мультипрограммном режиме помимо активного процесса, то есть процесса, коды которого в настоящий момент интерпретируются процессором, имеются приостановленные процессы, находящиеся в ожидании завершения ввода-вывода или освобождения ресурсов, а также процессы в состоянии готовности, стоящие в очереди к процессору. Образы таких неактивных процессов могут быть временно, до следующего цикла активности, выгружены на диск. Несмотря на то что коды и данные процесса отсутствуют в оперативной памяти, ОС «знает» о его существовании и в полной мере учитывает это при распределении процессорного времени и других системных ресурсов. К моменту, когда подходит очередь выполнения выгруженного процесса, его образ возвращается с диска в оперативную память. Если при этом обнаруживается, что свободного места в оперативной памяти не хватает, то на диск выгружается другой процесс.

Такая подмена (виртуализация) оперативной памяти дисковой памятью позволяет повысить уровень мультипрограммирования – объем оперативной памяти компьютера теперь не столь жестко ограничивает количество одновременно выполняемых процессов, поскольку суммарный объем памяти, занимаемой образами этих процессов, может существенно превосходить имеющийся объем оперативной памяти. Виртуальным называется ресурс, который пользователю или пользовательской программе представляется обладающим свойствами, которыми он в действительности не обладает. В данном случае в распоряжение прикладного программиста предоставляется виртуальная оперативная память, размер которой намного превосходит всю имеющуюся в системе реальную оперативную память. Пользователь пишет программу, а транслятор, используя виртуальные адреса, переводит ее в машинные коды так, как будто в распоряжении программы имеется однородная оперативная память большого объема. В действительности же все коды и

данные, используемые программой, хранятся на дисках и только при необходимости загружаются в реальную оперативную память. Понятно, однако, что работа такой «оперативной памяти» происходит значительно медленнее.

Виртуализация оперативной памяти осуществляется совокупностью программных модулей ОС и аппаратных схем процессора и включает решение следующих задач:

– размещение данных в запоминающих устройствах разного типа, например часть кодов программы – в оперативной памяти, а часть – на диске;

– выбор образов процессов или их частей для перемещения из оперативной памяти на диск и обратно;

– перемещение по мере необходимости данных между памятью и диском;

– преобразование виртуальных адресов в физические.

Виртуализация памяти может быть осуществлена на основе двух различных подходов:

– свопинг (swapping) – образы процессов выгружаются на диск и возвращаются в оперативную память целиком;

– виртуальная память (virtual memory) – между оперативной памятью и диском перемещаются части (сегменты, страницы и т. п.) образов процессов.

Свопинг представляет собой частный случай виртуальной памяти и, следовательно, более простой в реализации способ совместного использования оперативной памяти и диска. Однако подкачке свойственна избыточность: когда ОС решает активизировать процесс, для его выполнения, как правило, не требуется загружать в оперативную память все его сегменты полностью – достаточно загрузить небольшую часть кодового сегмента с подлежащей выполнению инструкцией и частью сегментов Данных, с которыми работает эта инструкция, а также отвести место под сегмент стека. Аналогично при освобождении памяти для загрузки нового процесса очень часто вовсе не требуется выгружать другой процесс на диск целиком, достаточно вытеснить на диск только часть его образа. Перемещение избыточной информации замедляет работу системы, а также приводит к неэффективному использованию памяти. Кроме того, системы, поддерживающие свопинг, имеют еще один очень существенный недостаток: они не способны загрузить для выполнения процесс, виртуальное адресное пространство которого превышает имеющуюся в наличии свободную память.

Гридина Елена Ивановна

СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

ПОСОБИЕ
по одноименной дисциплине для слушателей специальностей 1-40 01 73 «Программное обеспечение информационных систем» заочной формы обучения

Подписано к размещению в электронную библиотеку
ГГТУ им. П. О. Сухого в качестве электронного
учебно-методического документа 17.07.17.

Рег. № 95Е.
http://www.gstu.by