

Министерство образования и науки Российской Федерации
Федеральное агентство по образованию
Ярославский государственный университет им. П. Г. Демидова

В. В. Васильчиков

**Разработка
сетевых приложений
для ОС Windows
(практические примеры)**

Учебное пособие

*Рекомендовано
Научно-методическим советом университета
для студентов, обучающихся по специальностям
Прикладная математика и информатика
и Математическое обеспечение
и администрирование информационных систем*

Ярославль 2009

УДК 519.2
ББК 3973.2-018я73
В 19

*Рекомендовано
Редакционно-издательским советом университета
в качестве учебного издания. План 2009 года*

Рецензенты:

кандидат физико-математических наук С. И. Щукин;
кафедра теории и методики обучения информатике ЯГПУ
им. К. Д. Ушинского

В 19 **Васильчиков, В. В.** Разработка сетевых приложений для ОС Windows (практические примеры) : учебное пособие / В. В. Васильчиков ; Яросл. гос. ун-т им. П. Г Демидова. – Ярославль : ЯрГУ, 2009. – 214 с.
ISBN 978-5-8397-0659-0

На практических примерах рассмотрены основные моменты разработки сетевых Windows-приложений для платформы Win32 с использованием среды Microsoft Visual Studio и библиотеки MFC.

Предназначено для студентов, обучающихся по специальностям 010501 Прикладная математика и информатика и 010503 Математическое обеспечение и администрирование информационных систем (дисциплина специализации "Программирование в сетях Windows", блок ДС, СД), очной формы обучения.

Библиогр.: 7 назв.

УДК 519.2
ББК 3973.2-018я73

ISBN 978-5-8397-0659-0

© Ярославский
государственный
университет
им. П. Г. Демидова, 2009

Введение

В 2007 году автором было издано учебное пособие [7], предназначенное для поддержки лекционного курса по программированию в сетях Windows, читавшегося для студентов факультета ИВТ ЯрГУ, обучающихся по специальностям "Прикладная математика и информатика" и "Математическое обеспечение и администрирование информационных систем". В нем были рассмотрены существующие технологии и Win32 API-функции, использующиеся для создания сетевых приложений различных версий ОС Windows. В упомянутом учебном пособии, разумеется, присутствовали практические примеры для иллюстрации рассматриваемых средств и приемов программирования. Все примеры программ были написаны на языке С как консольные приложения и не были привязаны к использованию какой-либо конкретной среды программирования.

Использование таких примеров, как показал опыт преподавания данной дисциплины, не слишком удобно: в них отсутствует привычный графический интерфейс, они сложнее воспринимаются как при отладке, так и при выполнении. При переносе этого кода в приложение с традиционным Windows-интерфейсом приходится вносить много изменений в проекты. Например, использование библиотеки MFC накладывает изрядное количество ограничений на взаимодействие объектов в многопоточной среде, способы порождения и синхронизации потоков. Кроме того, нужно знать, какие библиотеки следует подключать для того, чтобы воспользоваться теми или иными возможностями.

Поэтому в настоящем учебном пособии автору хотелось бы привести примеры программирования сетевых приложений именно с Windows-интерфейсом и с использованием библиотеки MFC. Собственно, на учебных занятиях в компьютерных классах именно эти примеры в настоящее время и используются.

Все исходные коды, сгруппированные по темам учебного курса, доступны в локальной сети факультета. Проекты структурированы по темам учебного курса. Предполагаемый результат находится в папке Solution. Стартовый проект (заготовка с предлагаемым интерфейсом) находится в папке Starter. Как правило, стартовый проект содержит пользовательский интерфейс, чтобы студенты не тратили время на его разработку. В ряде случаев в нем также проделана часть рутинной работы: подключение библиотек, заголовочных файлов, объявление некоторых констант и переменных. Иногда также присутствует реализация некоторых обработчиков событий. Инструкции по выполнению задания помещены в файл ToDo.doc. Там используется следующее соглашение о шрифтах: код, который должен быть добавлен в проект, набран полужирным шрифтом. Обычный шрифт означает, что данный код в проекте уже присутствует.

Во избежание недоразумений автору хотелось бы обратить внимание на одну особенность нумерации предлагаемых примеров программ. Поскольку некоторые из рассматриваемых в учебном курсе тем не предполагают выполнения практических заданий, названия глав в данном пособии не полностью соответствуют таковым в учебном пособии [7] (некоторые выпущены). Однако для удобства студентов нумерация примеров базируется на номерах тем учебного курса и потому не является сплошной.

Изначально в качестве среды разработки предполагалось использование Microsoft Visual Studio версии 6.0, поскольку именно эта среда выступает в качестве основной при изучении курса "Программирование в Windows". Однако все проекты легко (автоматически) конвертируются для использования в более свежих версиях Visual Studio. По крайней мере, вплоть до Visual Studio 2008 никаких проблем с преобразованием проектов автор не заметил.

Следует отметить, что в настоящем учебном пособии нет никаких сведений теоретического плана – только практические примеры. Вся необходимая предварительная информация содержится, например, в учебном пособии [7]. Кроме того, предполагается, что читатели знают язык программирования C++, умеют пользоваться одной из версий среды Microsoft Visual Studio, имеют опыт работы с библиотекой MFC. Особое внимание следует обратить на навыки разработки многопоточных MFC-приложений, понимание и умение использовать средства синхронизации, предоставляемые этой библиотекой и Win32 API. В качестве издания, где в достаточно компактном виде изложен весь необходимый материал, можно порекомендовать [6].

Во всех примерах для организации работы с сетью используются только Win32 API-функции. Читателям, интересующимся более высокоуровневыми средствами, например классами библиотеки MFC для работы в сети, можно порекомендовать обратиться к книгам Олафсена, Скрайбера и Уайта [3], Круглински, Уингоу и Шеферда [4], а также к документации MSDN. Однако для понимания устройства и работы этих классов все же настоятельно рекомендуется изучить использование соответствующих функций Win32 API.

Интерфейс NetBIOS

В рамках данной темы предполагается отработать основные операции для организации передачи данных с использованием сетевого интерфейса NetBIOS.

Сначала рассматривается вопрос об организации передачи данных в рамках модели "сервер – клиент" с установлением соединения. Для этого мы разработаем четыре приложения:

- библиотеку общих функций для приложений NetBIOS (используются как сервером, так и клиентом);
- клиентское приложение, предназначенное для работы с эхо-сервером;
- эхо-сервер на основе использования функций обратного вызова;
- эхо-сервер, основанный на модели событий.

Далее разбирается пример организации передачи данных в виде дейтаграмм без установления постоянного соединения. В качестве примера читателям предлагается разработать приложение, которое может выступать в роли как отправителя, так и получателя дейтаграмм.

Пример 1.1. Библиотека общих функций для приложений NetBIOS

Конечной целью данного задания является разработка библиотеки функций, выполняющих типичные для приложения NetBIOS задачи: работа с номерами LANA, настройка среды NetBIOS, работа с таблицами имен и т. п. Эта библиотека (результатирующий файл NbCommon.lib) будет нами использоваться для разработки уже конкретных приложений, использующих интерфейс NetBIOS.

Создание проекта

С использованием мастера AppWizard сгенерируем проект статической Win32 библиотеки с именем NbCommon. В процессе прохождения шагов генерации отметим флажок поддержки MFC, но не будем использовать Pre-compiled Header.

Заголовочный файл библиотеки

Файл nbcommon.h содержит директивы для подключения необходимых системных заголовочных файлов, а также прототипы функций, образующих нашу библиотеку. Модификатор extern "C" необходим для успешной компоновки в Visual Studio приложений, использующих данную библиотеку.

Содержание файла nbcommon.h приводится ниже:

```
#include <windows.h>
#include <nb30.h>
```

```
extern "C" int Recv(int lana, int lsn, char *buffer,
                  DWORD *len);
extern "C" int Send(int lana, int lsn, char *data,
                  DWORD len);
extern "C" int AddName(int lana, char *name, int *num);
extern "C" int DelName(int lana, char *name);
extern "C" int AddGroupName(int lana, char *name, int *num);
extern "C" int ResetAll(LANA_ENUM *lenum,
                       UCHAR ucMaxSession, UCHAR ucMaxName, BOOL bFirstName);
extern "C" int LanaEnum(LANA_ENUM *lenum);
extern "C" int Hangup(int lana, int lsn);
extern "C" int Cancel(PNCB pncb);
extern "C" int FormatNetbiosName(char *nbname,
                                char *outname);

extern "C" char NbCommonErrorMsg[];
```

Строка NbCommonErrorMsg предназначена для записи сообщения с описанием последней произошедшей ошибки для того, чтобы вызывающая функция могла вывести это сообщение с учетом используемого ею пользовательского интерфейса. В начале каждой функции эта строка очищается. Наверное, можно было предложить и более изящный способ передачи сообщения об ошибке, но лучше сосредоточиться на выполнении основных задач.

Обратите также внимание, что файл nbcommon.h прописан в проекте библиотеки NbCommon, но нигде не подключается (это вызовет ошибки при компиляции), он предназначен только для других проектов, импортирующих функции данной библиотеки.

Основной модуль библиотеки

Имя файла основного модуля значения не имеет, пусть он называется, например, nbcommon.c.

В начале файла поместим необходимые директивы препроцессора и объявление строки для описания последней ошибки:

```
#include <windows.h>
#include <stdio.h>

char NbCommonErrorMsg[200];
```

Далее помещаем исходный код перечисленных выше основных функций библиотеки.

Назначение и исходный код библиотечных функций

Функция LANAEnum.

Функция предназначена для перечисления всех номеров LANA.

```
int LANAEnum(LANA_ENUM *lenum)
{
    NCB          ncb;

    NbCommonErrorMsg[0]=0;
    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBENUM;
    ncb.ncb_buffer = (PUCHAR)lenum;
    ncb.ncb_length = sizeof(LANA_ENUM);

    if (Netbios(&ncb) != NRC_GOODRET)
    {
        sprintf(NbCommonErrorMsg,
                "ERROR: Netbios: NCBENUM: %d\n",
                ncb.ncb_retcode);
        return ncb.ncb_retcode;
    }
    return NRC_GOODRET;
}
```

Функция ResetAll.

Функция осуществляет сбор сведений о LANA, перечисленных в структуре LANA_ENUM, а также настройку среды NetBIOS: максимальное число сеансов, максимальный размер таблицы имен, использование первого NetBIOS имени.

```
int ResetAll(LANA_ENUM *lenum, UCHAR ucMaxSession,
             UCHAR ucMaxName, BOOL bFirstName)
{
    NCB      ncb;
    int      i;

    NbCommonErrorMsg[0]=0;
    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBRESET;
    ncb.ncb_callname[0] = ucMaxSession;
    ncb.ncb_callname[2] = ucMaxName;
    ncb.ncb_callname[3] = (UCHAR)bFirstName;
```

```

for(i = 0; i < lenum->length; i++)
{
    ncb.ncb_lana_num = lenum->lana[i];
    if (Netbios(&ncb) != NRC_GOODRET)
    {
        sprintf(NbCommonErrorMsg,
            "ERROR: Netbios: NCBRESET[%d]: %d\n",
            ncb.ncb_lana_num, ncb.ncb_retcode);
        return ncb.ncb_retcode;
    }
}
return NRC_GOODRET;
}

```

Функция AddName.

Функция добавляет указанное имя данному номеру LANA, возвращает номер зарегистрированного имени.

```

int AddName(int lana, char *name, int *num)
{
    NCB          ncb;

    NbCommonErrorMsg[0]=0;
    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBADDNAME;
    ncb.ncb_lana_num = lana;
    memset(ncb.ncb_name, ' ', NCBNAMSZ);
    strncpy(ncb.ncb_name, name, strlen(name));

    if (Netbios(&ncb) != NRC_GOODRET)
    {
        sprintf(NbCommonErrorMsg,
            "ERROR: Netbios: NCBADDNAME[lana=%d;name=%s]: %d\n",
            lana, name, ncb.ncb_retcode);
        return ncb.ncb_retcode;
    }
    *num = ncb.ncb_num;
    return NRC_GOODRET;
}

```

Функция AddGroupName.

Функция добавляет указанное групповое имя NetBIOS данному номеру LANA, возвращает номер добавленного имени.


```

int AddGroupName(int lana, char *name, int *num)
{
    NCB          ncb;

    NbCommonErrorMsg[0]=0;
    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBADDGRNAME;
    ncb.ncb_lana_num = lana;
    memset(ncb.ncb_name, ' ', NCBNAMSZ);
    strncpy(ncb.ncb_name, name, strlen(name));

    if (Netbios(&ncb) != NRC_GOODRET)
    {
        sprintf(NbCommonErrorMsg,
            "ERROR: Netbios: NCBADDGRNAME[lana=%d;name=%s]: %d\n",
            lana, name, ncb.ncb_retcode);
        return ncb.ncb_retcode;
    }
    *num = ncb.ncb_num;
    return NRC_GOODRET;
}

```

Функция DelName.

Функция выполняет удаление указанного имени NetBIOS из таблицы имен для данного номера LANA.

```

int DelName(int lana, char *name)
{
    NCB          ncb;

    NbCommonErrorMsg[0]=0;
    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBDELNAME;
    ncb.ncb_lana_num = lana;
    memset(ncb.ncb_name, ' ', NCBNAMSZ);
    strncpy(ncb.ncb_name, name, strlen(name));

    if (Netbios(&ncb) != NRC_GOODRET)
    {
        sprintf(NbCommonErrorMsg,
            "ERROR: Netbios: NCBADDNAME[lana=%d;name=%s]: %d\n",
            lana, name, ncb.ncb_retcode);
        return ncb.ncb_retcode;
    }
    return NRC_GOODRET;
}

```

Функция Send.

Функция выполняет отправку len байтов по указанному сеансу (lsn) и номеру LANA.

```
int Send(int lana, int lsn, char *data, DWORD len)
{
    NCB      ncb;
    int      retcode;

    NbCommonErrorMsg[0]=0;
    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBSEND;
    ncb.ncb_buffer = (PUCHAR)data;
    ncb.ncb_length = len;
    ncb.ncb_lana_num = lana;
    ncb.ncb_lsn = lsn;

    retcode = Netbios(&ncb);

    return retcode;
}
```

Функция Recv.

Функция выполняет прием до len байтов по указанному сеансу (lsn) и номеру LANA.

```
int Recv(int lana, int lsn, char *buffer, DWORD *len)
{
    NCB      ncb;

    NbCommonErrorMsg[0]=0;
    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBRCV;
    ncb.ncb_buffer = (PUCHAR)buffer;
    ncb.ncb_length = *len;
    ncb.ncb_lana_num = lana;
    ncb.ncb_lsn = lsn;

    if (Netbios(&ncb) != NRC_GOODRET)
    {
        *len = -1;
        return ncb.ncb_retcode;
    }
    *len = ncb.ncb_length;

    return NRC_GOODRET;
}
```

Функция Hangup.

Функция выполняет прекращение указанного сеанса (lsn) на данном номере LANA.

```
int Hangup(int lana, int lsn)
{
    NCB      ncb;
    int      retcode;

    NbCommonErrorMsg[0]=0;
    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBHANGUP;
    ncb.ncb_lsn = lsn;
    ncb.ncb_lana_num = lana;

    retcode = Netbios(&ncb);

    return retcode;
}
```

Функция Cancel.

Функция отменяет асинхронную команду, указанную в структуре NCB.

```
int Cancel(PNCB pncb)
{
    NCB      ncb;

    NbCommonErrorMsg[0]=0;
    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBCANCEL;
    ncb.ncb_buffer = (PUCHAR)pncb;
    ncb.ncb_lana_num = pncb->ncb_lana_num;

    if (Netbios(&ncb) != NRC_GOODRET)
    {
        sprintf(NbCommonErrorMsg,
            "ERROR: NetBIOS: NCBCANCEL: %d\n", ncb.ncb_retcode);
        return ncb.ncb_retcode;
    }
    return NRC_GOODRET;
}
```

Функция FormatNetbiosName.

Функция осуществляет форматирование указанного имени NetBIOS для печати. Размер выходного буфера outname должен быть не менее NCBNAMSZ + 1 символов.

```
int FormatNetbiosName(char *nbname, char *outname)
{
    int    i;

    NbCommonErrorMsg[0]=0;
    strncpy(outname, nbname, NCBNAMSZ);
    outname[NCBNAMSZ - 1] = '\\0';
    for(i = 0; i < NCBNAMSZ - 1; i++)
    {
        // Непечатные символы заменяются точками
        //
        if (!((outname[i] >= 32) && (outname[i] <= 126)))
            outname[i] = '.';
    }
    return NRC_GOODRET;
}
```

Пример 1.2. Эхо-клиент на основе сетевого интерфейса NetBIOS

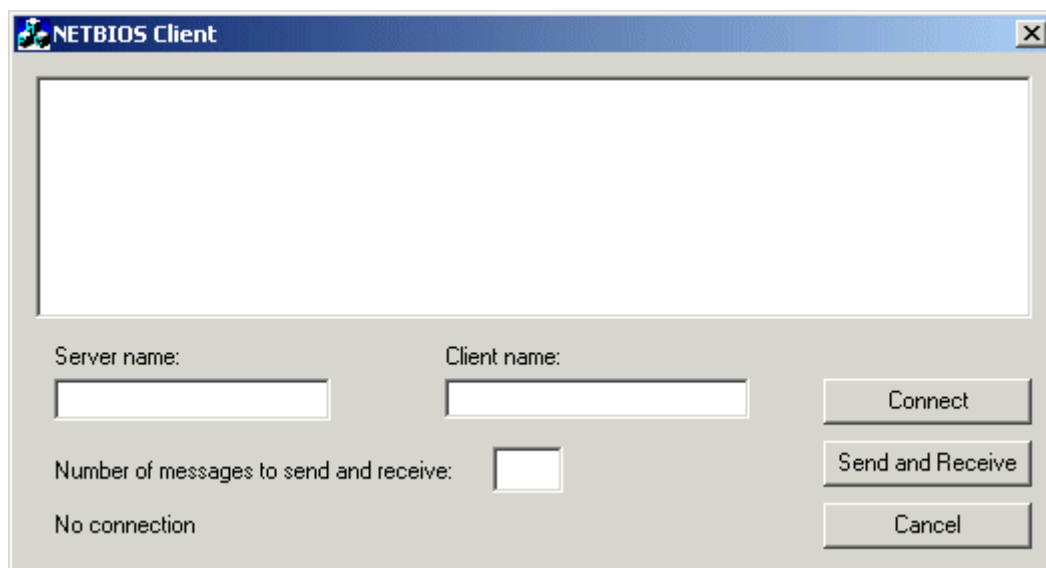
Данный клиент рассчитан на работу с эхо-сервером: он посылает строку и считывает ответ. Все это для простоты делается в одном потоке. Таким образом, если ответа не будет, то работа программы будет блокирована на функции приема. Не произойдет даже завершения функции обработки нажатия кнопки "Send and Receive" и соответственно появления сообщений, выведенных данной функцией в диалоговое окно. Студентам предлагается по желанию организовать рабочие потоки для выполнения действий, которые могли бы блокировать работу приложения. В большинстве последующих примеров мы именно так и будем поступать, однако здесь для простоты данный момент (весьма важный для разработки реальных приложений) опускаем – у нас пока иные цели.

Открытие проекта

Откройте файл рабочего пространства заготовки проекта Client.dsw, расположенный в директории 01_NetBIOS\Client\Starter.

Главное диалоговое окно

Проект организован как диалоговое приложение с главным окном следующего вида:



Ниже перечислены основные элементы управления, расположенные на главном диалоговом окне приложения, их атрибуты и назначение.

Список в верхней части окна (идентификатор IDC_LISTBOX) будет использоваться для вывода сообщений.

Поля ввода (идентификаторы IDC_SERVER, IDC_CLIENT и IDC_NUMBER) предназначены для ввода имени сервера, с которым устанавливается связь, имени клиента и количества сообщений, которое требуется отправить и соответственно принять. Метка с надписью "No connection" (идентификатор IDC_CURRENT_CONN) будет использоваться для вывода информации о текущем состоянии соединения.

Кнопка "Connect" (идентификатор IDC_CONNECT) предназначена для установки связи с сервером, а кнопка "Send and Receive" (идентификатор IDC_SEND) – для запуска процесса отправки сообщений и приема ответов на них. Назначение кнопки "Cancel", как полагает автор, в комментариях не нуждается, и в дальнейшем она упоминаться не будет.

Добавление файлов, подключение библиотек, объявление констант и глобальных переменных

Скопируем в папку проекта nbcommon.h и NbCommon.lib и добавим их в проект.

В окне Project Settings на вкладке Link в поле Object/library modules: подключим библиотеку netapi32.lib.

Подключим заголовок библиотеки NbCommon в файле ClientDlg.h:

```
// ...
#include "nbcommon.h"
////////////////////////////////////
// CClientDlg dialog
```

В начале файла ClientDlg.cpp добавим необходимые объявления констант и глобальных переменных:

```
#define MAX_SESSIONS    254
#define MAX_NAMES       254
#define MAX_BUFFER      1024
```

```
HANDLE    *hArray;
NCB       *pnpcb;
DWORD     dwIndex;
char      szServerName [NCBNAMSZ] ;
char      szClientName [NCBNAMSZ] ;
char      Str[200];
```

```
LANA_ENUM    lenum;
```

Определение переменных и функций-членов класса CClientDlg

Добавим переменную `m_IsConnected` типа `bool` для указания, было ли уже произведено подключение к серверу. Также, используя `ClassWizard`, добавим целую переменную `m_Number`, связанную с полем `IDC_NUMBER`, установив пределы ее изменения от 1 до 20.

Объявим также функцию `SetConnected()` для изменения состояния элементов управления в зависимости от наличия подключения и функцию `Connect()` для установки соединения с сервером на заданном номере `LANA`.

```
void SetConnected(bool IsConnected);
int Connect(PNCB pnpcb, int lana, char *server,
            char *client);
```

Код функции `SetConnected()` (в файле реализации класса) может выглядеть примерно так:

```
void CClientDlg::SetConnected(bool IsConnected)
{
    m_IsConnected = IsConnected;

    GetDlgItem(IDC_SEND) ->EnableWindow(IsConnected);
    GetDlgItem(IDC_NUMBER) ->EnableWindow(IsConnected);
    GetDlgItem(IDC_SERVER) ->EnableWindow(!IsConnected);
    GetDlgItem(IDC_CLIENT) ->EnableWindow(!IsConnected);
    GetDlgItem(IDC_CONNECT) ->EnableWindow(!IsConnected);

    if (IsConnected)
    {
        char Str[180], Server[81], Client[81];
        GetDlgItem(IDC_SERVER) ->GetWindowText(Server, 80);
        GetDlgItem(IDC_CLIENT) ->GetWindowText(Client, 80);
        sprintf (Str, "Connected: Server: %s Client:%s",
                Server, Client);
```

```

        GetDlgItem(IDC_CURRENT_CONN) ->SetWindowText(Str);
    }
    else
    {
        GetDlgItem(IDC_CURRENT_CONN)
            ->SetWindowText ("No connection");
    }
}

```

Код реализации функции Connect():

```

//
// Функция: Connect
// Устанавливает соединение с сервером на заданном номере LANA.
// В переданной структуре NCB поле ncb_event уже заполнено
int CClientDlg::Connect
    (PNCB pncb, int lana, char *server, char *client)
{
    pncb->ncb_command = NCBCALL | ASYNCH;
    pncb->ncb_lana_num = lana;

    memset(pncb->ncb_name, ' ', NCBNAMSZ);
    strncpy((char *)pncb->ncb_name, client, strlen(client));

    memset(pncb->ncb_callname, ' ', NCBNAMSZ);
    strncpy((char *)pncb->ncb_callname, server,
        strlen(server));

    if (Netbios(pncb) != NRC_GOODRET)
    {
        sprintf (Str, "ERROR: Netbios: NCBCONNECT: %d",
            pncb->ncb_retcode);
        ((CListBox *)GetDlgItem(IDC_LISTBOX))
            ->AddString(Str);
        return pncb->ncb_retcode;
    }
    return NRC_GOODRET;
}

```

В коде функции CClientDlg::OnInitDialog() настроим главное окно диалога перед его первым появлением:

```

// ...
// TODO: Add extra initialization here
GetDlgItem(IDC_SERVER) ->SetWindowText ("TEST-SERVER");
GetDlgItem(IDC_CLIENT) ->SetWindowText ("TEST-CLIENT");
SetConnected(false);
// ...

```

Определение обработчиков событий

Теперь необходимо написать обработчики событий, получаемых главным диалоговым окном при нажатии кнопок "Connect" (идентификатор IDC_CONNECT) и "Send and Receive" (идентификатор IDC_SEND).

Обработчик нажатия кнопки "Connect" должен инициализировать интерфейс NetBIOS, распределить ресурсы, затем дать команду NCBCALL для каждого номера LANA на сервере. Когда соединение создано, он должен отменить или разорвать неудавшиеся соединения.

Соответствующий код может выглядеть, например, так. Сначала опишем необходимые переменные. В переменную pLB для удобства запишем адрес окна для вывода сообщений

```
DWORD      dwNum;
DWORD      i;
CListBox *  pLB = ((CListBox *)GetDlgItem(IDC_LISTBOX));
```

Далее перечислим и сбросим все номера LANA, а также запомним имена сервера и клиента. В случае, если номеров LANA не обнаружено (например, если в системе не установлена поддержка NetBIOS), мы сделаем неактивными все элементы управления, кроме кнопки Cancel, и завершим работу функции. В случае неуспешного завершения вызова функции NetBIOS в окно выводится сформированное ею сообщение об ошибке.

```
if (LanaEnum(&lenum) != NRC_GOODRET)
{
    if (*NbCommonErrorMsg)
        pLB->AddString(NbCommonErrorMsg);
    sprintf(Str, "LanaEnum failed with error %d:",
            GetLastError());
    pLB->AddString(Str);
    return;
}

if (lenum.length == 0)
{
    pLB->AddString("Sorry, no existing LANA...");
    GetDlgItem(IDC_SERVER)->EnableWindow(false);
    GetDlgItem(IDC_CLIENT)->EnableWindow(false);
    GetDlgItem(IDC_CONNECT)->EnableWindow(false);
    return;
}

if (ResetAll(&lenum, (UCHAR)MAX_SESSIONS,
            (UCHAR)MAX_NAMES, FALSE) != NRC_GOODRET)
{
    if (*NbCommonErrorMsg)
        pLB->AddString(NbCommonErrorMsg);
```



```

        sprintf(Str, "ResetAll failed with error %d:",
                                GetLastError());
    pLB->AddString(Str);
    return;
}

GetDlgItem(IDC_SERVER)->GetWindowText(szServerName,
                                NCBNAMSZ);
GetDlgItem(IDC_CLIENT)->GetWindowText(szClientName,
                                NCBNAMSZ);

```

Теперь разместим в памяти массив дескрипторов событий, а также массив структур NCB. Для каждого номера LANA создадим объект типа "событие" и пропишем его в структуре NCB. Добавим имя клиента к номеру LANA и вызовем функцию Connect, чтобы попытаться связаться с сервером.

```

hArray = (HANDLE *)GlobalAlloc(GMEM_FIXED,
                                sizeof(HANDLE) * lenum.length);
pncb    = (NCB *)GlobalAlloc(GMEM_FIXED | GMEM_ZEROINIT,
                                sizeof(NCB) * lenum.length);
for(i = 0; i < lenum.length; i++)
{
    hArray[i] = CreateEvent(NULL, TRUE, FALSE, NULL);
    pncb[i].ncb_event = hArray[i];

    AddName(lenum.lana[i], szClientName, (int *)&dwNum);
    if (*NbCommonErrorMsg)
        pLB->AddString(NbCommonErrorMsg);
    Connect(&pncb[i], lenum.lana[i], szServerName,
            szClientName);
}

```

Дождемся, пока произойдет соединение хотя бы на одном номере LANA:

```

dwIndex = WaitForMultipleObjects(lenum.length, hArray,
                                FALSE, INFINITE);

if (dwIndex == WAIT_FAILED)
{
    sprintf (Str, "ERROR: WaitForMultipleObjects: %d",
                                GetLastError());

    pLB->AddString(Str);
}
else
{
    // Если успешно хотя бы одно соединение, остальные
    // мы разрываем. Используем соединение, возвращенное
    // WaitForMultipleObjects.
    // Остальные, если они не установлены еще, отменим

```

```

for(i = 0; i < lenum.length; i++)
{
    if (i != dwIndex)
    {
        if (pncb[i].ncb_cmd_cplt == NRC_PENDING)
            Cancel(&pncb[i]);
        else
        {
            Hangup(pncb[i].ncb_lana_num,
                    pncb[i].ncb_lsn);
            if (*NbCommonErrorMsg)
                pLB->AddString(NbCommonErrorMsg);
        }
    }
}
sprintf (Str, "Connected on LANA: %d",
          pncb[dwIndex].ncb_lana_num);
pLB->AddString(Str);
SetConnected(true);
}

```

Обработчик нажатия кнопки "Send and Receive" (идентификатор IDC_SEND) посылает и получает данные. По завершении он производит окончательную очистку. Далее приводится примерный код.

Сначала опишем необходимые переменные:

```

char    szSendBuff[MAX_BUFFER];
DWORD   dwBufferLen,
        dwRet;
int      i;
CListBox * pLB = ((CListBox *)GetDlgItem(IDC_LISTBOX));

```

Освежаем значения переменных, связанных с элементами управления диалогового окна. Далее отправляем тестовые сообщения и получаем ответы. Выводим информацию об этом процессе в окне диалога. После этого разрываем соединение с сервером:

```

UpdateData();
for(i = 0; i < m_Number; i++)
{
    wsprintf(szSendBuff, "Test message %03d", i);
    dwRet = Send(pncb[dwIndex].ncb_lana_num,
                 pncb[dwIndex].ncb_lsn, szSendBuff,
                 strlen(szSendBuff));
    if (dwRet != NRC_GOODRET)
    {
        if (*NbCommonErrorMsg)
            pLB->AddString(NbCommonErrorMsg);
    }
}

```

```

        sprintf(Str,
            "Send failed with error %d, message number:",
            GetLastError(), i);
        pLB->AddString(Str);
        break;
    }

    dwBufferLen = MAX_BUFFER;
    dwRet = Recv(pncb[dwIndex].ncb_lana_num,
        pncb[dwIndex].ncb_lsn, szSendBuff, &dwBufferLen);
    if (dwRet != NRC_GOODRET)
    {
        if (*NbCommonErrorMsg)
            pLB->AddString(NbCommonErrorMsg);
        sprintf(Str,
            "Recv failed with error %d, meesage number:",
            GetLastError(), i);
        pLB->AddString(Str);
        break;
    }
    szSendBuff[dwBufferLen] = 0;
    sprintf (Str, "Read: '%s'", szSendBuff);
    pLB ->AddString(Str);
}

Hangup(pncb[dwIndex].ncb_lana_num,
    pncb[dwIndex].ncb_lsn);
if (*NbCommonErrorMsg)
    pLB->AddString(NbCommonErrorMsg);

```

Наконец, освобождаем захваченные ресурсы и изменяем состояние главного окна диалога:

```

for(i = 0; i < lenum.length; i++)
{
    DelName(lenum.lana[i], szClientName);
    if (*NbCommonErrorMsg)
        pLB->AddString(NbCommonErrorMsg);
    CloseHandle(hArray[i]);
}
GlobalFree(hArray);
GlobalFree(pncb);

SetConnected(false);

```

Пример 1.3. Эхо-сервер NetBIOS, использующий функции обратного вызова

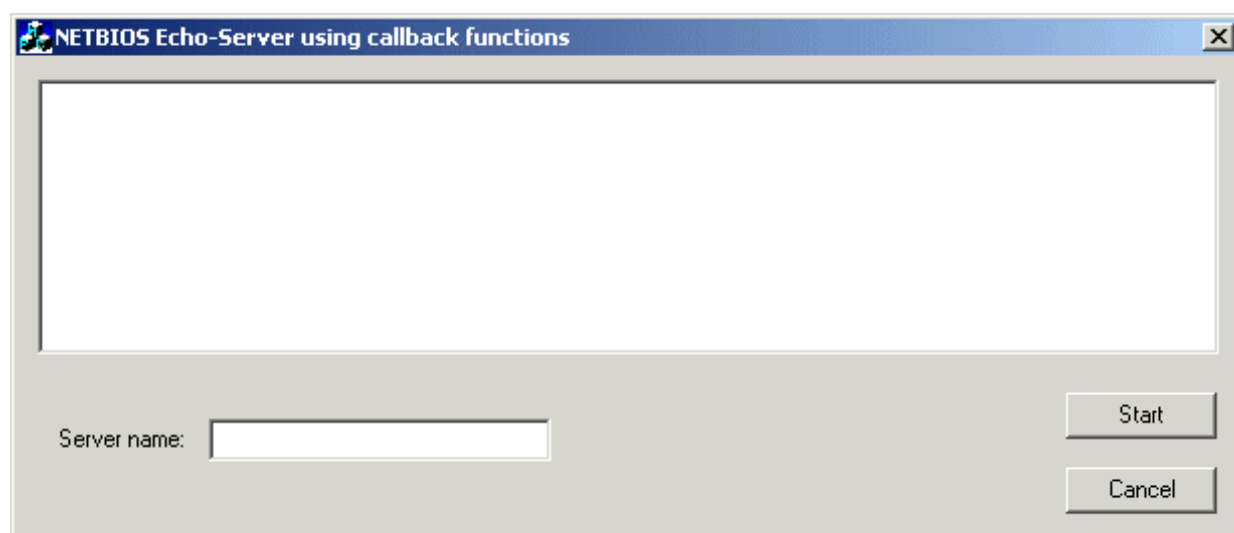
Работа приложения осуществляется по следующему сценарию. Сервер добавляет свое имя к каждому номеру LANА на данном компьютере и прослушивает каждый номер LANА в ожидании запроса клиента. Ожидание подключения клиентов производится с использованием механизма функций обратного вызова. Для сеанса связи с каждым клиентом создается отдельный поток.

Открытие проекта

Откройте файл рабочего пространства заготовки проекта Server.dsw, расположенный в директории 01_NetBIOS\ServerCb\Starter.

Главное диалоговое окно

Проект организован как диалоговое приложение с главным окном следующего вида:



Список в верхней части окна (идентификатор IDC_LISTBOX) предназначен для вывода сообщений. Поле ввода (идентификатор IDC_SERVER) предназначено для ввода имени сервера, по которому должен обращаться клиент для подключения.

Кнопка "Start" (идентификатор IDC_START) предназначена для запуска сервера.

Добавление файлов, подключение библиотек, объявление констант и глобальных переменных

Скопируем в папку проекта файлы nbcommon.h и NbCommon.lib и добавим их в проект.

В окне Project Settings на вкладке Link в поле Object/library modules: подключим библиотеку netapi32.lib.

Подключим заголовок библиотеки NbCommon в файле ServerDlg.h:

```
// ...
#include "nbcommon.h"
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// CServerDlg dialog
```

В начале файла ServerDlg.cpp добавим необходимые объявления констант, глобальных переменных и объявления глобальных функций:

```
#define MAX_BUFFER      2048
char      szServerName [NCBNAMSZ] ;

HWND      hWnd_LB; // Для использования в потоках

int Listen(int lana, char *name);
void CALLBACK ListenCallback(PNCB pncb);
UINT ClientThread(PVOID lpParam);
```

Здесь переменная hWnd_LB нужна для того, чтобы функции, не являющиеся членами класса CServerDlg, имели возможность выводить информацию в окне диалога. При этом они могут быть вызваны из других потоков. Напомним, что объекты классов, производных от CWnd (у нас это список IDC_LISTBOX), не могут быть использованы в других потоках непосредственно. Для этого требуется создать временный объект, передав функции FromHandle дескриптор окна, к которому мы хотим получить доступ. Глобальная переменная hWnd_LB и предназначена для хранения этого дескриптора.

Определение переменных и функций-членов класса CServerDlg

Добавим переменную m_IsStarted типа bool для указания, был ли сервер запущен. Для управления списком IDC_LISTBOX создадим переменную m_ListBox типа CListBox и свяжем ее со списком.

Объявим также функцию SetStarted() для изменения состояния элементов управления в зависимости от состояния сервера.

```
void SetStarted(bool IsStarted);
```

Код реализации функции SetStarted () может выглядеть примерно так:

```
void CServerDlg::SetStarted(bool IsStarted)
{
    m_IsStarted = IsStarted;

    GetDlgItem(IDC_SERVER)->EnableWindow(!IsStarted);
    GetDlgItem(IDC_START)->EnableWindow(!IsStarted);
}
```

В коде функции CServerDlg::OnInitDialog() настроим главное окно диалога перед его первым появлением:

```
// ...
// TODO: Add extra initialization here
GetDlgItem(IDC_SERVER) ->SetWindowText("TEST-SERVER");
SetStarted(false);
// ...
```

Определение обработчиков событий

Теперь необходимо написать обработчик события, происходящего при нажатии кнопки "Start" (идентификатор IDC_START).

Обработчик должен инициализировать интерфейс NetBIOS, распределить ресурсы, добавить имя сервера к каждому номеру LANA. Далее он слушает каждый номер LANA с соответствующим обратным вызовом.

Соответствующий код может выглядеть, например, так. Сначала опишем необходимые переменные и инициализируем глобальную переменную hWnd_LB так, чтобы функции, выполняющиеся в других потоках, имели возможность создать ссылку для вывода информации в окно m_ListBox:

```
LANA_ENUM    lenum;
int          i,
            num;
char        Str[200];
HWND_LB = m_ListBox.m_hWnd; // Для использования в потоках
```

Далее перечисляются и инициализируются все номера LANA. В случае, если таковых не обнаружено (вероятнее всего потому, что в системе не установлена поддержка NetBIOS), поток сразу прекращает свою работу. В случае неуспешного завершения вызова функции NetBIOS в окно выводится сформированное ею сообщение об ошибке.

```
if (LanaEnum(&lenum) != NRC_GOODRET)
{
    if (*NbCommonErrorMsg)
        m_ListBox.AddString(NbCommonErrorMsg);
    sprintf(Str, "LanaEnum failed with error %d:",
            GetLastError());
    m_ListBox.AddString(Str);
    return;
}
if (lenum.length == 0)
{
    m_ListBox.AddString("Sorry, no existing LANA...");
    GetDlgItem(IDC_SERVER) ->EnableWindow(false);
    GetDlgItem(IDC_START) ->EnableWindow(false);
    return;
}
```

```

if (ResetAll(&lenum, 254, 254, FALSE) != NRC_GOODRET)
{
    if (*NbCommonErrorMsg)
        m_ListBox.AddString(NbCommonErrorMsg);
    sprintf(Str, "ResetAll failed with error %d:",
            GetLastError());
    m_ListBox.AddString(Str);
    return;
}

```

Теперь запомним имя сервера в глобальной переменной, добавим к каждому номеру LANA имя сервера и дадим команду на прослушивание соединения:

```

GetDlgItem(IDC_SERVER)->GetWindowText(szServerName,
                                       NCBNAMSZ);

for(i = 0; i < lenum.length; i++)
{
    if (*NbCommonErrorMsg)
        m_ListBox.AddString(NbCommonErrorMsg);
    AddName(lenum.lana[i], szServerName, &num);
    Listen(lenum.lana[i], szServerName);
}

m_ListBox.AddString("Server started");
SetStarted(true);

```

Глобальная функция Listen

Функция иницирует асинхронное прослушивание сети с помощью функции обратного вызова. С этой целью создается и инициализируется для асинхронного прослушивания *глобальная* структура NCB. В ее поле ncb_post прописывается функция обратного вызова (ListenCallback), которая будет вызываться при обращении клиента за подключением.

Соответствующий исходный код может выглядеть примерно так:

```

int Listen(int lana, char *name)
{
    PNCBpncb = NULL;
    pncb = (PNCB)GlobalAlloc
            (GMEM_FIXED | GMEM_ZEROINIT, sizeof(NCB));
    pncb->ncb_command = NCBLISTEN | ASYNCH;
    pncb->ncb_lana_num = lana;
    pncb->ncb_post = ListenCallback;

    // Имя, с которым клиенты будут соединяться
    memset(pncb->ncb_name, ' ', NCBNAMSZ);
}

```

```

strncpy((char *)pncb->ncb_name, name, strlen(name));

// Имя клиента,
// '*' означает, что примем соединение от любого клиента
memset(pncb->ncb_callname, ' ', NCBNAMSZ);
pncb->ncb_callname[0] = '*';

if (Netbios(pncb) != NRC_GOODRET)
{
    char    Str[200];
    CListBox * pLB =
        (CListBox *) (CListBox::FromHandle(hWndd_LB));

    sprintf(Str, "ERROR: Netbios: NCBLISTEN: %d",
        pncb->ncb_retcode);
    pLB->AddString(Str);
    return pncb->ncb_retcode;
}
return NRC_GOODRET;
}

```

Функция асинхронного обратного вызова ListenCallback

Данная функция вызывается при положительном результате прослушивания клиента. Если нет ошибок, то она создает поток для работы с клиентом. После этого дается команда на продолжение прослушивания.

Далее приводится ее примерный код. Обратите внимание, что поскольку мы используем библиотеку MFC, то для запуска потока должна использоваться именно функция AfxBeginThread.

```

void CALLBACK ListenCallback(PNCB pncb)
{
    if (pncb->ncb_retcode != NRC_GOODRET)
    {
        char    Str[200];
        CListBox * pLB =
            (CListBox *) (CListBox::FromHandle(hWndd_LB));

        sprintf(Str, "ERROR: ListenCallback: %d",
            pncb->ncb_retcode);
        pLB->AddString(Str);
        return;
    }

    Listen(pncb->ncb_lana_num, szServerName);
    AfxBeginThread(ClientThread, (PVOID)pncb);
    return;
}

```


Главная функция потока обслуживания клиента *ClientThread*

Клиентский поток получает данные от клиентов и отправляет их назад. Используются блокирующие функции приема и отправки данных. Поток работает, пока не закрыт сеанс или пока не произойдет ошибка. Перед выходом структура NCB освобождается.

Ниже приводится соответствующий исходный код, достаточно понятный и без комментариев.

```
UINT ClientThread(PVOID lpParam)
{
    PNCB      pncb = (PNCB)lpParam;
    NCB       ncb;
    char      szRecvBuff[MAX_BUFFER];
    DWORD     dwBufferLen = MAX_BUFFER,
              dwRetVal = NRC_GOODRET;
    char      szClientName[NCBNAMSZ+1];
    char      Str[200];
    CListBox * pLB =
        (CListBox *) (CListBox::FromHandle(hWnd_LB));

    FormatNetbiosName((char *)pncb->ncb_callname,
                     szClientName);

    while (1)
    {
        dwBufferLen = MAX_BUFFER;

        dwRetVal = Recv(pncb->ncb_lana_num, pncb->ncb_lsn,
                        szRecvBuff, &dwBufferLen);
        if (dwRetVal != NRC_GOODRET)
        {
            if (*NbCommonErrorMsg)
                pLB->AddString(NbCommonErrorMsg);
            sprintf(Str, "Recv failed with error %d",
                    GetLastError());
            pLB->AddString(Str);
            break;
        }
        szRecvBuff[dwBufferLen] = 0;
        sprintf(Str, "READ [LANA=%d]: '%s'",
                pncb->ncb_lana_num, szRecvBuff);
        pLB->AddString(Str);

        dwRetVal = Send(pncb->ncb_lana_num, pncb->ncb_lsn,
                        szRecvBuff, dwBufferLen);
    }
}
```

```

    if (dwRetVal != NRC_GOODRET)
    {
        if (*NbCommonErrorMsg)
            pLB->AddString(NbCommonErrorMsg);
        sprintf(Str, "Send failed with error %d",
                GetLastError());
        pLB->AddString(Str);
        break;
    }
}
sprintf(Str, "Client '%s' on LANA %d disconnected",
        szClientName, pncb->ncb_lana_num);
pLB->AddString(Str);

if (dwRetVal != NRC_SCLOSED)
{
    // Непредусмотренная ошибка. Соединение закрывается.
    //
    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBHANGUP;
    ncb.ncb_lsn = pncb->ncb_lsn;
    ncb.ncb_lana_num = pncb->ncb_lana_num;

    if (Netbios(&ncb) != NRC_GOODRET)
    {
        sprintf(Str, "ERROR: Netbios: NCBHANGUP: %d",
                ncb.ncb_retcode);
        pLB->AddString(Str);
        dwRetVal = ncb.ncb_retcode;
    }
    GlobalFree(pncb);
    return dwRetVal;
}
GlobalFree(pncb);
return NRC_GOODRET;
}

```

Пример 1.4. Эхо-сервер NetBIOS, основанный на модели событий

Рассматриваемый пример – это еще один вариант простого эхо-сервера, однако в качестве механизма оповещения о завершении операции используются события Win32.

Последовательность действий аналогична действиям сервера обратного вызова:

1. Перечисляются номера LANA.

2. Каждый номер LANA сбрасывается.
3. Имя сервера добавляется к каждому номеру LANA.
4. На каждом номере LANA иницируется прослушивание.

При этом необходимо отслеживать все невыполненные команды прослушивания, чтобы сопоставить завершение события с соответствующими блоками NCB, инициализирующими конкретную команду.

Процесс прослушивания производится в специально созданном рабочем потоке, чтобы не блокировать поток, отвечающий за пользовательский интерфейс. Для сеанса связи с каждым клиентом также создается отдельный поток.

Открытие проекта

Откройте файл рабочего пространства заготовки проекта Server.dsw, расположенный в директории 01_NetBIOS\ServerEv\Starter.

Главное диалоговое окно

Внешний вид главного диалогового окна, элементы управления, расположенные на нем, их атрибуты и назначение ничем не отличаются от рассмотренных в предыдущем примере.

Список (идентификатор IDC_LISTBOX) предназначен для вывода сообщений. Поле ввода (идентификатор IDC_SERVER) предназначено для ввода имени сервера, по которому должен обращаться клиент для подключения.

Кнопка "Start" (идентификатор IDC_START) предназначена для запуска сервера.

Добавление файлов, подключение библиотек, объявление констант и глобальных переменных

Скопируем в папку проекта файлы nbcommon.h и NbCommon.lib и добавим их в проект.

В окне Project Settings на вкладке Link в поле Object/library modules: подключим библиотеку netapi32.lib.

Подключим заголовок библиотеки NbCommon в файле ServerDlg.h:

```
// ...
#include "nbcommon.h"
////////////////////////////////////
// CServerDlg dialog
```

В начале файла ServerDlg.cpp добавим необходимые объявления констант, глобальных переменных и объявления глобальных функций:

```
#define MAX_SESSIONS 254
#define MAX_NAMES      254
#define MAX_BUFFER      2048
```

```
char    szServerName[NCBNAMSZ];
char    Str[200];
HWND    hWnd_LB; // Для использования в потоках
```

```
NCB *g_Clients=NULL; // Глобальная структура NCB для клиентов
```

```
int      Listen(PNCB pncb, int lana, char *name);
UINT     ClientThread(PVOID lpParam);
UINT     ListenThread(PVOID lpParam);
```

Глобальная переменная hWnd_LB предназначена для хранения дескриптора окна списка IDC_LISTBOX и нужна для того, чтобы функции, выполняющиеся в других потоках, имели возможность выводить информацию в окне диалога.

Определение переменных и функций-членов класса CServerDlg

Добавим переменную m_IsStarted типа bool для указания, был ли сервер запущен. Для удобства управления списком IDC_LISTBOX создадим переменную m_ListBox типа CListBox.

Объявим также функцию SetStarted() для изменения состояния элементов управления в зависимости от состояния сервера.

```
void SetStarted(bool IsStarted);
```

Код функции SetStarted () (в файле реализации) может выглядеть примерно так:

```
void CServerDlg::SetStarted(bool IsStarted)
{
    m_IsStarted = IsStarted;

    GetDlgItem(IDC_SERVER)->EnableWindow(!IsStarted);
    GetDlgItem(IDC_START)->EnableWindow(!IsStarted);
}
```

В коде функции CServerDlg::OnInitDialog() настроим главное окно диалога перед его первым появлением:

```
// ...
// TODO: Add extra initialization here
GetDlgItem(IDC_SERVER)->SetWindowText("TEST-SERVER");
SetStarted(false);
// ...
```

Определение обработчиков событий

Создадим обработчик события, происходящего при нажатии кнопки "Start" (идентификатор IDC_START). Он должен сохранить дескриптор

окна IDC_LISTBOX для использования в других потоках, а также сохранить в глобальной переменной имя сервера. После этого он иницирует процесс прослушивания в отдельном рабочем потоке.

Исходный код может выглядеть так:

```
void CServerDlg::OnStart()
{
    hWnd_LB = m_ListBox.m_hWnd;
    GetDlgItem(IDC_SERVER) ->GetWindowText(szServerName,
                                           NCBNAMSZ);

    AfxBeginThread(ListenThread, NULL);
    SetStarted(true);
}
```

Главная функция потока прослушивания ListenThread

Функция ListenThread должна инициализировать интерфейс NetBIOS, распределить ресурсы, добавить имя сервера к каждому номеру LANA. Затем она дает асинхронную команду слушать каждому LANA, используя события, и ждет, пока событие не сработает. В ответ на срабатывание события запускается отдельный поток обслуживания клиента.

Соответствующий код может выглядеть, например, так. Сначала опишем необходимые переменные:

```
PNCB          pncb=NULL;
HANDLE         hArray[64];
DWORD          dwHandleCount=0,
               dwRet;
int            i,
               num;
LANA_ENUM      lenum;

char           Str[200];
CListBox* pLB =
               (CListBox *) (CListBox::FromHandle(hWnd_LB));
```

Здесь в переменную pLB записывается указатель на окно для вывода информации, действительный в данном потоке и полученный через дескриптор hWnd_LB. В дальнейшем мы будем использовать этот прием без дополнительных комментариев.

Далее перечисляются и инициализируются все номера LANA. В случае, если таковых не обнаружено (вероятнее всего потому, что в системе не установлена поддержка NetBIOS), поток сразу прекращает свою работу. В случае неуспешного завершения вызова функции NetBIOS в окно выводится сформированное ею сообщение об ошибке.

```

if (LanaEnum(&lenum) != NRC_GOODRET)
{
    if (*NbCommonErrorMsg)
        pLB->AddString(NbCommonErrorMsg);
    sprintf(Str, "LanaEnum failed with error %d:",
            GetLastError());
    pLB->AddString(Str);
    return 1;
}

if (lenum.length==0)
{
    pLB->AddString("Sorry, no existing LANA...");
    return 1;
}

if (ResetAll(&lenum, (UCHAR)MAX_SESSIONS,
            (UCHAR)MAX_NAMES, FALSE) != NRC_GOODRET)
{
    if (*NbCommonErrorMsg)
        pLB->AddString(NbCommonErrorMsg);
    sprintf(Str, "ResetAll failed with error %d:",
            GetLastError());
    pLB->AddString(Str);
    return 1;
}

```

Теперь разместим в памяти массив структур NCB (по одной на каждый номер LANA), свяжем его элементы с вновь созданными событиями, добавим к каждому номеру LANA имя сервера и дадим команду на прослушивание соединения:

```

g_Clients = (PNCB)GlobalAlloc(
    GMEM_FIXED | GMEM_ZEROINIT,
    sizeof(NCB) * lenum.length);

for(i = 0; i < lenum.length; i++)
{
    hArray[i] = g_Clients[i].ncb_event =
        CreateEvent(NULL, TRUE, FALSE, NULL);
    AddName(lenum.lana[i], szServerName, &num);
    if (*NbCommonErrorMsg)
        pLB->AddString(NbCommonErrorMsg);
    Listen(&g_Clients[i], lenum.lana[i], szServerName);
}
pLB->AddString("Wait for connections");

```

После этого в цикле ожидаем подключения клиентов и для каждого подключения организуем обслуживание в отдельном потоке:

```
while (1)
{
    // Ожидание подключения клиента
    dwRet = WaitForMultipleObjects(lenum.length, hArray,
        FALSE, INFINITE);
    if (dwRet == WAIT_FAILED)
    {
        sprintf(Str, "ERROR: WaitForMultipleObjects: %d",
            GetLastError());
        pLB->AddString(Str);
        break;
    }
    pLB->AddString("Client connected");

    // Проверка всех структур NCB для определения,
    // достигла ли успеха более, чем одна структура.
    // Если поле ncb_cmd_plt не содержит значение
    // NRC_PENDING, значит существует клиент, необходимо
    // создать поток и выделить ему новую структуру NCB.
    // Старая структура используется повторно.
    for(i = 0; i < lenum.length; i++)
    {
        if (g_Clients[i].ncb_cmd_cplt != NRC_PENDING)
        {
            pncb = (PNCB)GlobalAlloc(GMEM_FIXED,
                sizeof(NCB));
            memcpy(pncb, &g_Clients[i], sizeof(NCB));
            pncb->ncb_event = 0;

            AfxBeginThread(ClientThread, (PVOID)pncb);
            // Событие сбрасывается, начинаем еще
            // одно прослушивание
            ResetEvent(hArray[i]);
            Listen(&g_Clients[i], lenum.lana[i],
                szServerName);
        }
    }
}
```

В завершение функции организуем освобождение захваченных ресурсов:

```
for(i = 0; i < lenum.length; i++)
{
    DelName(lenum.lana[i], szServerName);
    if (*NbCommonErrorMsg)
        pLB->AddString(NbCommonErrorMsg);
    CloseHandle(hArray[i]);
}

GlobalFree(g_Clients);
```

Глобальная функция Listen

Основная задача функции – выдать команду асинхронного прослушивания на указанном LANA. В переданной структуре NCB полю ncb_event уже присвоено значение дескриптора события.

Исходный код функции:

```
int Listen(PNCB pncb, int lana, char *name)
{
    char    Str[200];
    CListBox* pLB =
        (CListBox *) (CListBox::FromHandle(hWnd_LB));

    pncb->ncb_command = NCBLISTEN | ASYNCH;
    pncb->ncb_lana_num = lana;

    // Имя, с которым клиенты будут соединяться
    memset(pncb->ncb_name, ' ', NCBNAMSZ);
    strncpy((char *)pncb->ncb_name, name, strlen(name));

    // Имя клиента,
    // '*' означает, что примем соединение от любого клиента
    memset(pncb->ncb_callname, ' ', NCBNAMSZ);
    pncb->ncb_callname[0] = '*';
    if (Netbios(pncb) != NRC_GOODRET)
    {
        sprintf(Str, "ERROR: Netbios: NCBLISTEN: %d",
            pncb->ncb_retcode);
        pLB->AddString(Str);
        return pncb->ncb_retcode;
    }
    return NRC_GOODRET;
}
```


Главная функция потока обслуживания клиента *ClientThread*

Клиентский поток берет структуру NCB из сеанса соединения и в цикле, пока не будет закрыт, ждет входящих данных, затем отправляет их обратно.

Перед выходом структура NCB освобождается.

Ниже приводится соответствующий исходный код.

```
UINT ClientThread(PVOID lpParam)
{
    PNCB    pncb = (PNCB)lpParam;
    NCB      ncb;
    char     szRecvBuff[MAX_BUFFER],
            szClientName[NCBNAMSZ + 1];
    DWORD    dwBufferLen = MAX_BUFFER,
            dwRetVal = NRC_GOODRET;
    char     Str[200];
    CListBox* pLB =
        (CListBox *) (CListBox::FromHandle(hWnd_LB));

    //
    // Отправка и прием сообщений, пока сеанс не закрыт
    //
    FormatNetbiosName((char *)pncb->ncb_callname,
                     szClientName);
    if (*NbCommonErrorMsg)
        pLB->AddString(NbCommonErrorMsg);
    while (1)
    {
        dwBufferLen = MAX_BUFFER;
        dwRetVal = Recv(pncb->ncb_lana_num, pncb->ncb_lsn,
                       szRecvBuff, &dwBufferLen);
        if (dwRetVal != NRC_GOODRET)
        {
            if (*NbCommonErrorMsg)
                pLB->AddString(NbCommonErrorMsg);
            sprintf(Str, "Recv failed with error %d",
                   GetLastError());
            pLB->AddString(Str);
            break;
        }

        szRecvBuff[dwBufferLen] = 0;
        sprintf(Str, "READ [LANA=%d]: '%s'",
                pncb->ncb_lana_num, szRecvBuff);
        pLB->AddString(Str);
    }
}
```

```

        dwRetVal = Send(pncb->ncb_lana_num, pncb->ncb_lsn,
            szRecvBuff, dwBufferLen);
        if (dwRetVal != NRC_GOODRET)
        {
            if (*NbCommonErrorMsg)
                pLB->AddString(NbCommonErrorMsg);
            sprintf(Str, "Send failed with error %d",
                GetLastError());
            pLB->AddString(Str);
            break;
        }
    }
    sprintf(Str, "Client '%s' on LANA %d disconnected ",
        szClientName, pncb->ncb_lana_num);
    pLB->AddString(Str);

    //
    // Обработка ошибок в ходе чтения или записи
    //
    if (dwRetVal != NRC_SCLOSED)
    {
        ZeroMemory(&ncb, sizeof(NCB));
        ncb.ncb_command = NCBHANGUP;
        ncb.ncb_lsn = pncb->ncb_lsn;
        ncb.ncb_lana_num = pncb->ncb_lana_num;

        if (Netbios(&ncb) != NRC_GOODRET)
        {
            sprintf(Str, "ERROR: Netbios: NCBHANGUP: %d ",
                ncb.ncb_retcode);
            pLB->AddString(Str);
            GlobalFree(pncb);
            dwRetVal = ncb.ncb_retcode;
        }
    }
}

//
// Удаление динамически выделенной структуры NCB
//
GlobalFree(pncb);
return NRC_GOODRET;
}

```

Пример 1.5. Приложение для отправки и приема дейтаграмм с использованием интерфейса NetBIOS

Обмен дейтаграммами – это способ связи без установления логического соединения. Отправитель просто посылает каждый пакет по соответствующему имени NetBIOS. При этом целостность данных и порядок доставки не гарантируется.

Рассматриваемый пример – это приложение для обмена дейтаграммами в сети с использованием интерфейса NetBIOS. Оно может выступать как в роли отправителя, так и в роли приемника дейтаграмм. Реализованы три возможных варианта отправки дейтаграмм:

1. Отправка на определенное (уникальное) имя.
2. Отправка дейтаграммы на групповое имя, так что ее могут получить только процессы, зарегистрировавшие это имя.
3. Широковещательная рассылка ее по всей сети.

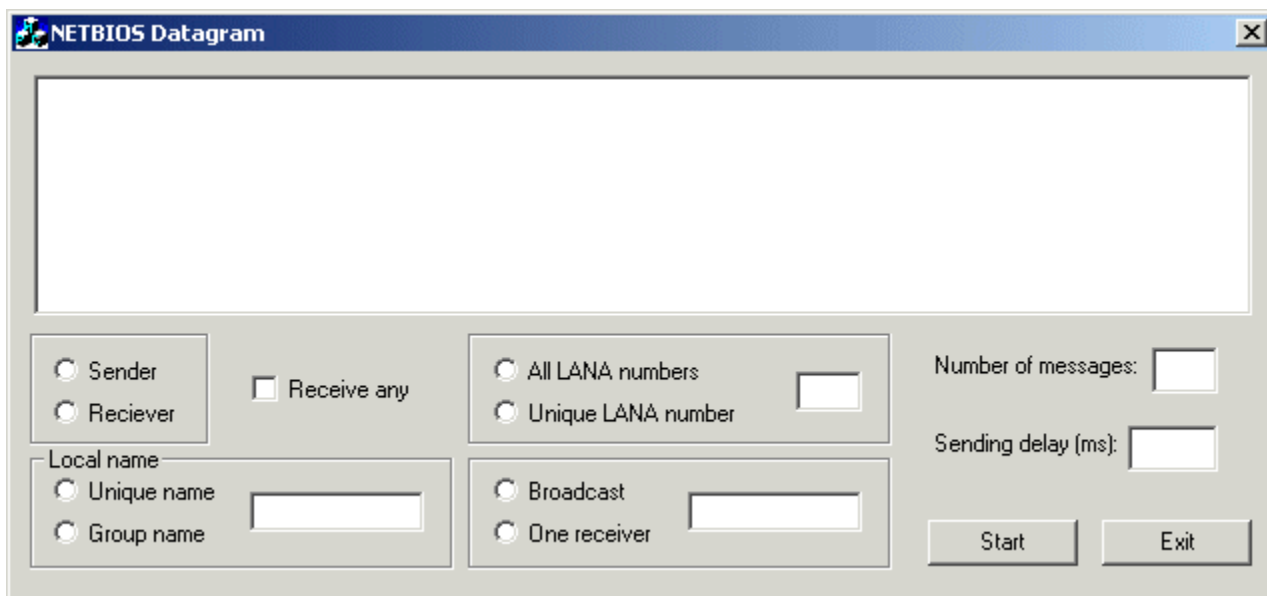
Процесс отправки и приема производится в специально созданном рабочем потоке, чтобы не блокировать поток, отвечающий за пользовательский интерфейс.

Открытие проекта

Откройте файл рабочего пространства заготовки проекта Datagram.dsw, расположенный в директории 01_NetBIOS\NbDgram\Starter.

Главное диалоговое окно

Проект организован как диалоговое приложение с главным окном следующего вида:



Ниже перечисляются основные элементы управления, расположенные на главном диалоговом окне приложения, их атрибуты и назначение.

Список в верхней части окна (идентификатор IDC_LISTBOX) предназначен для вывода сообщений. Кнопка "Start" (идентификатор IDC_START) – для запуска процесса отправки или приема дейтаграмм.

Переключатели позволяют выбрать один из следующих вариантов работы:

- отправлять или получать дейтаграммы – радиокнопки с идентификаторами IDC_SENDER и IDC_RECEIVER;
- использовать уникальное или групповое имя – радиокнопки с идентификаторами IDC_UNIQUE_NAME и IDC_GROUP_NAME;
- отправлять дейтаграммы широковещательно или на конкретное имя (уникальное или групповое) – радиокнопки с идентификаторами IDC_BROADCAST и IDC_ONE_RECEIVER;
- использовать для обмена информацией конкретный номер LANA или просматривать все.

Поля ввода доступны в зависимости от комбинации выбранных переключателей и позволяют задать:

- сетевое имя (уникальное или групповое) – поле ввода с идентификатором IDC_NAME;
- используемый номер LANA – поле ввода IDC_LANA_NUMBER;
- имя приемника дейтаграмм – поле ввода IDC_RECEIVER_NAME;
- количество отправляемых или принимаемых дейтаграмм – поле ввода IDC_NUMBER_MES;
- задержку (в мс.) между отправками – поле ввода IDC_DELAY.

Добавление файлов, подключение библиотек, объявление констант и глобальных переменных

В предложенный стартовый проект уже включены файлы nbcommon.h и NbCommon.lib, подключена библиотека netapi32.lib и заголовок библиотеки NbCommon в файле DatagramDlg.h.

В начале файла DatagramDlg.cpp добавлены необходимые объявления констант и глобальных переменных.

```
#define MAX_SESSIONS          254
#define MAX_NAMES             254
#define MAX_DATAGRAM_SIZE     512

BOOL    bSender = FALSE,      // Прием или отправка
        bRecvAny = FALSE,     // Принимать с любого имени
        bUniqueName = TRUE,   // Уникальное или групповое имя
        bBroadcast = FALSE,   // Отправлять широковещательно
        bOneLana = FALSE;     // Использовать один/все LANA

char    szLocalName[NCBNAMSZ + 1], // Свое NetBIOS имя
        szRecipientName[NCBNAMSZ + 1]; // Имя приемника
```

```

DWORD  dwNumDatagrams = 25, // Число отправляемых дейтаграмм
        dwOneLana,          // Номер LANA, если конкретный
        dwDelay = 0;        // Задержка между отправками

HWND    hDlg;
HWND    hWnd_LB;           // Дескриптор окна вывода для потоков

```

Дополнительно после этих объявлений следует поместить объявления глобальных функций для отправки и приема дейтаграмм:

```

int DatagramSend(int lana, int num, char *recipient,
                 char *buffer, int buflen);
int DatagramSendBC(int lana, int num, char *buffer,
                  int buflen);
int DatagramRecv(PNCB pncb, int lana, int num, char *buffer,
                int buflen, HANDLE hEvent);
int DatagramRecvBC(PNCB pncb, int lana, int num,
                  char *buffer, int buflen, HANDLE hEvent);

UINT MainThread(PVOID lpParam);

```

Изначальная функциональность стартового проекта

Для избавления пользователя от некоторого количества рутинной работы автор уже добавил в стартовый проект некоторую функциональность, а именно:

- В файл реализации класса `CDatagramDlg`, как только что упоминалось, добавлены некоторые константы и переменные, причем последние проинициализированы значениями по умолчанию. Также в функцию `OnInitDialog()` добавлен код для корректного отображения элементов управления в начале работы программы.
- С радиокнопками `IDC_SENDER` и `IDC_RECEIVER` связана переменная `m_Sender`, добавлены обработчики событий, связанных с изменением выбора этих кнопок.
- Добавлена функция `SetUI()` для изменения доступности тех или иных элементов управления в зависимости от состояния программы.

Определение переменных и функций-членов класса `CDatagramDlg`

Нам потребуются переменные для связи с элементами управления главного диалогового окна, их параметры указаны в следующей таблице.

Замечание: переменная `m_Sender`, как только что было отмечено, уже была добавлена при подготовке стартового проекта.

Идентификатор	Имя переменной	Тип переменной
IDC_LISTBOX	m_ListBox	CListBox
IDC_SENDER	m_Sender	int
IDC_DELAY	m_Delay	int
IDC_LANA_NUMBER	m_LanaNumber	int
IDC_NUMBER_MES	m_Mes_Number	int

Объявим также функцию RefreshValues() для оперативного изменения значений этих переменных в зависимости от состояния элементов управления.

```
void RefreshValues();
```

Код реализации функции RefreshValues() может выглядеть, например, так:

```
void CDatagramDlg::RefreshValues()
{
    UpdateData();
    GetDlgItem(IDC_NAME)->GetWindowText(szLocalName,
                                     NCBNAMSZ);
    GetDlgItem(IDC_RECEIVER_NAME)->
        GetWindowText(szRecipientName, NCBNAMSZ);
    dwNumDatagrams = m_Mes_Number;
    dwOneLana = m_LanaNumber;
    dwDelay = m_Delay;
    bSender = (((CButton *)GetDlgItem(IDC_SENDER))->
               GetCheck()==1) ? TRUE : FALSE;
    bUniqueName = (((CButton *)GetDlgItem(IDC_UNIQUE_NAME))->
                  GetCheck()==1) ? TRUE : FALSE;
    bBroadcast = (((CButton *)GetDlgItem(IDC_BROADCAST))->
                  GetCheck()==1) ? TRUE : FALSE;
    bOneLana = (((CButton *)GetDlgItem(IDC_ALL_LANA))->
                GetCheck()==1) ? FALSE : TRUE;
    bRecvAny = (((CButton *)GetDlgItem(IDC_RECV_ANY))->
                GetCheck()==1) ? TRUE : FALSE;
}
```

Как отмечалось выше, в коде функции CDatagramDlg::OnInitDialog() уже произведена настройка главного окна диалога перед его первым появлением. Данный участок кода достаточно очевиден, и с ним предлагается ознакомиться самостоятельно.

Определение обработчиков событий

В предложенном стартовом проекте большая часть пользовательского интерфейса уже реализована. В частности, написаны обработчики событий `CDatagramDlg::OnSender()` и `CDatagramDlg::OnReceiver()`, а также функция `CDatagramDlg::SetUI()`, которые осуществляют настройку интерфейса в зависимости от сделанных установок.

Нам остается создать обработчик события, происходящего при нажатии кнопки "Start" (идентификатор `IDC_START`). Он должен сохранить дескриптор окна `IDC_LISTBOX` для использования в других потоках, а также сохранить в глобальных переменных текущие значения настроек, сделанные в главном окне. После этого он инициирует процесс обмена дейтаграммами в отдельном рабочем потоке (главная функция потока `MainThread`).

Исходный код обработчика:

```
void CDatagramDlg::OnStart()
{
    // TODO: Add your control notification handler code here
    hWnd_LB = m_ListBox.m_hWnd;
    RefreshValues();

    AfxBeginThread(MainThread, NULL);
}
```

Вспомогательные глобальные функции

Главная функция потока для отправки или приема дейтаграмм использует глобальные функции `DatagramSend()`, `DatagramSendBC()`, `DatagramRecv()` и `DatagramRecvBC()`.

Функция `DatagramSend`.

Функция пересылает дейтаграмму указанному приемнику на заданном номере LANA. Параметры соответственно задают номер LANA, номер зарегистрированного локального имени, имя приемника, буфер данных и его размер:

```
int DatagramSend(int lana, int num, char *recipient,
                 char *buffer, int buflen)
{
    NCB      ncb;

    char      Str[200];
    CListBox* pLB =
        (CListBox *) (CListBox::FromHandle(hWnd_LB));

    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBDGSEND;
```

```

ncb.ncb_lana_num = lana;
ncb.ncb_num = num;
ncb.ncb_buffer = (PUCHAR)buffer;
ncb.ncb_length = buflen;

memset(ncb.ncb_callname, ' ', NCBNAMSZ);
strncpy((char *)ncb.ncb_callname, recipient,
        strlen(recipient));

if (Netbios(&ncb) != NRC_GOODRET)
{
    sprintf(Str, "Netbios: NCBDGSEND failed: %d",
            ncb.ncb_retcode);
    pLB->AddString(Str);
    return ncb.ncb_retcode;
}
return NRC_GOODRET;
}

```

Функция DatagramSendBC.

Функция посылает широковещательную дейтаграмму по конкретному номеру LANA с данного номера имени:

```

int DatagramSendBC(int lana, int num, char *buffer,
                   int buflen)
{
    NCB      ncb;

    char      Str[200];
    CListBox* pLB =
        (CListBox *) (CListBox::FromHandle(hWnd_LB));

    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBDGSENDBC;
    ncb.ncb_lana_num = lana;
    ncb.ncb_num = num;
    ncb.ncb_buffer = (PUCHAR)buffer;
    ncb.ncb_length = buflen;

    if (Netbios(&ncb) != NRC_GOODRET)
    {
        sprintf(Str, "Netbios: NCBDGSENDBC failed: %d",
                ncb.ncb_retcode);
        pLB->AddString(Str);
        return ncb.ncb_retcode;
    }
    return NRC_GOODRET;
}

```


Функция DatagramRecv.

Функция получает на заданном номере LANA дейтаграмму, направленную по имени, представленном параметром num. Принятые данные копируются в предоставленный буфер. Если hEvent не равно 0, прием осуществляется асинхронно с указанным описателем события. Если параметр num равен 0xFF, принимаются дейтаграммы, предназначенные для любого имени NetBIOS, зарегистрированного процессом.

```
int DatagramRecv(PNCB pncb, int lana, int num, char *buffer,
                 int buflen, HANDLE hEvent)
{
    char    Str[200];
    CListBox* pLB =
        (CListBox *) (CListBox::FromHandle(hWnd_LB));

    ZeroMemory(pncb, sizeof(NCB));
    if (hEvent)
    {
        pncb->ncb_command = NCBDGRECV | ASYNCH;
        pncb->ncb_event = hEvent;
    }
    else
    {
        pncb->ncb_command = NCBDGRECV;
        pncb->ncb_lana_num = lana;
        pncb->ncb_num = num;
        pncb->ncb_buffer = (PUCHAR)buffer;
        pncb->ncb_length = buflen;

        if (Netbios(pncb) != NRC_GOODRET)
        {
            sprintf(Str, "Netbos: NCBDGRECV failed: %d",
                    pncb->ncb_retcode);
            pLB->AddString(Str);
            return pncb->ncb_retcode;
        }
        return NRC_GOODRET;
    }
}
```

Функция DatagramRecvBC.

Функция получает на заданном номере LANA широковещательную дейтаграмму. Принятые данные копируются в предоставленный буфер. Если hEvent не равно 0, прием осуществляется асинхронно с указанным описателем события.

```

int DatagramRecvBC(PNCB pncb, int lana, int num,
                   char *buffer, int buflen, HANDLE hEvent)
{
    char    Str[200];
    CListBox* pLB =
        (CListBox *) (CListBox::FromHandle(hWnd_LB));

    ZeroMemory(pncb, sizeof(NCB));
    if (hEvent)
    {
        pncb->ncb_command = NCBDGRECVBC | ASYNCH;
        pncb->ncb_event = hEvent;
    }
    else
        pncb->ncb_command = NCBDGRECVBC;
    pncb->ncb_lana_num = lana;
    pncb->ncb_num = num;
    pncb->ncb_buffer = (PUCHAR)buffer;
    pncb->ncb_length = buflen;

    if (Netbios(pncb) != NRC_GOODRET)
    {
        sprintf(Str, "Netbios: NCBDGRECVBC failed: %d",
                pncb->ncb_retcode);
        pLB->AddString(Str);
        return pncb->ncb_retcode;
    }
    return NRC_GOODRET;
}

```

Главная функция основного потока MainThread

Функция MainThread инициализирует интерфейс NetBIOS, выделяет ресурсы. После этого она отправляет или получает дейтаграммы в соответствии с заданными пользователем параметрами.

Разберем основные участки исходного кода. Сначала опишем необходимые переменные:

```

LANA_ENUM    lenum;
int           i, j;
char          szMessage[MAX_DATAGRAM_SIZE],
              szSender[NCBNAMSZ + 1];
DWORD *       dwNum = NULL,
              dwBytesRead,
              dwErr;
char          Str[200];
CListBox *    pLB =
              (CListBox *) (CListBox::FromHandle(hWnd_LB));

```

Далее перечисляются и сбрасываются все номера LANA. Если таковых не обнаружено (например, потому, что в системе не установлена поддержка NetBIOS), поток сразу прекращает свою работу. В случае неуспешного завершения вызова функции NetBIOS в окно выводится сформированное ею сообщение об ошибке.

```

if ((dwErr = LanaEnum(&lenum)) != NRC_GOODRET)
{
    if (*NbCommonErrorMsg)
        pLB->AddString(NbCommonErrorMsg);
    sprintf(Str, "LanaEnum failed: %d", dwErr);
    pLB->AddString(Str);
    return 1;
}

if (lenum.length==0)
{
    pLB->AddString("Sorry, no existing LANA...");
    return 1;
}

if ((dwErr = ResetAll(&lenum, (UCHAR)MAX_SESSIONS,
    (UCHAR)MAX_NAMES, FALSE)) != NRC_GOODRET)
{
    if (*NbCommonErrorMsg)
        pLB->AddString(NbCommonErrorMsg);
    sprintf(Str, "ResetAll failed: %d", dwErr);
    pLB->AddString(Str);
    return 1;
}

```

Теперь разместим в памяти массив, элементы которого будут содержать номера NetBIOS имен, добавленных к каждому LANA:

```

dwNum = (DWORD *)GlobalAlloc(GMEM_FIXED | GMEM_ZEROINIT,
    sizeof(DWORD) * lenum.length);

if (dwNum == NULL)
{
    sprintf(Str, "Out of memory");
    pLB->AddString(Str);
    return 1;
}

```

Производим регистрацию своего имени. Если мы собираемся работать только на одном номере LANA, имя регистрируется только на этом номере, в противном случае – на всех номерах.

```

if (bOneLana)
{
    if (bUniqueName)
    {
        AddName(dwOneLana, szLocalName,
                (int *)&dwNum[0]);
        if (*NbCommonErrorMsg)
            pLB->AddString(NbCommonErrorMsg);
    }
    else
    {
        AddGroupName(dwOneLana, szLocalName,
                (int *)&dwNum[0]);
        if (*NbCommonErrorMsg)
            pLB->AddString(NbCommonErrorMsg);
    }
}
else
{
    for(i = 0; i < lenum.length; i++)
    {
        if (bUniqueName)
        {
            AddName(lenum.lana[i], szLocalName,
                    (int *)&dwNum[i]);
            if (*NbCommonErrorMsg)
                pLB->AddString(NbCommonErrorMsg);
        }
        else
        {
            AddGroupName(lenum.lana[i], szLocalName,
                    (int *)&dwNum[i]);
            if (*NbCommonErrorMsg)
                pLB->AddString(NbCommonErrorMsg);
        }
    }
}

```

Наконец, осуществляем посылку или прием дейтаграмм. Данный участок кода имеет довольно большой объем, назначение его отдельных участков поясняется комментариями.

```

if (bSender) // Посылаем дейтаграммы
{
    if (bBroadcast) // Широковещательная рассылка
    {

```

```

if (bOneLana)
{ // Широковещательная рассылка только на одном LANA
  for(j = 0; j < dwNumDatagrams; j++)
  {
    wsprintf(szMessage,
      "[%03d] Test broadcast datagram", j);
    if (DatagramSendBC(dwOneLana, dwNum[0],
      szMessage, strlen(szMessage)) != NRC_GOODRET)
      return 1;
    Sleep(dwDelay);
  }
}
else
{ // Широковещательная отправка на каждом номере LANA
  // локальной машины
  for(j = 0; j < dwNumDatagrams; j++)
  {
    for(i = 0; i < lenum.length; i++)
    {
      wsprintf(szMessage,
        "[%03d] Test broadcast datagram", j);
      if (DatagramSendBC(lenum.lana[i], dwNum[i],
        szMessage, strlen(szMessage)) != NRC_GOODRET)
        return 1;
    }
    Sleep(dwDelay);
  }
}
else // Отправка конкретному получателю
{
  if (bOneLana) // Отправка на конкретный номер LANA
  {
    for(j = 0; j < dwNumDatagrams; j++)
    {
      wsprintf(szMessage,
        "[%03d] Test directed datagram", j);
      if (DatagramSend(dwOneLana, dwNum[0],
        szRecipientName, szMessage,
        strlen(szMessage)) != NRC_GOODRET)
        return 1;
      Sleep(dwDelay);
    }
  }
}

```

```

else
{ // Отправка направленного сообщения на каждом
  // номере LANA локальной машины
  for(j = 0; j < dwNumDatagrams; j++)
  {
    for(i = 0; i < lenum.length; i++)
    {
      wsprintf(szMessage,
        "[%03d] Test directed datagram", j);
      sprintf(Str, "count: %d.%d", j, i);
      pLB->AddString(Str);
      if (DatagramSend(lenum.lana[i], dwNum[i],
        szRecipientName, szMessage,
        strlen(szMessage)) != NRC_GOODRET)
        return 1;
    }
    Sleep(dwDelay);
  }
}
}
}
else // Принимаем дейтаграммы
{
  NCB *ncb=NULL;
  char **szMessageArray = NULL;
  HANDLE *hEvent=NULL;
  DWORD dwRet;

  // Выделение массива структур NCB для передачи каждому
  // приемнику на каждом номере LANA
  ncb = (NCB *)GlobalAlloc(GMEM_FIXED | GMEM_ZEROINIT,
    sizeof(NCB) * lenum.length);

  // Выделение массива буферов входящих данных
  szMessageArray = (char **)GlobalAlloc(GMEM_FIXED,
    sizeof(char *) * lenum.length);
  for(i = 0; i < lenum.length; i++)
    szMessageArray[i] = (char *)GlobalAlloc(GMEM_FIXED,
      MAX_DATAGRAM_SIZE);

  // Выделение массива описателей событий для
  // асинхронного приема
  hEvent =
    (HANDLE *)GlobalAlloc(GMEM_FIXED | GMEM_ZEROINIT,
      sizeof(HANDLE) * lenum.length);

```

```

for(i = 0; i < lenum.length; i++)
    hEvent[i] = CreateEvent(0, TRUE, FALSE, 0);

if (bBroadcast) // Широковещание
{
    if (bOneLana)
    { // Синхронный широковещательный прием
      // на указанном номере LANA
      for(j = 0; j < dwNumDatagrams; j++)
      {
          if (DatagramRecvBC(&ncb[0], dwOneLana, dwNum[0],
                              szMessageArray[0], MAX_DATAGRAM_SIZE,
                              NULL) != NRC_GOODRET)
              return 1;
          FormatNetbiosName((char *)ncb[0].ncb_callname,
                           szSender);
          sprintf(Str, "%03d [LANA %d] Message: '%s' "
                      "received from: %s", j, ncb[0].ncb_lana_num,
                      szMessageArray[0], szSender);
          pLB->AddString(Str);
      }
    }
    else
    { // Асинхронный широковещательный прием на каждом
      // доступном номере LANA. В случае успеха выдается
      // сообщение, иначе команда отменяется
      for(j = 0; j < dwNumDatagrams; j++)
      {
          for(i = 0; i < lenum.length; i++)
          {
              dwBytesRead = MAX_DATAGRAM_SIZE;
              if (DatagramRecvBC(&ncb[i], lenum.lana[i],
                                  dwNum[i], szMessageArray[i],
                                  MAX_DATAGRAM_SIZE, hEvent[i])
                  != NRC_GOODRET)
                  return 1;
          }
          dwRet = WaitForMultipleObjects(lenum.length,
                                          hEvent, FALSE, INFINITE);
          if (dwRet == WAIT_FAILED)
          {
              sprintf(Str, "WaitForMultipleObjects failed: %d",
                      GetLastError());
              pLB->AddString(Str);
              return 1;
          }
      }
    }
}

```

```

for(i = 0; i < lenum.length; i++)
{
    if (ncb[i].ncb_cmd_cplt == NRC_PENDING)
    {
        Cancel(&ncb[i]);
        if (*NbCommonErrorMsg)
            pLB->AddString(NbCommonErrorMsg);
    }
    else
    {
        ncb[i].ncb_buffer[ncb[i].ncb_length] = 0;
        FormatNetbiosName((char *)ncb[i].ncb_callname,
            szSender);
        sprintf(Str, "%03d [LANA %d] Message: '%s' "
            "received from: %s", j, ncb[i].ncb_lana_num,
            szMessageArray[i], szSender);
        pLB->AddString(Str);
    }
    ResetEvent(hEvent[i]);
}
}
}
}
else // Широковещание отключено
{
    if (bOneLana)
    { // Блокирующий прием дейтаграмм на указанном
        // номере LANA.
        for(j = 0; j < dwNumDatagrams; j++)
        {
            if (bRecvAny)
            {
                // Прием данных, предназначенных для любого
                // имени NetBIOS в таблице имен этого процесса.
                if (DatagramRecv(&ncb[0], dwOneLana, 0xFF,
                    szMessageArray[0], MAX_DATAGRAM_SIZE,
                    NULL) != NRC_GOODRET)
                    return 1;
            }
            else
            { // Прием данных, направленных на наше
                // имя NetBIOS.
                if (DatagramRecv(&ncb[0], dwOneLana,
                    dwNum[0], szMessageArray[0],
                    MAX_DATAGRAM_SIZE, NULL)
                    != NRC_GOODRET)
                    return 1;
            }
        }
    }
}

```



```

        FormatNetbiosName((char *)ncb[0].ncb_callname,
                           szSender);
        sprintf(Str,"%03d [LANA %d] Message: '%s' "
                "received from: %s", j, ncb[0].ncb_lana_num,
                szMessageArray[0], szSender);
        pLB->AddString(Str);
    }
}
else
{
    // Асинхронный прием дейтаграмм на каждом доступном
    // номере LANA. В случае успеха выдается
    // сообщение, иначе команда отменяется
    for(j = 0; j < dwNumDatagrams; j++)
    {
        for(i = 0; i < lenum.length; i++)
        {
            if (bRecvAny)
            {
                // Прием данных, предназначенных для любого
                // имени NetBIOS в таблице имен этого процесса
                if (DatagramRecv(&ncb[i], lenum.lana[i],
                                0xFF, szMessageArray[i],
                                MAX_DATAGRAM_SIZE, hEvent[i])
                    != NRC_GOODRET)
                    return 1;
            }
            else
            {
                // Прием данных, направленных на наше
                // имя NetBIOS.
                if (DatagramRecv(&ncb[i], lenum.lana[i],
                                dwNum[i], szMessageArray[i],
                                MAX_DATAGRAM_SIZE, hEvent[i])
                    != NRC_GOODRET)
                    return 1;
            }
        }
    }
    dwRet = WaitForMultipleObjects(lenum.length,
                                   hEvent, FALSE, INFINITE);
    if (dwRet == WAIT_FAILED)
    {
        sprintf(Str,"WaitForMultipleObjects failed: %d",
                GetLastError());
        pLB->AddString(Str);
        return 1;
    }
}

```

```

        for(i = 0; i < lenum.length; i++)
        {
            if (ncb[i].ncb_cmd_cplt == NRC_PENDING)
                Cancel(&ncb[i]);
            else
            {
                ncb[i].ncb_buffer[ncb[i].ncb_length] = 0;
                FormatNetbiosName((char *)ncb[i].ncb_callname,
                                szSender);
                sprintf(Str, "%03d [LANA %d] Message: '%s' "
                        "from: %s", j, ncb[i].ncb_lana_num,
                        szMessageArray[i], szSender);
                pLB->AddString(Str);
            }
            ResetEvent(hEvent[i]);
        }
    }
}

// Очистка
for(i = 0; i < lenum.length; i++)
{
    CloseHandle(hEvent[i]);
    GlobalFree(szMessageArray[i]);
}
GlobalFree(hEvent);
GlobalFree(szMessageArray);
}

// Очистка
if (bOneLana)
{
    DelName(dwOneLana, szLocalName);
    if (*NbCommonErrorMsg)
        pLB->AddString(NbCommonErrorMsg);
}
else
{
    for(i = 0; i < lenum.length; i++)
    {
        DelName(lenum.lana[i], szLocalName);
        if (*NbCommonErrorMsg)
            pLB->AddString(NbCommonErrorMsg);
    }
}
GlobalFree(dwNum);
return 0;

```

Перенаправитель

Пример 2.1. Пример создания файла по UNC-соединению

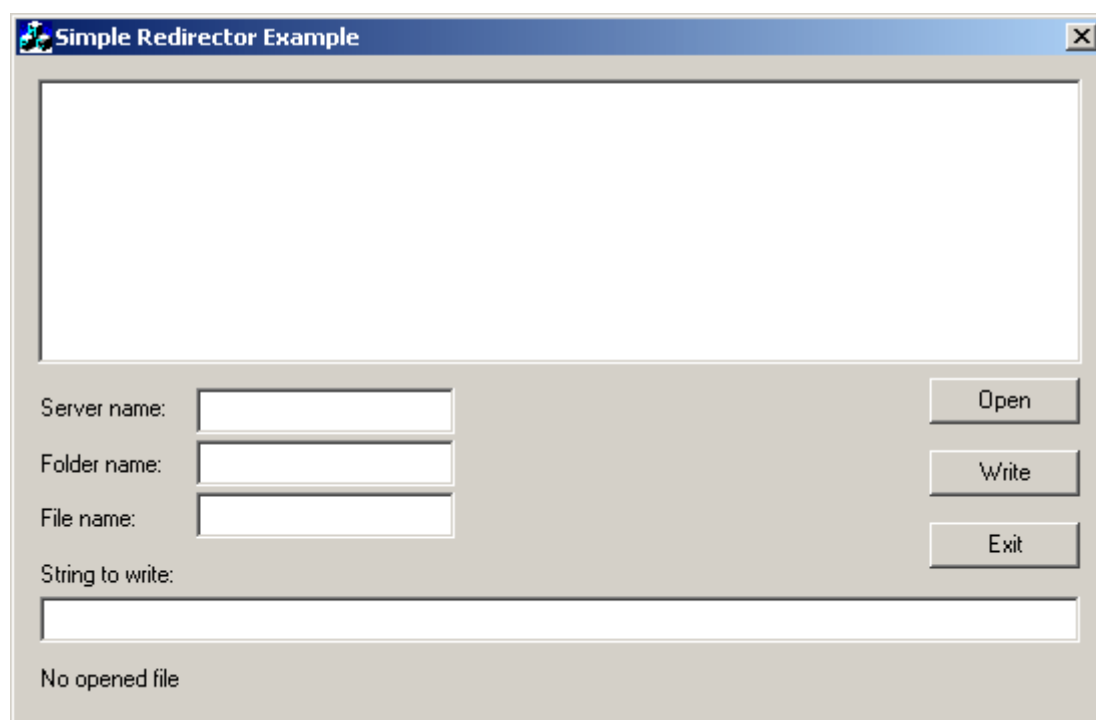
Приложения Win32 могут использовать API-функции CreateFile(), ReadFile() и WriteFile() для создания, открытия и изменения файлов по сети с использованием перенаправителя. Предлагаемый пример демонстрирует основные действия, которые должна выполнить программа для получения доступа к файлу с использованием данного механизма. В нем создается файл по UNC-соединению и туда записывается одна строка.

Открытие проекта

Откройте файл рабочего пространства проекта Redirector.dsw, расположенный в директории 02_Redirector\Starter.

Главное диалоговое окно

В стартовом проекте главный диалог приложения спроектирован следующим образом:



Ниже перечислены основные элементы управления, расположенные на главном диалоговом окне приложения, их атрибуты и назначение.

В верхней части диалогового окна расположен список (идентификатор IDC_LISTBOX), предназначенный для вывода сообщений. Поля ввода (идентификаторы IDC_SERVER, IDC_FOLDER, IDC_FILE и IDC_STRING) предназначены соответственно для ввода имени сервера, с которым устанавливается связь, папки на сервере, имени файла, который требуется создать, и строки, которую надо туда записать, соответственно.

Кнопка "Open" (идентификатор IDC_OPEN) предназначена для открытия файла, а кнопка "Write" (идентификатор IDC_WRITE) – для записи строки в файл.

Надпись (идентификатор IDC_CURRENT_FILE) будет использоваться для вывода информации о текущем состоянии подключения.

Определение переменных и функций-членов класса CRedirectorDlg

Объявим переменную m_IsOpened типа bool для указания, был ли уже открыт файл, а также переменную m_FileHandle типа HANDLE для хранения дескриптора открытого файла.

Объявим также функцию SetOpened() для изменения состояния элементов управления в зависимости от состояния подключения к серверу.

```
void SetOpened(bool IsOpened);
```

Код реализации функции SetOpened() может выглядеть примерно так:

```
void CRedirectorDlg::SetOpened(bool IsOpened)  
{  
    m_IsOpened = IsOpened;  
  
    GetDlgItem(IDC_WRITE) ->EnableWindow(IsOpened);  
    GetDlgItem(IDC_STRING) ->EnableWindow(IsOpened);  
    GetDlgItem(IDC_SERVER) ->EnableWindow(!IsOpened);  
    GetDlgItem(IDC_FOLDER) ->EnableWindow(!IsOpened);  
    GetDlgItem(IDC_FILE) ->EnableWindow(!IsOpened);  
  
    if (IsOpened)  
    {  
        char Str[200], Server[81], Folder[81], File[81];  
        GetDlgItem(IDC_SERVER) ->GetWindowText(Server, 80);  
        GetDlgItem(IDC_FOLDER) ->GetWindowText(Folder, 80);  
        GetDlgItem(IDC_FILE) ->GetWindowText(File, 80);  
        sprintf (Str, "Opened file: \\\\:s\\\:s\\\:s",  
                Server, Folder, File);  
        GetDlgItem(IDC_CURRENT_FILE) ->SetWindowText(Str);  
  
        GetDlgItem(IDC_OPEN) ->SetWindowText("Close file");  
    }  
    else  
    {  
        GetDlgItem(IDC_CURRENT_FILE) ->  
            SetWindowText("No opened file");  
        GetDlgItem(IDC_OPEN) ->SetWindowText("Open file");  
    }  
}
```

В коде функции CRedirectorDlg::OnInitDialog() настроим главное окно диалога перед его первым появлением:

```
// ...
// TODO: Add extra initialization here
GetDlgItem(IDC_SERVER)->SetWindowText("localhost");
GetDlgItem(IDC_FOLDER)->SetWindowText("MyShared");
GetDlgItem(IDC_FILE)->SetWindowText("test.txt");
GetDlgItem(IDC_STRING)->
    SetWindowText("This is a test string");
SetOpened(false);
// ...
```

Определение обработчиков событий

Теперь необходимо написать обработчики событий, получаемых главным диалоговым окном при нажатии кнопок "Open" (идентификатор IDC_OPEN) и "Write" (идентификатор IDC_WRITE).

Обработчик нажатия кнопки "Open" должен собрать строку с UNC-именем открываемого файла и открыть его посредством вызова функции CreateFile().

Пример исходного кода, выполняющего эти действия:

```
void CRedirectorDlg::OnOpen()
{
    if (m_IsOpened)
    {
        CloseHandle(m_FileHandle);
        SetOpened(false);
        return;
    }

    char Str[256], ServerName[241], Server[81],
        Folder[81], File[81];

    GetDlgItem(IDC_SERVER)->GetWindowText(Server, 80);
    GetDlgItem(IDC_FOLDER)->GetWindowText(Folder, 80);
    GetDlgItem(IDC_FILE)->GetWindowText(File, 80);
    sprintf (ServerName, "\\\\\\"%s\\"%s\\"%s",
        Server, Folder, File);

    if ((m_FileHandle = CreateFile(
        ServerName, GENERIC_WRITE | GENERIC_READ,
        FILE_SHARE_READ | FILE_SHARE_WRITE, NULL,
        CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL)) ==
        INVALID_HANDLE_VALUE)
    {
        sprintf(Str, "CreateFile failed with error %d",
            GetLastError());
    }
}
```

```

        ((CListBox *)GetDlgItem(IDC_LISTBOX)) ->
            AddString(Str);
    }
    else
    {
        sprintf(Str, "Successfully opened: %s", ServerName);
        ((CListBox *)GetDlgItem(IDC_LISTBOX)) ->
            AddString(Str);

        SetOpened(true);
    }
}

```

Обработчик нажатия кнопки "Write" записывает строку в открытый файл, используя функцию WriteFile():

```

void CRedirectorDlg::OnWrite()
{
    char Str[121];
    DWORD BytesWritten;

    GetDlgItem(IDC_STRING) -> GetWindowText(Str, 120);

    if (WriteFile(m_FileHandle, Str, strlen(Str),
        &BytesWritten, NULL) == 0)
    {
        sprintf(Str, "WriteFile failed with error %d",
            GetLastError());
        ((CListBox *)GetDlgItem(IDC_LISTBOX)) ->
            AddString(Str);
    }
    else
    {
        sprintf(Str, "Wrote %d bytes", BytesWritten);
        ((CListBox *)GetDlgItem(IDC_LISTBOX)) ->
            AddString(Str);
    }
}

```

Наконец, создадим обработчик нажатия кнопки "Cancel" и освободим там дескриптор открытого файла:

```

void CRedirectorDlg::OnCancel()
{
    // TODO: Add extra cleanup here
    if (m_IsOpened)
        CloseHandle(m_FileHandle);
    CDialog::OnCancel();
}

```

Почтовые ящики

Пример 3.1. Простой сервер почтовых ящиков

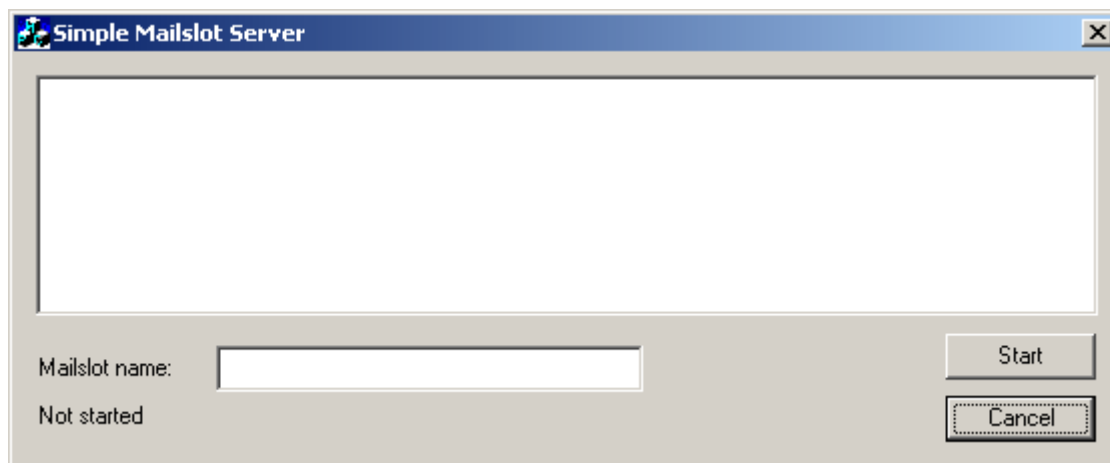
Сценарий работы нашего сервера до предела прост: он создает почтовый ящик с заданным именем и в бесконечном цикле читает поступающие туда сообщения, выводя их на экран.

Открытие проекта

Откройте файл рабочего пространства заготовки проекта Server1.dsw, расположенный в директории 03_Mailslot\Server1\Starter.

Главное диалоговое окно

Стартовый проект представляет собой диалоговое приложение с главным окном следующего вида



Ниже перечислены основные элементы управления, расположенные на главном диалоговом окне приложения, их атрибуты и назначение.

Список в верхней части окна (его идентификатор IDC_LISTBOX) предназначен для вывода сообщений. Поле ввода (идентификатор IDC_MAILSLOT) предназначено для ввода имени создаваемого почтового ящика. Надпись (идентификатор IDC_CURRENT_CONN) будет использоваться для вывода информации о текущем состоянии подключения.

Кнопка "Start" (идентификатор IDC_START) предназначена для запуска сервера.

Объявление глобальных переменных и функций

Для того чтобы не блокировать пользовательский интерфейс программы, процесс чтения информации из почтового ящика будет осуществляться в отдельном потоке.

Объявим в файле Server1Dlg.h главную функцию этого потока и структуру для передачи ей необходимых параметров:

```

struct ThreadArgs {
    HANDLE ms;
    HWND hWnd;
};

UINT ServeMailslot(LPVOID lpParameter);

////////////////////////////////////
// Server1Dlg dialog

```

Через поле `ms` мы будем передавать дескриптор почтового ящика, а через поле `hWnd` – дескриптор окна списка `IDC_LISTBOX`, для того чтобы рабочий поток имел возможность вывода информации в главном окне нашего приложения.

Определение переменных и функций-членов класса *Server1Dlg*

Добавим переменную `m_Mailslot` типа `HANDLE` для хранения дескриптора почтового ящика, а также переменную `m_pMailslotThread` – указатель на `CWinThread` – для того чтобы иметь возможность управления рабочим потоком.

```

HANDLE m_Mailslot;
CWinThread* m_pMailslotThread;

```

Для управления списком `IDC_LISTBOX` создадим переменную `m_ListBox` типа `CListBox` (через `ClassWizard`).

В коде функции `Server1Dlg::OnInitDialog()` настроим главное окно диалога перед его первым появлением:

```

// ...
// TODO: Add extra initialization here
GetDlgItem(IDC_MAILSLOT) ->SetWindowText("Myslot");
// ...

```

Определение обработчиков событий

Создадим обработчики событий, происходящих при нажатии кнопки "Start" (идентификатор `IDC_START`) и кнопки "Cancel" (идентификатор `IDCANCEL`).

Обработчик `OnStart()` создает на сервере почтовый ящик с заданным именем. Чтобы он был создан на локальном компьютере, в качестве имени сервера в UNC-строке указывается точка.

После этого заполняется *глобальная* структура `TA`, создается рабочий поток для чтения, причем его главной функции эта структура передается как параметр (через указатель). Наконец, делаем неактивными некоторые элементы управления на главном окне.

Соответствующий код может выглядеть, например, так:


```

void CServer1Dlg::OnStart()
{
    char ServerName[120], Mailslot[81];
    char Str[200];

    // Создание почтового ящика
    GetDlgItem(IDC_MAILSLOT)->GetWindowText(Mailslot, 80);
    sprintf (ServerName, "\\.\Mailslot\\%s", Mailslot);
    if ((m_Mailslot = CreateMailslot(ServerName, 0,
                                     MAILSLOT_WAIT_FOREVER, NULL)) ==
        INVALID_HANDLE_VALUE)
    {
        sprintf(Str, "Failed to create a mailslot %d",
                GetLastError());
        ((CListBox *)GetDlgItem(IDC_LISTBOX))->
            AddString(Str);
        return;
    }
    else
    {
        sprintf(Str, "Serving mailslot: %s", ServerName);
        GetDlgItem(IDC_CURRENT_CONN)->SetWindowText(Str);

        sprintf(Str, "Mailslot %s was successfully created",
                ServerName);
        ((CListBox *)GetDlgItem(IDC_LISTBOX))->
            AddString(Str);
    }

    TA.ms = m_Mailslot;
    TA.hWnd = m_ListBox.m_hWnd;

    m_pMailslotThread = AfxBeginThread(ServeMailslot, &TA);

    GetDlgItem(IDC_START)->EnableWindow(false);
    GetDlgItem(IDC_MAILSLOT)->EnableWindow(false);
}

```

Обработчик OnCancel() должен прервать выполнение потока чтения и закрыть почтовый ящик:

```

void CServer1Dlg::OnCancel()
{
    ::TerminateThread(m_pMailslotThread, 0);
    CloseHandle(m_Mailslot);
    CDialog::OnCancel();
}

```

Главная функция потока ServeMailslot

Для передачи ей параметров опишем в начале файла Server1Dlg.cpp глобальную структуру

```
struct ThreadArgs TA;
```

Функция в бесконечном цикле читает информацию из почтового ящика и выводит ее в главном окне приложения.

Соответствующий исходный код может выглядеть примерно так:

```
UINT ServeMailslot(LPVOID lpParameter)  
{  
    struct ThreadArgs *pTA =  
        (struct ThreadArgs *)lpParameter;  
    char buffer[256];  
    DWORD NumberOfBytesRead;  
    CListBox * pLB =  
        (CListBox *) (CListBox::FromHandle(pTA->hWnd));  
  
    pLB -> AddString ("Reading thread started");  
  
    // Бесконечное чтение данных  
    while(ReadFile(pTA->ms, buffer, 256, &NumberOfBytesRead,  
        NULL) != 0)  
    {  
        buffer[NumberOfBytesRead] = 0;  
        pLB->AddString(buffer);  
    }  
  
    pLB->AddString("Reading thread finished");  
  
    return 0;  
}
```

Замечания об использовании почтовых ящиков в Windows 95 и Windows 98

Почтовые ящики в этих версиях Windows, созданные с параметром MAILSLT_WAIT_FOREVER, при попытке чтения блокируются вплоть до получения данных, поэтому вариант закрытия потока, предложенный в данном примере, не сработает.

В этом случае единственный способ снять приложение – перезагрузка системы. Чтобы решить эту проблему, можно заставить сервер открыть описатель почтового ящика в отдельном потоке и отправить данные, чтобы прервать блокирующий запрос чтения. Такой вариант приложения читателям предлагается разработать самостоятельно. В папке с упражнениями имеется консольный вариант такого сервера.

Пример 3.2. Простой клиент почтовых ящиков

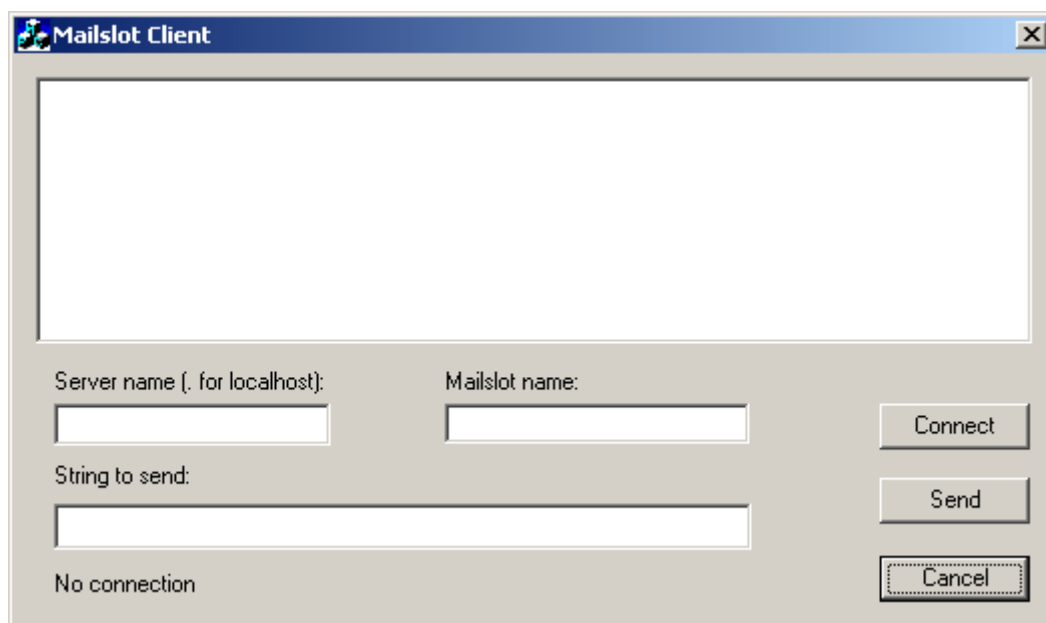
Данный клиент просто открывает на указанном сервере требуемый почтовый ящик и записывает туда заданную пользователем строку.

Открытие проекта

Откройте файл рабочего пространства заготовки проекта Client.dsw, расположенный в директории 03_Mailslot\Client\Starter.

Главное диалоговое окно

Стартовый проект, как обычно, выполнен в виде диалогового окна следующего вида



Список (идентификатор IDC_LISTBOX) предназначен для вывода сообщений. Поля ввода (идентификаторы IDC_SERVER, IDC_MAILSLLOT и IDC_STRING) предназначены для ввода имени сервера, с которым устанавливается связь, имени почтового ящика и строки сообщения, которое требуется отправить.

Кнопка "Connect" (идентификатор IDC_CONNECT) предназначена для открытия почтового ящика, а кнопка "Send" (идентификатор IDC_SEND) – для отправки заданной строки.

Надпись (идентификатор IDC_CURRENT_CONN) используется для вывода информации о текущем состоянии подключения.

Определение переменных и функций-членов класса CClientDlg

Добавим переменную m_IsConnected типа bool для указания, было ли уже произведено подключение к серверу, а также переменную m_Mailslot типа HANDLE для хранения дескриптора почтового ящика.

```
bool m_IsConnected;
HANDLE m_Mailslot;
```

Объявим также функцию SetConnected() для изменения состояния элементов управления в зависимости от наличия подключения.

```
void SetConnected(bool IsConnected);
```

Примерный код реализации функции SetConnected():

```
void CClientDlg::SetConnected(bool IsConnected)
{
    m_IsConnected = IsConnected;

    GetDlgItem(IDC_SEND) ->EnableWindow(IsConnected);
    GetDlgItem(IDC_STRING) ->EnableWindow(IsConnected);
    GetDlgItem(IDC_SERVER) ->EnableWindow(!IsConnected);
    GetDlgItem(IDC_MAIL SLOT) ->EnableWindow(!IsConnected);

    if (IsConnected)
    {
        char Str[180], Server[81], Mailslot[81];
        GetDlgItem(IDC_SERVER) ->GetWindowText(Server, 80);
        GetDlgItem(IDC_MAIL SLOT) ->
            GetWindowText(Mailslot, 80);
        sprintf (Str, "\\\\\\" %s "\\Mailslot\\" %s",
                Server, Mailslot);
        GetDlgItem(IDC_CURRENT_CONN) ->SetWindowText(Str);
        GetDlgItem(IDC_CONNECT) ->SetWindowText("Reconnect");
    }
    else
    {
        GetDlgItem(IDC_CURRENT_CONN) ->
            SetWindowText("No connection");
        GetDlgItem(IDC_CONNECT) ->SetWindowText("Connect");
    }
}
```

В коде функции CClientDlg::OnInitDialog() настроим главное окно диалога перед его первым появлением:

```
// ...
// TODO: Add extra initialization here
GetDlgItem(IDC_SERVER) ->SetWindowText(".");
GetDlgItem(IDC_MAIL SLOT) ->SetWindowText("Myslot");
GetDlgItem(IDC_STRING) ->
    SetWindowText("This is a test string");
SetConnected(false);
// ...
```

Определение обработчиков событий

Теперь необходимо написать обработчики событий, происходящих при нажатии кнопок "Connect" (идентификатор IDC_CONNECT), "Send" (идентификатор IDC_SEND), а также кнопки "Cancel" (идентификатор IDCANCEL).

Обработчик нажатия кнопки "Connect" должен открыть указанный почтовый ящик на указанном сервере. Если соединение уже создано, он должен разорвать текущее соединение, чтобы дать пользователю возможность открыть другой почтовый ящик.

Соответствующий код может выглядеть, например, так.

```
void CClientDlg::OnConnect()
{
    if (m_IsConnected)
    {
        CloseHandle(m_Mailslot);
        SetConnected(false);
        return;
    }

    char ServerName[180], Server[81], Mailslot[81];
    char Str[200];

    GetDlgItem(IDC_SERVER)->GetWindowText(Server, 80);
    GetDlgItem(IDC_MAILSLLOT)->GetWindowText(Mailslot, 80);
    sprintf (ServerName, "\\\\"%s\\"Mailslot\\"%s",
            Server, Mailslot);

    if ((m_Mailslot = CreateFile(ServerName, GENERIC_WRITE,
        FILE_SHARE_READ, NULL, OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL, NULL)) ==
        INVALID_HANDLE_VALUE)
    {
        sprintf(Str, "CreateFile failed with error %d",
            GetLastError());
        ((CListBox *)GetDlgItem(IDC_LISTBOX))->
            AddString(Str);
    }
    else
    {
        sprintf(Str, "Successfully opened %s", ServerName);
        ((CListBox *)GetDlgItem(IDC_LISTBOX))->
            AddString(Str);

        SetConnected(true);
    }
}
```

Обработчик нажатия кнопки "Send" просто записывает указанную пользователем строку в почтовый ящик. Для доступа к ящику используется его дескриптор, хранящийся в переменной m_Mailslot.

```
void CClientDlg::OnSend()
{
    char Str[121];
    DWORD BytesWritten;

    GetDlgItem(IDC_STRING) -> GetWindowText(Str, 120);

    if (WriteFile(m_Mailslot, Str, strlen(Str),
        &BytesWritten, NULL) == 0)
    {
        sprintf(Str, "WriteFile failed with error %d",
            GetLastError());
        ((CListBox *)GetDlgItem(IDC_LISTBOX)) ->
            AddString(Str);
    }
    else
    {
        sprintf(Str, "Wrote %d bytes", BytesWritten);
        ((CListBox *)GetDlgItem(IDC_LISTBOX)) ->
            AddString(Str);
    }
}
```

Обработчик OnCancel() должен просто закрыть почтовый ящик:

```
void CClientDlg::OnCancel()
{
    CloseHandle(m_Mailslot);
    CDialog::OnCancel();
}
```

Именованные каналы

В данной теме рассматриваются примеры использования именованных каналов для организации передачи данных между приложениями. Необходимые сведения об этом процессе изложены в соответствующей главе учебного пособия [7]. Всего рассматривается три примера:

- многопоточный эхо-сервер именованных каналов;
- клиентское приложение для взаимодействия этим эхо-сервером;
- эхо-сервер именованных каналов, работающий в режиме перекрытого ввода-вывода.

Пример 4.1. Пример многопоточного эхо-сервера именованных каналов

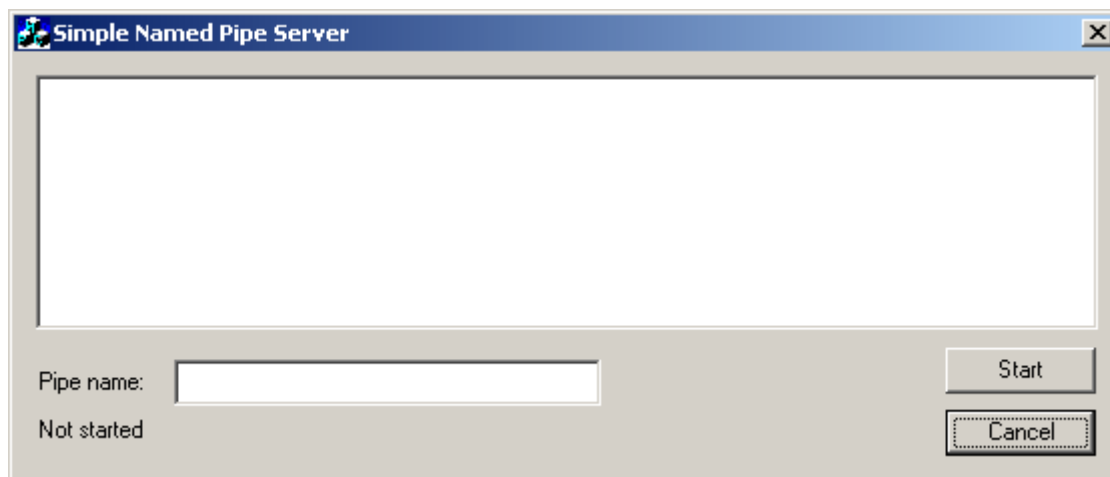
Это один из возможных вариантов сервера именованного канала, способного обслуживать несколько клиентов. Он функционирует по следующему сценарию. Сервер запускает определенное (фиксированное) количество потоков по обслуживанию клиентов. В потоке при создании канала посредством вызова функции `CreateNamedPipe()` ей передается параметр (`nMaxInstances`), указывающий, какое количество экземпляров канала сервер может поддерживать одновременно. После этого сервер в цикле читает из канала сообщения и отправляет их обратно.

Открытие проекта

Откройте файл рабочего пространства заготовки проекта `Server1.dsw`, расположенный в директории `04_Pipes\Server1\Starter`.

Главное диалоговое окно

Проект представляет собой диалоговое приложение с главным окном следующего вида



Ниже перечислены основные элементы управления, расположенные на главном диалоговом окне приложения, их атрибуты и назначение.

Список (идентификатор IDC_LISTBOX) предназначен для вывода сообщений. Поле ввода (идентификатор IDC_PIPE) предназначено для ввода имени создаваемого канала. Надпись (IDC_CURRENT_CONN) используется для вывода информации о текущем состоянии подключения.

Кнопка "Start" (идентификатор IDC_START) предназначена для запуска сервера.

Объявление констант, глобальных переменных и функций

Объявим в файле Server1Dlg.h константы, задающие количество потоков для обслуживания клиентов и размер буфера приема, а также главную функцию создаваемых потоков и структуру для передачи ей необходимых параметров:

```
#define NUM_PIPES 5
#define BUFFER_SIZE 256

struct ThreadArgs {
    HWND hWnd;
    int Num;
    char *pName;
};

UINT ServePipe(LPVOID lpParameter);
////////////////////////////////////
// Server1Dlg dialog
```

Через поле Num мы будем передавать номер потока, через поле pName – строку с именем канала, а через поле hWnd дескриптор окна списка IDC_LISTBOX, чтобы рабочий поток имел возможность вывода информации в главном окне нашего приложения.

Определение переменных и функций-членов класса Server1Dlg

Добавим переменную m_PipeName типа HANDLE для хранения UNC-строки с полным именем канала.

```
char m_PipeName[128];
```

Для управления списком IDC_LISTBOX создадим переменную m_ListBox типа CListBox (через ClassWizard).

В коде функции Server1Dlg::OnInitDialog() настроим главное окно диалога перед его первым появлением:

```
// ...
// TODO: Add extra initialization here
GetDlgItem(IDC_PIPE) ->SetWindowText("Мypipe");
// ...
```

Определение обработчиков событий

Создадим обработчик события, происходящего при нажатии кнопки "Start" (идентификатор IDC_START).

Обработчик OnStart() формирует UNC-строку с полным именем создаваемого канала. Чтобы он был создан на локальном компьютере, в качестве имени сервера в строке указывается точка.

После этого запускается требуемое количество рабочих потоков по обслуживанию клиентов (количество задается константой NUM_PIPES). Для каждого потока заполняется соответствующий элемент массива *глобальных* структур TA (описывается далее), который передается его главной функции как параметр. Наконец, мы делаем неактивными некоторые элементы управления на главном окне.

Соответствующий код может выглядеть, например, так:

```
void CServer1Dlg::OnStart()
{
    char Pipe[81];
    char Str[200];
    int i;

    GetDlgItem(IDC_PIPE) ->GetWindowText(Pipe, 80);
    sprintf (m_PipeName, "\\.\Pipe\\%s", Pipe);

    sprintf(Str, "Serving Pipe: %s", m_PipeName);
    GetDlgItem(IDC_CURRENT_CONN) ->SetWindowText(Str);

    // Создание потока для каждого экземпляра канала
    for(i = 0; i < NUM_PIPES; i++)
    {
        TA[i].hWnd = m_ListBox.m_hWnd;
        TA[i].pName = m_PipeName;
        TA[i].Num = i;
        AfxBeginThread(ServePipe, TA+i);
    }

    GetDlgItem(IDC_START) ->EnableWindow(false);
    GetDlgItem(IDC_PIPE) ->EnableWindow(false);
}
```

Главная функция потока *ServePipe*

Для передачи ей параметров опишем в начале файла *Server1Dlg.cpp* глобальный массив структур:

```
struct ThreadArgs TA[NUM_PIPES];
```

Функция обслуживает только один канал. Сначала она получает дескриптор канала посредством вызова функции *CreateNamedPipe*. После этого она в бесконечном цикле ждет подключения клиента, читает информацию из канала и отправляет ее обратно.

Соответствующий исходный код может выглядеть так:

```
UINT ServePipe(LPVOID lpParameter)  
{  
    struct ThreadArgs *pTA =  
        (struct ThreadArgs *)lpParameter;  
    char Str[256];  
    CListBox * pLB =  
        (CListBox *) (CListBox::FromHandle(pTA->hWnd));  
  
    HANDLE PipeHandle;  
    DWORD BytesRead;  
    DWORD BytesWritten;  
    CHAR Buffer[256];  
  
    sprintf(Str, "Thread [%d] started", pTA->Num);  
    pLB -> AddString (Str);  
  
    if ((PipeHandle = CreateNamedPipe(pTA->pName,  
        PIPE_ACCESS_DUPLEX,  
        PIPE_TYPE_BYTE | PIPE_READMODE_BYTE,  
        NUM_PIPES, 0, 0, 1000, NULL)) ==  
        INVALID_HANDLE_VALUE)  
    {  
        sprintf(Str, "CreateNamedPipe failed with error %d",  
            GetLastError());  
        pLB -> AddString(Str);  
        return 0;  
    }  
  
    // Обслуживание соединений клиентов в бесконечном цикле  
    while(1)  
    {  
        if (ConnectNamedPipe(PipeHandle, NULL) == 0)  
        {  
            sprintf(Str, "ConnectNamedPipe failed with error %d",  
                GetLastError());  
            pLB -> AddString(Str);  
        }  
    }
```

```

        break;
    }

    // Чтение данных и отправка их обратно клиенту,
    // пока он не закроет соединение
    while(ReadFile(PipeHandle, Buffer, sizeof(Buffer),
        &BytesRead, NULL) > 0)
    {
        sprintf(Str, "Echo %d bytes to client",
            BytesRead);
        pLB -> AddString(Str);

        if (WriteFile(PipeHandle, Buffer, BytesRead,
            &BytesWritten, NULL) == 0)
        {
            sprintf(Str, "WriteFile failed with error %d",
                GetLastError());
            pLB -> AddString(Str);
            break;
        }
    }

    if (DisconnectNamedPipe(PipeHandle) == 0)
    {
        sprintf(Str,
            "DisconnectNamedPipe failed with error %d",
            GetLastError());
        pLB -> AddString(Str);
        break;
    }
}

CloseHandle(PipeHandle);
return 0;
}

```

Пример 4.2. Простой клиент именованных каналов

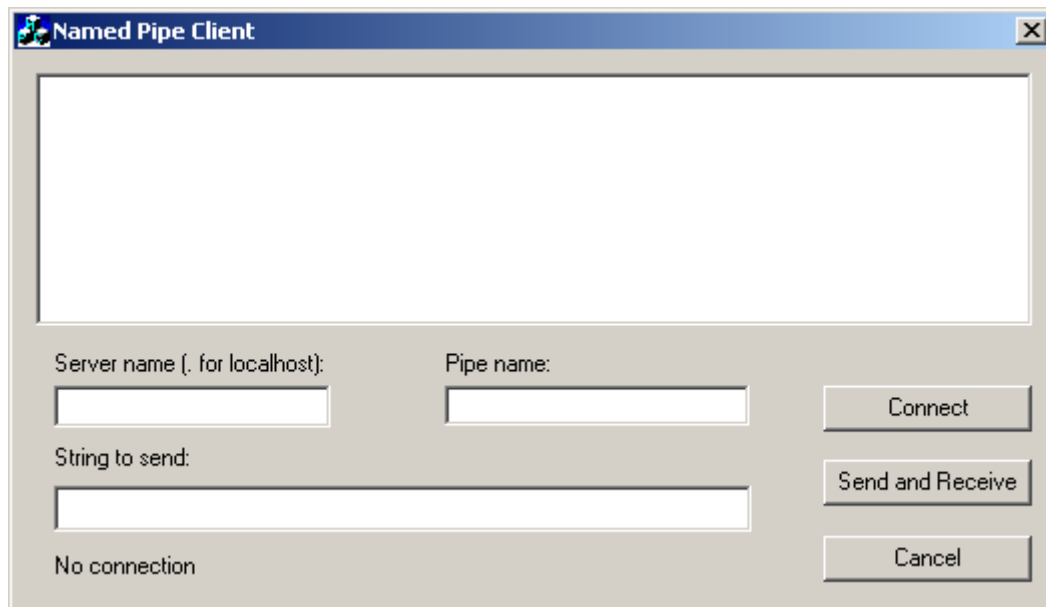
Данный клиент открывает на указанном сервере именованный канал, посылает туда заданную пользователем строку, читает из канала ответ и выводит его в диалоговом окне. Все действия по пересылке информации для простоты здесь реализованы в главном потоке программы. Поэтому при отсутствии ответа от сервера программа будет заблокирована. Студентам предлагается в качестве упражнения усовершенствовать ее так, чтобы процесс отправки и приема данных не блокировал исполнение потока пользовательского интерфейса.

Открытие проекта

Откройте файл рабочего пространства заготовки проекта Client.dsw, расположенный в директории 04_Pipes\Client\Starter.

Главное диалоговое окно

Главное окно стартового проекта имеет следующий внешний вид



Перечислим основные элементы управления, расположенные на главном диалоговом окне приложения, и их назначение.

Список (идентификатор IDC_LISTBOX) предназначен для вывода сообщений. Поля ввода (идентификаторы IDC_SERVER, IDC_PIPE и IDC_STRING) предназначены для ввода имени сервера, с которым устанавливается связь, имени канала и строки сообщения, которое требуется отправить.

Кнопка "Connect" (идентификатор IDC_CONNECT) предназначена для открытия канала, а кнопка "Send and Receive" (идентификатор IDC_SEND) – для отправки заданной строки и получения ответа.

Надпись (идентификатор IDC_CURRENT_CONN) используется для вывода информации о текущем состоянии подключения.

Определение переменных и функций-членов класса CClientDlg

Добавим переменную m_IsConnected типа bool для указания, было ли уже произведено подключение к серверу, а также переменную m_Pipe типа HANDLE для хранения дескриптора открытого канала.

```
bool m_IsConnected;  
HANDLE m_Pipe;
```

Объявим также функцию SetConnected() для изменения состояния элементов управления в зависимости от наличия подключения.

```
void SetConnected(bool IsConnected);
```

Код реализации функции SetConnected() может выглядеть, например, так:

```
void CClientDlg::SetConnected(bool IsConnected)  
{  
    m_IsConnected = IsConnected;  
  
    GetDlgItem(IDC_SEND) ->EnableWindow(IsConnected);  
    GetDlgItem(IDC_STRING) ->EnableWindow(IsConnected);  
    GetDlgItem(IDC_SERVER) ->EnableWindow(!IsConnected);  
    GetDlgItem(IDC_PIPE) ->EnableWindow(!IsConnected);  
  
    if (IsConnected)  
    {  
        char Str[180], Server[81], Pipe[81];  
        GetDlgItem(IDC_SERVER) ->GetWindowText(Server, 80);  
        GetDlgItem(IDC_PIPE) ->GetWindowText(Pipe, 80);  
        sprintf (Str, "\\\\\\"%s\\"Pipe\\"%s", Server, Pipe);  
        GetDlgItem(IDC_CURRENT_CONN) ->SetWindowText(Str);  
  
        GetDlgItem(IDC_CONNECT) ->  
            SetWindowText("Disconnect");  
    }  
    else  
    {  
        GetDlgItem(IDC_CURRENT_CONN) ->  
            SetWindowText("No connection");  
        GetDlgItem(IDC_CONNECT) ->SetWindowText("Connect");  
    }  
}
```

В коде функции CClientDlg::OnInitDialog() настроим главное окно диалога перед его первым появлением:

```
// ...  
    // TODO: Add extra initialization here  
    GetDlgItem(IDC_SERVER) ->SetWindowText(".");  
    GetDlgItem(IDC_PIPE) ->SetWindowText("Mypipe");  
    GetDlgItem(IDC_STRING) ->  
        SetWindowText("This is a test string");  
    SetConnected(false);  
// ...
```

Определение обработчиков событий

Теперь необходимо написать обработчики событий, происходящих при нажатии кнопок "Connect" (идентификатор IDC_CONNECT), "Send and Receive" (идентификатор IDC_SEND), а также кнопки "Cancel" (идентификатор IDCANCEL).

Обработчик нажатия кнопки "Connect" должен подключить клиента к указанному каналу на указанном сервере. Если соединение уже создано, он должен разорвать текущее соединение, чтобы дать пользователю возможность подключиться к другому каналу.

Код его реализации может выглядеть, например, так:

```
void CClientDlg::OnConnect()
{
    if (m_IsConnected)
    {
        CloseHandle(m_Pipe);
        SetConnected(false);
        return;
    }

    char ServerName[180], Server[81], Pipe[81];
    char Str[200];

    GetDlgItem(IDC_SERVER)->GetWindowText(Server, 80);
    GetDlgItem(IDC_PIPE)->GetWindowText(Pipe, 80);
    sprintf (ServerName, "\\\\%s\\Pipe\\%s", Server, Pipe);

    if ((m_Pipe = CreateFile(ServerName,
        GENERIC_WRITE|GENERIC_READ, FILE_SHARE_READ,
        NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
        NULL)) == INVALID_HANDLE_VALUE)
    {
        sprintf(Str, "CreateFile failed with error %d",
            GetLastError());
        ((CListBox *)GetDlgItem(IDC_LISTBOX))->
            AddString(Str);
    }
    else
    {
        sprintf(Str, "Successfully opened %s", ServerName);
        ((CListBox *)GetDlgItem(IDC_LISTBOX))->
            AddString(Str);
        SetConnected(true);
    }
}
```

Обработчик нажатия кнопки "Send and Receive" записывает указанную пользователем строку в именованный канал, дожидается ответа и выводит его в окно вывода. Для доступа к каналу используется его дескриптор, хранящийся в переменной m_Pipe.

```
void CClientDlg::OnSend()
{
    char Str[121], Buffer[121];
    DWORD BytesWritten, BytesRead;

    GetDlgItem(IDC_STRING) -> GetWindowText(Str, 120);

    if (WriteFile(m_Pipe, Str, strlen(Str), &BytesWritten,
        NULL) == 0)
    {
        sprintf(Str, "WriteFile failed with error %d",
            GetLastError());
        ((CListBox *)GetDlgItem(IDC_LISTBOX)) ->
            AddString(Str);
    }
    else
    {
        sprintf(Str, "Wrote %d bytes", BytesWritten);
        ((CListBox *)GetDlgItem(IDC_LISTBOX)) ->
            AddString(Str);
    }

    if (ReadFile(m_Pipe, Buffer, sizeof(Buffer), &BytesRead,
        NULL) != 0)
    {
        Buffer[BytesRead] = 0;
        sprintf(Str, "Recieved %d bytes: %s",
            BytesRead, Buffer);
        ((CListBox *)GetDlgItem(IDC_LISTBOX)) ->
            AddString(Str);
    }
    else
    {
        sprintf(Str, "ReadFile failed with error %d",
            GetLastError());
        ((CListBox *)GetDlgItem(IDC_LISTBOX)) ->
            AddString(Str);
    }
}
```

Обработчик OnCancel() должен просто закрыть канал:

```
void CClientDlg::OnCancel()  
{  
    CloseHandle(m_Pipe);  
    CDialog::OnCancel();  
}
```

Пример 4.3. Пример эхо-сервера именованных каналов, работающего в режиме перекрытого ввода-вывода

Это еще один подход к организации сервера именованного канала, способного обслуживать несколько клиентов.

Перекрытый ввод-вывод – это механизм асинхронного выполнения функций чтения и записи (ReadFile и WriteFile). Функциям необходимо передать структуру OVERLAPPED, которая затем будет использована для получения результатов запроса ввода-вывода с помощью API-функции GetOverlappedResult.

Сервер, так же, как и предыдущий, рассчитан на определенное количество клиентов (для простоты задается константой).

Открытие проекта

Откройте файл рабочего пространства заготовки проекта Server2.dsw, расположенный в директории 04_Pipes\Server2\Starter.

Главное диалоговое окно

Внешний вид главного диалогового окна, элементы управления, расположенные на нем, их атрибуты и назначение ничем не отличаются от рассмотренного ранее (пример 4.1) варианта.

Объявление констант, глобальных переменных и функций

Объявим в начале файла Server2Dlg.h константы, которые задают количество потоков для обслуживания клиентов и размер буфера приема, а также главную функцию создаваемых потоков и структуру для передачи ей необходимых параметров:

```
#define NUM_PIPES 5  
#define BUFFER_SIZE 256  
  
struct ThreadArgs {  
    HWND hWnd;  
    char *pName;  
};  
  
UINT ServePipes(LPVOID lpParameter);
```


Через `pName` структуры `ThreadArgs` мы будем передавать строку с именем канала, а через поле `hWnd` – дескриптор окна списка `IDC_LISTBOX`, чтобы рабочий поток имел возможность вывода информации в главное окно нашего приложения.

Определение переменных и функций-членов класса `Server2Dlg`

Добавим переменную `m_PipeName` типа `HANDLE` для хранения UNC-строки с полным именем канала.

```
char m_PipeName[128];
```

Для управления списком `IDC_LISTBOX` создадим переменную `m_ListBox` типа `CListBox` (через `ClassWizard`).

В коде функции `Server2Dlg::OnInitDialog()` настроим главное окно диалога перед его первым появлением:

```
// ...
// TODO: Add extra initialization here
GetDlgItem(IDC_PIPE) ->SetWindowText("Мypipe");
// ...
```

Определение обработчиков событий

Создадим обработчик события, происходящего при нажатии кнопки "Start" (идентификатор `IDC_START`).

Обработчик `OnStart()` формирует UNC-строку с полным именем создаваемого канала. Чтобы он был создан на локальном компьютере, в качестве имени сервера в строке указывается точка.

После этого запускается рабочий поток по обслуживанию клиентов. Перед запуском заполняется *глобальная* структура `TA` (описана ниже), которая передается главной функции потока как параметр. Перед возвратом управления мы делаем неактивными некоторые элементы управления на главном окне.

Соответствующий исходный код:

```
void CServer2Dlg::OnStart()
{
    char Pipe[81];
    char Str[200];

    // Create the Pipe
    GetDlgItem(IDC_PIPE) ->GetWindowText(Pipe, 80);
    sprintf(m_PipeName, "\\.\Pipe\\%s", Pipe);

    sprintf(Str, "Serving Pipe: %s", m_PipeName);
    GetDlgItem(IDC_CURRENT_CONN) ->SetWindowText(Str);
}
```

```

TA.hWnd = m_ListBox.m_hWnd;
TA.pName = m_PipeName;

AfxBeginThread(ServePipes, &TA);

GetDlgItem(IDC_START) ->EnableWindow(false);
GetDlgItem(IDC_PIPE) ->EnableWindow(false);
GetDlgItem(IDC_START) ->EnableWindow(false);
GetDlgItem(IDC_PIPE) ->EnableWindow(false);
}

```

Главная функция потока ServePipes

Для передачи ей параметров опишем в начале файла Server2Dlg.cpp глобальную переменную:

```
struct ThreadArgs TA;
```

Функция обслуживает все каналы, используя механизм перекрытого ввода-вывода. Разберем основные участки кода.

Сначала опишем необходимые переменные:

```

UINT ServePipes(LPVOID lpParameter)
{
    struct ThreadArgs *pTA =
        (struct ThreadArgs *)lpParameter;

    CListBox * pLB =
        (CListBox *) (CListBox::FromHandle(pTA->hWnd));
    char Str[256];

    pLB -> AddString ("Reading thread started");

    HANDLE PipeHandles[ NUM_PIPES ];
    DWORD BytesTransferred;
    CHAR Buffer[ NUM_PIPES ][ BUFFER_SIZE ];
    INT i;
    OVERLAPPED Ovlap[ NUM_PIPES ];
    HANDLE Event[ NUM_PIPES ];

    BOOL DataRead[ NUM_PIPES ];

    DWORD Ret;
    DWORD Pipe;

```

Здесь описаны массивы для хранения дескрипторов каналов, буферов ввода-вывода, структур, описывающих режим перекрытого ввода-вывода, событий, а также массив DataRead, который определяет тип текущей операции (равен FALSE, если данные требуется читать). Каждый элемент этих массивов относится к конкретному каналу. Количество последних задается константой NUM_PIPES.

Далее мы в цикле создаем экземпляры каналов, а также связанные с ними объекты типа событие, после чего иницилируем на каждом канале прослушивание клиентских соединений:

```
for(i = 0; i < NUM_PIPES; i++)
{
    // Создание экземпляра именованного канала
    if ((PipeHandles[i] = CreateNamedPipe(pTA->pName,
        PIPE_ACCESS_DUPLEX | FILE_FLAG_OVERLAPPED,
        PIPE_TYPE_BYTE | PIPE_READMODE_BYTE, NUM_PIPES,
        0, 0, 1000, NULL)) == INVALID_HANDLE_VALUE)
    {
        sprintf(Str, "CreateNamedPipe for pipe %d "
            "failed with error %d", i, GetLastError());
        pLB->AddString(Str);
        return 1;
    }

    // Для каждого экземпляра канала создается событие,
    // которое будет использоваться для определения
    // активности операций перекрытого ввода-вывода.
    if ((Event[i] = CreateEvent(NULL, TRUE, FALSE, NULL))
        == NULL)
    {
        sprintf(Str, "CreateEvent for pipe %d "
            "failed with error %d", i, GetLastError());
        pLB->AddString(Str);
        continue;
    }

    // Инициализация флага состояния, определяющего
    // писать или читать из канала
    DataRead[i] = FALSE;

    ZeroMemory(&Ovlap[i], sizeof(OVERLAPPED));
    Ovlap[i].hEvent = Event[i];

    // Прослушивание клиентских соединений
    if (ConnectNamedPipe(PipeHandles[i], &Ovlap[i]) == 0)
    {
        if (GetLastError() != ERROR_IO_PENDING)
        {
            sprintf(Str, "ConnectNamedPipe for pipe %d "
                "failed with error %d", i,
                GetLastError());
            pLB->AddString(Str);
        }
    }
}
```

```

        CloseHandle(PipeHandles[i]);
        return 1;
    }
}
}

```

```

sprintf(Str, "Server is now running");
pLB->AddString(Str);

```

Далее в бесконечном цикле мы читаем данные из канала и отправляем их обратно. Для ожидания очередного события используется функция WaitForMultipleObjects. В зависимости от значения элемента массива DataRead производится чтение или отправка данных. По завершении соединения с клиентом сервер вызывает функцию DisconnectNamedPipe и повторно иницирует прослушивание канала, чтобы мог подключиться следующий клиент.

```

// Чтение данных и отправка их обратно клиенту
// в бесконечном цикле
while(1)
{
    if ((Ret = WaitForMultipleObjects(NUM_PIPES, Event,
        FALSE, INFINITE)) == WAIT_FAILED)
    {
        sprintf(Str, "WaitForMultipleObjects failed "
            "with error %d", GetLastError());
        pLB->AddString(Str);
        return 1;
    }

    Pipe = Ret - WAIT_OBJECT_0;

    ResetEvent(Event[Pipe]);

    // Проверка результатов. В случае ошибки соединение
    // устанавливается заново,
    // иначе выполняется чтение и запись
    if (GetOverlappedResult(PipeHandles[Pipe],
        &Ovlap[Pipe], &BytesTransferred, TRUE) == 0)
    {
        sprintf(Str, "GetOverlappedResult failed "
            "with error %d", GetLastError());
        pLB->AddString(Str);

        if (DisconnectNamedPipe(PipeHandles[Pipe]) == 0)
        {
            sprintf(Str, "DisconnectNamedPipe failed "
                "with error %d", GetLastError());
            pLB->AddString(Str);
        }
    }
}

```

```

        return 1;
    }

    if (ConnectNamedPipe(PipeHandles[Pipe],
        &Ovlap[Pipe]) == 0)
    {
        if (GetLastError() != ERROR_IO_PENDING)
        {
            // Обработка ошибки канала.
            // Закрывание описателя.
            sprintf(Str, "ConnectNamedPipe for pipe"
                "%d failed with error %d",
                i, GetLastError());
            pLB->AddString(Str);
            CloseHandle(PipeHandles[Pipe]);
        }
    }

    DataRead[Pipe] = FALSE;
}
else
{
    // Проверка состояния канала. Если значение
    // DataRead равно FALSE, асинхронно читаем
    // данные клиента, иначе отправляем их обратно.
    if (DataRead[Pipe] == FALSE)
    {
        // Подготовка к чтению данных от клиента
        // путем асинхронного вызова функции ReadFile
        ZeroMemory(&Ovlap[Pipe], sizeof(OVERLAPPED));
        Ovlap[Pipe].hEvent = Event[Pipe];

        if (ReadFile(PipeHandles[Pipe], Buffer[Pipe],
            BUFFER_SIZE, NULL, &Ovlap[Pipe]) == 0)
        {
            if (GetLastError() != ERROR_IO_PENDING)
            {
                sprintf(Str, "ReadFile failed with "
                    "error %d", GetLastError());
                pLB->AddString(Str);
            }
        }

        DataRead[Pipe] = TRUE;
    }
}

```

```

else
{
    // Отправка данных обратно клиенту путем
    // асинхронного вызова функции WriteFile
    sprintf(Str, "Received %d bytes, echo "
               "bytes back", BytesTransferred);
    pLB->AddString(Str);

    ZeroMemory(&Ovlap[Pipe], sizeof(OVERLAPPED));
    Ovlap[Pipe].hEvent = Event[Pipe];

    if (WriteFile(PipeHandles[Pipe],
                  Buffer[Pipe], BytesTransferred,
                  NULL, &Ovlap[Pipe]) == 0)
    {
        if (GetLastError() != ERROR_IO_PENDING)
        {
            sprintf(Str, "WriteFile failed with "
                       "error %d", GetLastError());
            pLB->AddString(Str);
        }
    }

    DataRead[Pipe] = FALSE;
}
}
}
}
}

```

Сетевые протоколы

Пример 5.1. Перечисление установленных в системе сетевых протоколов

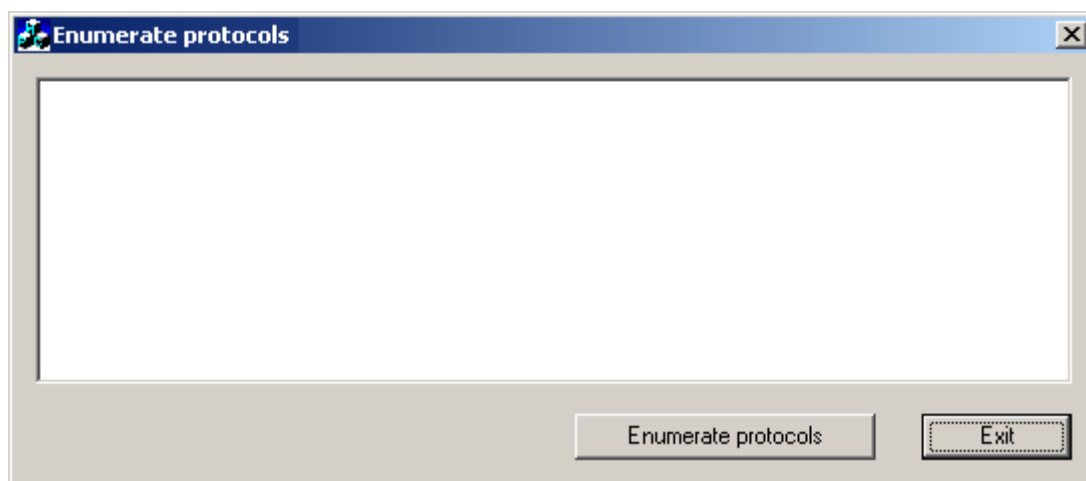
Пример демонстрирует способ перечисления доступных через интерфейс Winsock сетевых протоколов, а также определения их характеристик.

Открытие проекта

Откройте файл рабочего пространства заготовки проекта Enum.dsw, расположенный в директории 05_Protocols\Starter.

Главное диалоговое окно

Предлагаемый проект относится, как обычно, к диалоговым приложениям. Главное его окно выглядит следующим образом



Ниже перечислены основные элементы управления, расположенные на главном диалоговом окне приложения, их атрибуты и назначение.

Список (идентификатор IDC_LISTBOX) предназначен для вывода сообщений. Кнопка "Enumerate protocols" (идентификатор IDC_START) – для запуска процесса перечисления протоколов.

Добавление файлов, подключение библиотек, объявление констант и глобальных переменных

Скопируем в папку проекта файл af_irda.h (он не входит в стандартную установку Visual Studio 6).

В окне Project Settings на вкладке Link в поле Object/library modules: подключим библиотеку ws2_32.lib.

В начале файла EnumDlg.cpp добавим необходимые директивы препроцессора, объявления констант и глобальных переменных, а также глобальных функций:

```
//...

#include <winsock2.h>
#include <stdio.h>
#include <stdlib.h>

#define _WIN32_WINNT

#include "af_irda.h"

#include <ws2atm.h>
#include <wsipx.h>
#include <atalkwsh.h>

HWND      hWnd_LB;      // Для передачи рабочему потоку
CFont      font;

UINT MainThread(PVOID lpParam);
void PrintProtocolInfo(WSAPROTOCOL_INFO *wsapi);
BOOL EnumerateProtocols
        (WSAPROTOCOL_INFO **wsapi, DWORD *dwCount);

////////////////////////////////////
// CAboutDlg dialog used for App About
```

Здесь hWnd_LB – дескриптор окна списка IDC_LISTBOX, для того, чтобы рабочий поток имел возможность вывода информации в главном окне нашего приложения. Глобальная переменная font предназначена для создания шрифта, который будет использоваться при выводе в это окно списка. Имея в виду характер выводимой информации, представляется разумным использовать моноширинный шрифт.

Функция MainThread – главная функция рабочего потока, вся основная работа производится в отдельном потоке, чтобы не блокировать пользовательский интерфейс. Глобальная функция EnumerateProtocols осуществляет собственно перечисление протоколов, а функция PrintProtocolInfo предназначена для вывода информации о конкретном протоколе.

Определение переменных-членов класса EnumDlg

Для управления списком IDC_LISTBOX посредством ClassWizard создадим переменную m_ListBox типа CListBox.

Определение обработчиков событий

Создадим обработчик события, происходящего при нажатии кнопки "Enumerate protocols" (идентификатор IDC_START).

Обработчик OnStart() в предлагаемом ниже коде создает шрифт для вывода информации в окне списка и устанавливает его для данного окна. Он сохраняет дескриптор этого окна в глобальной переменной hWnd_LB, чтобы им могли пользоваться функции, запущенные в других потоках. После этого он создает основной рабочий поток приложения.

Вот ее исходный код:

```
void CEnumDlg::OnStart()
{
    font.CreateFont(16, 0, 0, 0, 400, FALSE, FALSE, 0,
        ANSI_CHARSET, OUT_DEFAULT_PRECIS,
        CLIP_DEFAULT_PRECIS, PROOF_QUALITY,
        DEFAULT_PITCH | FF_MODERN, "Courier New");

    hWnd_LB = m_ListBox.m_hWnd; // Для рабочего потока
    m_ListBox.SetFont(&font);
    m_ListBox.ResetContent();

    AfxBeginThread(MainThread, NULL);
}
```

Главная функция рабочего потока MainThread

Функция загружает библиотеку Winsock, затем последовательно вызывает функцию перечисления протоколов EnumerateProtocols и затем в цикле функцию вывода информации о каждом обнаруженном протоколе PrintProtocolInfo. Наконец, она производит необходимую очистку.

```
UINT MainThread(PVOID lpParam)
{
    WSAPROTOCOL_INFO    *wsapi=NULL;
    WSADATA              wsd;
    DWORD               dwCount;

    int    i;
    char    Str[200];
    CListBox * pLB =
        (CListBox *) (CListBox::FromHandle(hWnd_LB));

    if (WSAStartup(MAKEWORD(2, 2), &wsd) != 0)
    {
        sprintf(Str, "Unable to load Winsock2 DLL!");
        pLB->AddString(Str);
        return 1;
    }
}
```

```

if (EnumerateProtocols(&wsapi, &dwCount) == FALSE)
{
    sprintf(Str, "Failed to enumerate protocols!");
    LB->AddString(Str);
    return 1;
}
sprintf(Str, "Num Protocols found: %ld", dwCount);
pLB->AddString(Str);

for(i=0; i < dwCount ;i++)
{
    PrintProtocolInfo(&wsapi[i]);
}

WSACleanup();
return 0;
}

```

Функция перечисления протоколов EnumerateProtocols

Функция EnumerateProtocols для выполнения поставленной задачи использует функцию WSAEnumProtocols, которая заполняет массив структур WSAPROTOCOL_INFO информацией о поддерживаемых протоколах. Поскольку заранее размер буфера под этот массив неизвестен, используется следующий прием. При первом вызове функции WSAEnumProtocols в качестве адреса этого буфера передается значение NULL. Та через третий параметр возвращает размер требуемого буфера. После этого в памяти динамически размещается блок памяти требуемого размера, и при повторном вызове WSAEnumProtocols заполняет его требуемой информацией. При этом функция возвращает количество структур типа WSAPROTOCOL_INFO, помещенных в буфер.

```

BOOL EnumerateProtocols(WSAPROTOCOL_INFO **wsapi,
                        DWORD *dwCount)
{
    DWORD    dwErr,
             dwRet,
             dwBufLen=0;

    *dwCount = 0;
    *wsapi = NULL;

    char    Str[200];
    CListBox * pLB =
        (CListBox *) (CListBox::FromHandle(hWnd_LB));

```

```

if (SOCKET_ERROR !=
    WSAEnumProtocols(NULL, *wsapi, &dwBufLen))
{
    // Этого не может быть со значением NULL
    // в качестве буфера!
    //
    sprintf(Str, "WSAEnumProtocols failed!");
    pLB->AddString(Str);
    return FALSE;
}
else if (WSAENOBUFFS != (dwErr = WSAGetLastError()))
{
    // Ошибка по причине, не связанной с размером буфера
    // Такого здесь также не может произойти!
    //
    sprintf(Str, "WSAEnumProtocols failed: %d", dwErr);
    pLB->AddString(Str);
    return FALSE;
}
// Размещаем в памяти буфер требуемого размера
//
*wsapi =
    (WSAPROTOCOL_INFO *)GlobalAlloc(GMEM_FIXED, dwBufLen);

if (*wsapi == NULL)
{
    sprintf(Str, "GlobalAlloc failed: %d",
        GetLastError());
    pLB->AddString(Str);
    return FALSE;
}
dwRet = WSAEnumProtocols(NULL, *wsapi, &dwBufLen);

if (dwRet == SOCKET_ERROR)
{
    sprintf(Str, "WSAEnumProtocols failed: %d",
        GetLastError());
    pLB->AddString(Str);
    GlobalFree(*wsapi);
    return FALSE;
}

*dwCount = dwRet;
return TRUE;
}

```

Функция вывода информации о протоколе `PrintProtocolInfo`

Функция выводит в окно списка в удобном для восприятия виде информацию из структуры `WSAPROTOCOL_INFO`, описывающую характеристики данного протокола.

Исходный код данной функции весьма объемен и сводится исключительно к расшифровке отдельных полей структуры. Поэтому для краткости мы его здесь не приводим. Полностью его можно найти в файле `05_Protocols\Solution\EnumDlg.cpp`.

Основы интерфейса Winsock

Целью рассматриваемых в рамках данной темы примеров является изучение базовых (и только их) операций для осуществления установления связи и передачи информации по сети на основе интерфейса Winsock. Примеры, реализующие более сложные модели взаимодействия, рассматриваются в следующей теме.

Поскольку единственной зависимой от конкретного транспортного протокола операцией является фактически только создание сокета, все примеры здесь и далее базируются на использовании протокола IP, точнее протоколов, являющихся надстройками над ним: TCP и UDP.

Всего в рамках данной темы будет разработано четыре проекта. Первые два реализуют схему с установлением постоянного соединения по сценарию "сервер-клиент". Два следующих демонстрируют процесс обмена информацией без установления постоянного соединения.

Пример 7.1. Эхо-сервер на основе протокола TCP

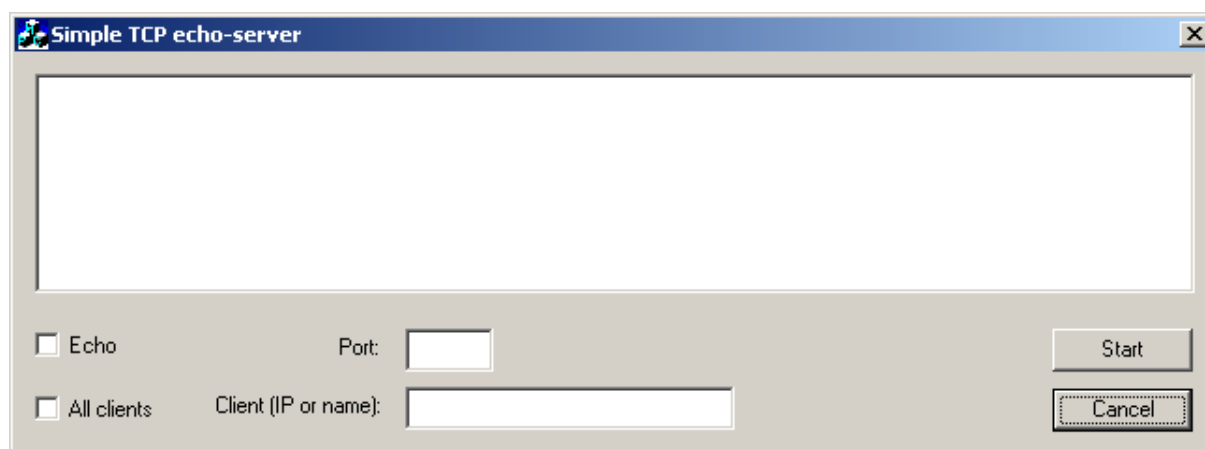
Сервер работает по следующему сценарию. Он создает сокет, привязывает его к локальному IP-интерфейсу и порту и слушает соединения клиентов. После приема от клиента запроса на соединение создается новый сокет, который передается клиентскому потоку. Поток читает данные и возвращает их клиенту.

Открытие проекта

Откройте файл рабочего пространства заготовки проекта Server.dsw, расположенный в директории 07_WinsockBasis\TCP\Server\Starter.

Главное диалоговое окно

Предлагаемая заготовка главного диалогового окна приложения выглядит следующим образом



На нем расположены следующие элементы управления.

Список (идентификатор IDC_LISTBOX) предназначен для вывода сообщений. Поля ввода (идентификаторы IDC_PORT и IDC_CLIENT) служат для ввода номера порта, на котором мы ждем соединений с клиентом, и имени клиента, если мы обслуживаем только конкретного адресата. Флажки (идентификаторы IDC_ECHO и IDC_ALL_ADDR) позволяют включить либо выключить эхо-режим, а также разрешить обслуживание всех обратившихся клиентов. В случае установки флажка IDC_ALL_ADDR обслуживаться должен только клиент, адрес которого введен в поле ввода IDC_CLIENT.

Кнопка "Start" (идентификатор IDC_START) предназначена для запуска сервера.

Подключение библиотек, объявление констант и глобальных переменных

В окне Project Settings на вкладке Link в поле Object/library modules: подключим библиотеку ws2_32.lib.

Подключим заголовок библиотеки Winsock2 в файле ServerDlg.h:

```
// ...
#include <winsock2.h>
/////////////////////////////////////////////////////////////////
// CServerDlg dialog
```

В начале файла ServerDlg.cpp добавим необходимые объявления констант, глобальных переменных и объявления глобальных функций:

```
#define DEFAULT_PORT    5150
#define DEFAULT_BUFFER  4096

int      iPort = DEFAULT_PORT; // Порт для прослушивания
BOOL     bInterface = FALSE,   // Конкретный клиент?
         bRecvOnly = FALSE;    // Только прием данных
char     szAddress[128];       // Интерфейс для прослушивания
HWND     hWnd_LB;              // Для вывода в других потоках

UINT ClientThread(PVOID lpParam);
UINT ListenThread(PVOID lpParam);
```

Назначение переменных ясно из комментариев. Объявления функций задают главную функцию потока прослушивания интерфейса и главную функцию потока обслуживания клиента соответственно.

Определение переменных и функций-членов класса CServerDlg

Добавим переменную `m_IsStarted` типа `bool` для указания, был ли сервер запущен. Для управления списком `IDC_LISTBOX` создадим переменную `m_ListBox` типа `CListBox`.

Объявим также функцию `SetStarted()` для изменения состояния элементов управления в зависимости от состояния сервера.

```
void SetStarted(bool IsStarted);
```

Код реализации функции `SetStarted()` может выглядеть, например, так:

```
void CServerDlg::SetStarted(bool IsStarted)
{
    m_IsStarted = IsStarted;

    GetDlgItem(IDC_ALL_ADDR) ->EnableWindow(!IsStarted);
    GetDlgItem(IDC_CLIENT) ->EnableWindow(bInterface);
    GetDlgItem(IDC_PORT) ->EnableWindow(!IsStarted);
    GetDlgItem(IDC_START) ->EnableWindow(!IsStarted);
}
```

В коде функции `CServerDlg::OnInitDialog()` настроим главное окно диалога перед его первым появлением:

```
// ...
// TODO: Add extra initialization here
char Str[128];

sprintf (Str, "%d", iPort);
GetDlgItem(IDC_PORT) ->SetWindowText (Str);
if (bRecvOnly)
    ((CButton *)GetDlgItem(IDC_ECHO)) ->SetCheck(0);
else
    ((CButton *)GetDlgItem(IDC_ECHO)) ->SetCheck(1);

if (bInterface)
    ((CButton *)GetDlgItem(IDC_ALL_ADDR)) ->SetCheck(0);
else
    ((CButton *)GetDlgItem(IDC_ALL_ADDR)) ->SetCheck(1);

SetStarted(false);
// ...
```

Определение и реализация обработчиков событий

Нам потребуются обработчики трех событий: два – для оперативного отслеживания состояний двух флажков (идентификаторы `IDC_ECHO` и

IDC_ALL_ADDR), а также обработчик события, происходящего при нажатии кнопки "Start" (идентификатор IDC_START).

Код первых двух не требует комментариев:

```
void CServerDlg::OnEcho()
{
    if (((CButton *)GetDlgItem(IDC_ECHO)) ->GetCheck() == 1)
        bRecvOnly = FALSE;
    else
        bRecvOnly = TRUE;
}
```

```
void CServerDlg::OnAllAddr()
{
    if (((CButton *)GetDlgItem(IDC_ALL_ADDR)) ->
                                                GetCheck() == 1)
        bInterface = FALSE;
    else
        bInterface = TRUE;

    GetDlgItem(IDC_CLIENT) ->EnableWindow(bInterface);
}
```

Обработчик OnStart() запоминает дескриптор окна списка для вывода информации, сохраняет введенные пользователем номер порта и адрес клиента, затем запускает поток прослушивания порта. В завершение он приводит состояние интерфейса пользователя в соответствие с состоянием приложения.

```
void CServerDlg::OnStart()
{
    char Str[81];

    hWnd_LB = m_ListBox.m_hWnd; // Для потоков
    GetDlgItem(IDC_PORT) ->GetWindowText(Str, sizeof(Str));
    iPort=atoi(Str);
    if (iPort<=0 || iPort>=0x10000)
    {
        AfxMessageBox("Incorrect Port number");
        return;
    }
    GetDlgItem(IDC_CLIENT) ->
        GetWindowText(szAddress, sizeof(szAddress));

    AfxBeginThread(ListenThread, NULL);

    SetStarted(true);
}
```


Главная функция потока прослушивания

Функция ListenThread инициализирует библиотеку Winsock, создает сокет для прослушивания, привязывает его к локальному адресу и ждет подключений клиентов. Для обслуживания каждого клиента создается отдельный поток с главной функцией ClientThread.

Разберем основные участки кода. Сначала мы описываем необходимые локальные переменные и загружаем библиотеку Winsock:

```
UINT ListenThread(PVOID lpParam)
{
    WSADATA    wsd;
    SOCKET     sListen,
               sClient;
    int        iAddrSize;

    struct sockaddr_in  local,
                        client;

    char    Str[200];
    CListBox* pLB =
        (CListBox *) (CListBox::FromHandle(hWndd_LB));

    if (WSAStartup(MAKEWORD(2,2), &wsd) != 0)
    {
        sprintf(Str, "Failed to load Winsock!");
        pLB->AddString(Str);
        return 1;
    }
}
```

Затем создаем сокет для прослушивания:

```
sListen = socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
if (sListen == SOCKET_ERROR)
{
    sprintf(Str, "socket() failed: %d",
            WSAGetLastError());
    pLB->AddString(Str);
    WSACleanup();
    return 1;
}
```

Далее в зависимости от значения переменной bInterface (равна true, если работаем с одним конкретным клиентом) выбираем интерфейс для привязки, вызываем функцию bind и выставляем сокет на прослушивание. Напомним здесь, что функция inet_addr() принимает IP-адрес в формате a.b.c.d и преобразует его в сетевой формат. В случае неправильно оформленной строки на входе она возвращает специальное значение INADDR_NONE, свидетельствующее об ошибке.

```

// Выбрать локальный интерфейс и привязаться к нему
if (bInterface)
{
    local.sin_addr.s_addr = inet_addr(szAddress);
    if (local.sin_addr.s_addr == INADDR_NONE)
    {
        AfxMessageBox("Incorrect local interface");
        closesocket(sListen);
        WSACleanup();
        return 1;
    }
}
else
    local.sin_addr.s_addr = htonl(INADDR_ANY);
local.sin_family = AF_INET;
local.sin_port = htons(iPort);

if (bind(sListen, (struct sockaddr *)&local,
        sizeof(local)) == SOCKET_ERROR)
{
    sprintf(Str, "bind() failed: %d",
                                                    WSAGetLastError());

    pLB->AddString(Str);
    closesocket(sListen);
    WSACleanup();
    return 1;
}

listen(sListen, 8);

```

Наконец, в цикле обслуживанием подключения клиентов. Для каждого нового подключения создаем поток обслуживания и продолжаем прослушивание:

```

// Ожидание клиентов, создание потока для каждого соединения
while (1)
{
    iAddrSize = sizeof(client);
    sClient = accept(sListen,
                    (struct sockaddr *)&client, &iAddrSize);
    if (sClient == INVALID_SOCKET)
    {
        sprintf(Str, "accept() failed: %d",
                                                    WSAGetLastError());
        pLB->AddString(Str);
        break;
    }
}

```

```

        sprintf(Str, "Accepted client: %s:%d",
                  inet_ntoa(client.sin_addr),
                  ntohs(client.sin_port));
        pLB->AddString(Str);

        AfxBeginThread(ClientThread, (LPVOID)sClient);
    }
    closesocket(sListen);

    WSACleanup();
    return 0;
}

```

Главная функция потока обслуживания клиента

Функция ClientThread() через свой параметр получает дескриптор сокета для созданного соединения (его вернула функция accept()). Она получает данные от клиентов и, если установлена соответствующая опция, отправляет их назад. Для простоты здесь используются блокирующие функции приема и отправки данных. Поток работает, пока не закрыт сеанс или пока не произойдет ошибка.

Ниже приводится соответствующий исходный код.

```

UINT ClientThread(PVOID lpParam)
{
    SOCKET  sock=(SOCKET) lpParam;
    char     szBuff[DEFAULT_BUFFER];
    int      ret,
            nLeft,
            idx;
    char     Str[200];
    CListBox* pLB =
                (CListBox *) (CListBox::FromHandle(hWnd_LB));

    while(1)
    {
        // Блокирующий вызов recv()
        //
        ret = recv(sock, szBuff, DEFAULT_BUFFER, 0);
        if (ret == 0)    // Корректное завершение
            break;
        else if (ret == SOCKET_ERROR)
        {
            sprintf(Str, "recv() failed: %d",
                    WSAGetLastError());
            pLB->AddString(Str);
            break;
        }
    }
}

```

```

szBuff[ret] = '\\0';
sprintf(Str, "RCV: '%s'", szBuff);
pLB->AddString(Str);

// Ответная отправка данных, если требуется
if (!bRecvOnly)
{
    nLeft = ret;
    idx = 0;

    // Проверка, что все данные записаны
    //
    while(nLeft > 0)
    {
        ret = send(sock, &szBuff[idx], nLeft, 0);
        if (ret == 0)
            break;
        else if (ret == SOCKET_ERROR)
        {
            sprintf(Str, "send() failed: %d",
                    WSAGetLastError());
            pLB->AddString(Str);
            break;
        }
        nLeft -= ret;
        idx += ret;
    }
}
return 0;
}

```

Пример 7.2. Клиент для эхо-сервера на основе протокола TCP

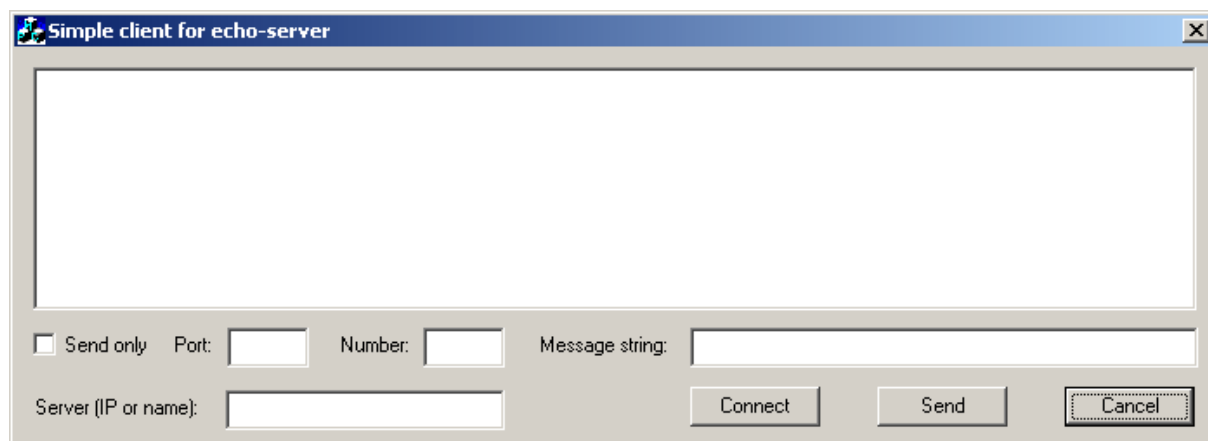
Клиент создает сокет, разрешает переданное приложению имя сервера и соединяется с сервером. Затем он отправляет несколько сообщений. После каждой отправки, если задана соответствующая опция, клиент ожидает эхо-ответа сервера. Информация об отправке и приеме сообщений выводится на экран. Для простоты вся работа происходит в одном потоке. Обратите внимание, как в данном примере происходит процесс разрешения имени сервера (превращение его из символьного вида в IP-адрес). В предыдущем примере (эхо-сервер) по отношению к адресу клиента мы такую возможность для краткости опустили. При желании читатели могут теперь легко дополнить только что созданный код.

Открытие проекта

Откройте файл рабочего пространства заготовки проекта Client.dsw, расположенный в директории 07_WinsockBasis\TCP\Client\Starter.

Главное диалоговое окно

Предлагаемая стартовая заготовка главного диалогового окна приложения выглядит следующим образом



На нем расположены следующие элементы управления.

Список (идентификатор IDC_LISTBOX) предназначен для вывода сообщений. Поля ввода (идентификаторы IDC_PORT, IDC_SERVER, IDC_NUMBER и IDC_MESSAGE) служат для ввода номера порта, имени или IP-адреса сервера, количества посылаемых сообщений и строки для отправки. Флажок (идентификатор IDC_NO_ECHO) указывает, что ответных сообщений дожидаться не требуется.

Обратите внимание на то, что в рассматриваемых здесь примерах работа сервера и клиента должна быть согласована. Если, например, клиент ждет, что сервер пришлет эхо-ответ, а у того выключена соответствующая опция, то работа клиента будет заблокирована. Причиной является крайне примитивная модель взаимодействия, используемая в рамках данной темы (мы здесь преследуем иные цели). Более сложные и, соответственно, более приспособленные для разработки реальных приложений модели ввода-вывода будут рассмотрены в следующей теме.

Наконец, кнопка "Connect" (идентификатор IDC_CONNECT) предназначена для подключения к серверу, а кнопка "Send" (идентификатор IDC_SEND) – для отправки и приема сообщений.

Подключение библиотек, объявление констант и глобальных переменных

В окне Project Settings на вкладке Link в поле Object/library modules: подключим библиотеку ws2_32.lib.

Подключим заголовок библиотеки Winsock2 в файле ClientDlg.h:

```
// ...
#include <winsock2.h>
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// CClientDlg dialog
```

В начале файла ClientDlg.cpp добавим необходимые объявления констант:

```
#define DEFAULT_COUNT    10
#define DEFAULT_PORT     5150
#define DEFAULT_BUFFER   2048
#define DEFAULT_MESSAGE  "This is a test message"
```

Определение переменных и функций-членов класса CClientDlg

Добавим переменную m_sClient типа SOCKET для хранения дескриптора сокета. Также добавим переменную m_IsConnected типа bool для указания, произошло ли подключение к серверу.

Объявим также функцию SetConnected() для изменения состояния элементов управления в зависимости от состояния клиента.

```
bool    m_IsConnected;
SOCKET  m_sClient;

void SetConnected (bool IsConnected);
```

Код реализации функции SetConnected() может выглядеть примерно так:

```
void CClientDlg::SetConnected (bool IsConnected)
{
    m_IsConnected = IsConnected;

    GetDlgItem(IDC_SEND) ->EnableWindow(IsConnected);
    GetDlgItem(IDC_MESSAGE) ->EnableWindow(IsConnected);
    GetDlgItem(IDC_NUMBER) ->EnableWindow(IsConnected);
    GetDlgItem(IDC_NO_ECHO) ->EnableWindow(IsConnected);
    GetDlgItem(IDC_SERVER) ->EnableWindow(!IsConnected);
    GetDlgItem(IDC_PORT) ->EnableWindow(!IsConnected);
    GetDlgItem(IDC_CONNECT) ->EnableWindow(!IsConnected);
}
```

Посредством ClassWizard для управления списком IDC_LISTBOX создадим переменную m_ListBox типа CListBox. Также свяжем с флажком IDC_NO_ECHO переменную m_NoEcho типа CButton, а с полем IDC_NUMBER переменную m_Number типа int.

В коде функции CClientDlg::OnInitDialog() настроим главное окно диалога перед его первым появлением:

```
// ...
// TODO: Add extra initialization here
char Str[128];

GetDlgItem(IDC_SERVER)->SetWindowText("localhost");
sprintf(Str, "%d", DEFAULT_COUNT);
GetDlgItem(IDC_NUMBER)->SetWindowText(Str);
sprintf(Str, "%d", DEFAULT_PORT);
GetDlgItem(IDC_PORT)->SetWindowText(Str);
GetDlgItem(IDC_MESSAGE)->SetWindowText(DEFAULT_MESSAGE);
m_NoEcho.SetCheck(0);
SetConnected(false);
// ...
```

Определение и реализация обработчиков событий

Нам потребуются обработчики нажатий кнопок "Connect" (идентификатор IDC_CONNECT) и "Send" (идентификатор IDC_SEND).

Обработчик OnConnect создает сокет, при необходимости производит разрешение имени сервера и вызывает функцию connect для подключения. Разберем основные этапы этого процесса.

Сначала опишем необходимые локальные переменные и загрузим библиотеку Winsock:

```
void CClientDlg::OnConnect()
{
    char    szServer[128]; // Имя или IP-адрес сервера
    int     iPort;        // Порт

    WSADATA wsd;

    struct sockaddr_in  server;
    struct hostent      *host = NULL;

    char Str[256];

    GetDlgItem(IDC_SERVER)->
        GetWindowText(szServer, sizeof(szServer));
    GetDlgItem(IDC_PORT)->GetWindowText(Str, sizeof(Str));
    iPort = atoi(Str);
    if (iPort<=0 || iPort>=0x10000)
    {
        m_ListBox.AddString("Port number incorrect");
        return;
    }
}
```

```

if (WSAStartup(MAKEWORD(2,2), &wsd) != 0)
{
    m_ListBox.AddString
        ("Failed to load Winsock library!");
    return;
}

```

Далее создадим сокет:

```

m_sClient = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if (m_sClient == INVALID_SOCKET)
{
    sprintf(Str, "socket() failed: %d\n",
        WSAGetLastError());
    m_ListBox.AddString(Str);
    return;
}
server.sin_family = AF_INET;
server.sin_port = htons(iPort);
server.sin_addr.s_addr = inet_addr(szServer);

```

Если адрес сервера не представлен в форме "aaa.bbb.ccc.ddd", это имя узла. Пробуем его разрешить, после этого вызываем функцию connect(). Наконец, перенастраиваем пользовательский интерфейс.

```

if (server.sin_addr.s_addr == INADDR_NONE)
{
    host = gethostbyname(szServer);
    if (host == NULL)
    {
        sprintf(Str, "Unable to resolve server: %s",
            szServer);
        m_ListBox.AddString(Str);
        return;
    }
    CopyMemory(&server.sin_addr, host->h_addr_list[0],
        host->h_length);
}
if (connect(m_sClient, (struct sockaddr *)&server,
    sizeof(server)) == SOCKET_ERROR)
{
    sprintf(Str, "connect() failed: %d",
        WSAGetLastError());
    m_ListBox.AddString(Str);
    return;
}
SetConnected(true);
}

```


Обработчик OnSend() отправляет и, если надо, принимает сообщения. По завершении сеанса он закрывает сокет и возвращает приложение в исходное состояние.

```
void CClientDlg::OnSend()
{
    charszMessage[1024];    // Сообщение для отправки
    BOOLbSendOnly = FALSE;  // Только отправка данных

    char    szBuffer[DEFAULT_BUFFER];
    int     ret,
           i;

    char    Str[256];

    UpdateData();
    if (m_Number<1 || m_Number>20)
    {
        m_ListBox.AddString
            ("Number of messages must be between 1 and 20");
        return;
    }

    GetDlgItem(IDC_MESSAGE)->
        GetWindowText(szMessage, sizeof(szMessage));
    if (m_NoEcho.GetCheck() == 1)
        bSendOnly = TRUE;

    // Отправка и прием данных
    //
    for(i = 0; i < m_Number; i++)
    {
        ret = send(m_sClient, szMessage, strlen(szMessage), 0);

        if (ret == 0)
            break;
        else if (ret == SOCKET_ERROR)
        {
            sprintf(Str, "send() failed: %d",
                    WSAGetLastError());
            m_ListBox.AddString(Str);
            break;
        }

        printf("Send %d bytes\n", ret);
    }
}
```

```

        if (!bSendOnly)
        {
            ret = recv(m_sClient, szBuffer, DEFAULT_BUFFER, 0);
            if (ret == 0) // Корректное завершение
                break;
            else if (ret == SOCKET_ERROR)
            {
                sprintf(Str, "recv() failed: %d",
                        WSAGetLastError());
                m_ListBox.AddString(Str);
                break;
            }
            szBuffer[ret] = '\\0';
            sprintf(Str, "RECV [%d bytes]: '%s'",
                    ret, szBuffer);
            m_ListBox.AddString(Str);
        }
    }
    closesocket(m_sClient);

    WSACleanup();

    SetConnected(false);
}

```

Пример 7.3. Получатель дейтаграмм на основе протокола UDP

Два следующих примера демонстрируют процесс обмена информацией без установления постоянного соединения с использованием протокола UDP.

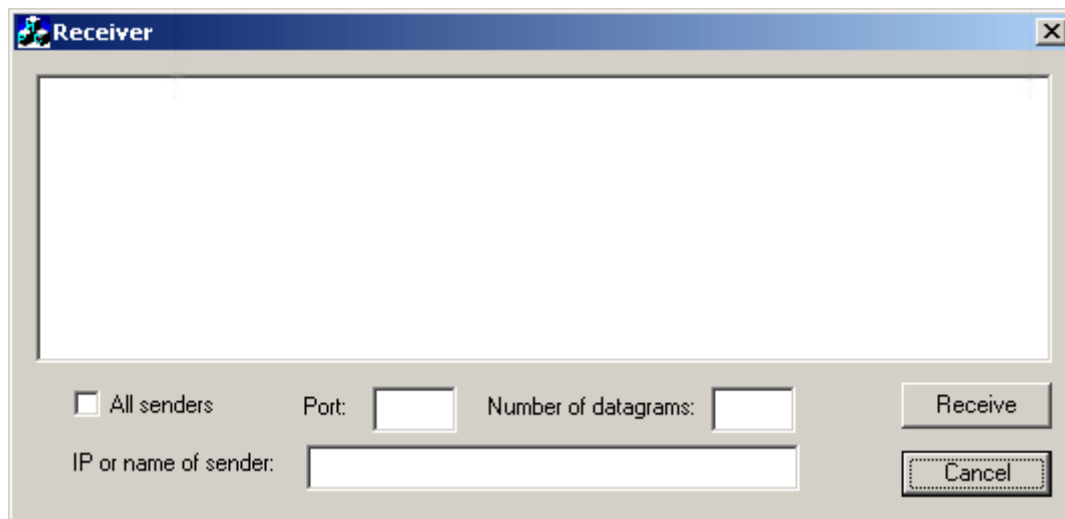
Действия приложения-приемника таковы. Создается сокет и привязывается к локальному интерфейсу. После этого для чтения данных просто вызывается функция `recvfrom()`. Для того чтобы не заблокировать интерфейс пользователя, прием данных осуществляется в отдельном потоке.

Открытие проекта

Откройте файл рабочего пространства заготовки проекта `Receiver.dsw`, расположенный в директории `07_WinsockBasis\UDP\Receiver\Starter`.

Главное диалоговое окно

Следующий рисунок демонстрирует внешний вид приложения, предлагаемого в качестве стартового проекта.



На главном диалоговом окне приложения расположены следующие элементы управления.

Список (идентификатор IDC_LISTBOX) предназначен для вывода сообщений. Поля ввода (идентификаторы IDC_PORT, IDC_SENDER и IDC_COUNT) служат для ввода номера порта, на котором мы ждем дейтаграмм, имени отправителя (или его IP-адреса), если мы принимаем данные только от конкретного адресата, а также количества принимаемых дейтаграмм. Установка флажка (идентификатор IDC_ALL_IP) позволяет принимать данные от всех отправителей.

Кнопка "Receive" (идентификатор IDC_RECEIVE) запускает процесс приема дейтаграмм.

Подключение библиотек, объявление констант, глобальных переменных и функций

В окне Project Settings на вкладке Link в поле Object/library modules: подключим библиотеку ws2_32.lib.

Подключим заголовок библиотеки Winsock2 в файле ReceiverDlg.h:

```
// ...
#include <winsock2.h>
/////////////////////////////////////////////////////////////////
// CReceiverDlg dialog
```

В начале файла ReceiverDlg.cpp добавим необходимые объявления констант, глобальных переменных и объявления глобальных функций:

```
#define DEFAULT_PORT          5150
#define DEFAULT_COUNT         25
#define DEFAULT_BUFFER_LENGTH 4096

int      iPort = DEFAULT_PORT;    // Номер порта
DWORD    dwCount = DEFAULT_COUNT, // Количество дейтаграмм
          dwLength = DEFAULT_BUFFER_LENGTH; // Размер буфера
BOOL     bInterface = FALSE;    // Конкретный отправитель?
```

```
char    szInterface[32];    // Адрес отправителя
HWND    hWnd_LB;           // Для вывода в другом потоке

UINT ReceiveThread(PVOID lpParam);
```

Назначение переменных ясно из комментариев. Глобальная функция ReceiveThread() представляет собой главную функцию рабочего потока получения дейтаграмм.

Определение переменных и функций-членов класса CReceiverDlg

Добавим переменную m_IsStarted типа bool для указания, был ли запущен процесс приема дейтаграмм. Для управления списком IDC_LISTBOX создадим переменную m_ListBox типа CListBox.

Объявим также функцию SetStarted() для изменения состояния элементов управления в зависимости от состояния сервера.

```
void SetStarted(bool IsStarted);
```

Предлагаемый вариант кода реализации функции SetStarted():

```
void CReceiverDlg::SetStarted(bool IsStarted)
{
    m_IsStarted = IsStarted;

    GetDlgItem(IDC_ALL_IP) ->EnableWindow(!IsStarted);
    GetDlgItem(IDC_SENDER) ->EnableWindow(bInterface);
    GetDlgItem(IDC_PORT) ->EnableWindow(!IsStarted);
    GetDlgItem(IDC_COUNT) ->EnableWindow(!IsStarted);
    GetDlgItem(IDC_RECEIVE) ->EnableWindow(!IsStarted);
}
```

В коде функции CReceiverDlg::OnInitDialog() настроим главное окно диалога перед его первым появлением:

```
// ...
// TODO: Add extra initialization here
char Str[128];

sprintf (Str, "%d", iPort);
GetDlgItem(IDC_PORT) ->SetWindowText (Str);

sprintf (Str, "%d", dwCount);
GetDlgItem(IDC_COUNT) ->SetWindowText (Str);

if (bInterface)
    ((CButton *)GetDlgItem(IDC_ALL_IP)) ->SetCheck(0);
else
    ((CButton *)GetDlgItem(IDC_ALL_IP)) ->SetCheck(1);

SetStarted(false);
// ...
```

Определение и реализация обработчиков событий

Нам потребуются обработчики двух событий: одно – для оперативного отслеживания состояния флажка (идентификатор IDC_ALL_IP), другое – для обработки события, происходящего при нажатии кнопки "Receive" (идентификатор IDC_RECEIVE).

Код первого не требует комментариев:

```
void CReceiverDlg::OnAllIp()
{
    if (((CButton *)GetDlgItem(IDC_ALL_IP)) ->GetCheck() == 1)
        bInterface = FALSE;
    else
        bInterface = TRUE;

    GetDlgItem(IDC_SENDER) ->EnableWindow(bInterface);
}
```

Обработчик OnReceive() запоминает в глобальных переменных информацию, необходимую для процесса приема, затем запускает поток получения дейтаграмм. В завершение он приводит состояние интерфейса пользователя в соответствие с состоянием приложения.

```
void CReceiverDlg::OnReceive()
{
    char Str[81];

    hWnd_LB = m_ListBox.m_hWnd; // Для потока
    GetDlgItem(IDC_PORT) ->GetWindowText(Str, sizeof(Str));
    iPort=atoi(Str);
    if (iPort<=0 || iPort>=0x10000)
    {
        AfxMessageBox("Incorrect Port number");
        return;
    }

    GetDlgItem(IDC_COUNT) ->GetWindowText(Str, sizeof(Str));
    dwCount=atoi(Str);
    if (dwCount<=0)
    {
        AfxMessageBox("Incorrect number of datagrams");
        return;
    }

    GetDlgItem(IDC_SENDER) ->
        GetWindowText(szInterface, sizeof(szInterface));

    AfxBeginThread(ReceiveThread, NULL);

    SetStarted(true);
}
```

Главная функция потока получения дейтаграмм

Функция `ReceiveThread()` инициализирует библиотеку Winsock, создает сокет для приема, привязывает его к локальному адресу и порту, после чего начинает прием дейтаграмм.

Разберем основные участки кода. Сначала мы описываем необходимые локальные переменные и загружаем библиотеку Winsock:

```
UINT ReceiveThread(PVOID lpParam)
{
    WSADATA    wsd;
    SOCKET     s;
    char       *recvbuf = NULL;
    int        ret,
              i;
    DWORD      dwSenderSize;
    SOCKADDR_IN sender,
              local;

    char       Str[200];
    CListBox*  pLB =
        (CListBox *) (CListBox::FromHandle(hWnd_LB));

    if (WSAStartup(MAKEWORD(2,2), &wsd) != 0)
    {
        sprintf(Str, "WSAStartup failed!");
        pLB->AddString(Str);
        return 1;
    }

    Затем создаем сокет для приема дейтаграмм и привязываем его к
    интерфейсу:

    s = socket(AF_INET, SOCK_DGRAM, 0);
    if (s == INVALID_SOCKET)
    {
        sprintf(Str, "socket() failed; %d",
                WSAGetLastError());
        pLB->AddString(Str);
        return 1;
    }
    local.sin_family = AF_INET;
    local.sin_port = htons((short)iPort);
    if (bInterface)
        local.sin_addr.s_addr = inet_addr(szInterface);
    else
        local.sin_addr.s_addr = htonl(INADDR_ANY);
```

```

if (bind(s, (SOCKADDR *)&local, sizeof(local)) ==
                                SOCKET_ERROR)
{
    sprintf(Str, "bind() failed: %d", WSAGetLastError());
    pLB->AddString(Str);
    return 1;
}

```

Выделяем буфер для приема информации:

```

recvbuf = (char *)GlobalAlloc(GMEM_FIXED, dwLength);
if (!recvbuf)
{
    sprintf(Str, "GlobalAlloc() failed: %d",
                                GetLastError());
    pLB->AddString(Str);
    return 1;
}

```

Наконец, в цикле получаем дейтаграммы. По завершении освобождаем захваченные ресурсы:

```

for(i = 0; i < dwCount; i++)
{
    dwSenderSize = sizeof(sender);
    ret = recvfrom(s, recvbuf, dwLength, 0,
                  (SOCKADDR *)&sender, (int *)&dwSenderSize);
    if (ret == SOCKET_ERROR)
    {
        sprintf(Str, "recvfrom() failed; %d",
                                WSAGetLastError());
        pLB->AddString(Str);
        break;
    }
    else if (ret == 0)
        break;
    else
    {
        recvbuf[ret] = '\0';
        sprintf(Str, "[%s] sent me: '%s'",
                inet_ntoa(sender.sin_addr), recvbuf);
        pLB->AddString(Str);
    }
}

```

```

sprintf(Str, "%d datagrams was received. Socket closed",
        dwCount);
pLB->AddString(Str);

```

```

    closesocket(s);

    GlobalFree(recvbuf);
    WSACleanup();

    return 0;
}

```

Пример 7.4. Отправитель дейтаграмм на основе протокола UDP

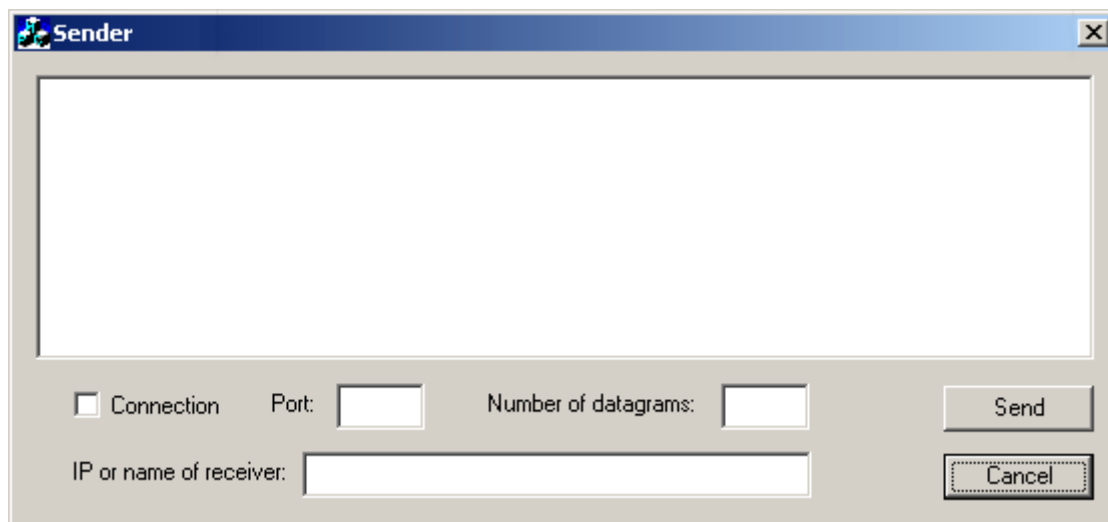
Отправитель через интерфейс пользователя получает IP-адрес приемника и номер порта. Он создает сокет для отправки данных и отправляет дейтаграммы, используя функцию `sendto()`. Можно дополнительно задать опцию вызова функции `connect` для привязки сокета к адресу (в этом случае для отправки используется функция `send()`, а не `sendto()`).

Открытие проекта

Откройте файл рабочего пространства заготовки проекта `Sender.dsw`, расположенный в директории `07_WinsockBasis\UDP\Sender\Starter`.

Главное диалоговое окно

Приложение организовано как диалоговое со следующим главным окном



На главном диалоговом окне приложения расположены следующие элементы управления.

Список (идентификатор `IDC_LISTBOX`) предназначен для вывода сообщений. Поля ввода (идентификаторы `IDC_PORT`, `IDC_RECEIVER` и `IDC_COUNT`) служат для ввода номера порта и имени получателя (или его IP-адреса), а также количества посылаемых дейтаграмм. Флажок (иденти-

фикатор IDC_CONNECTION) означает необходимость вызова функции connect() для привязки сокета к конкретному получателю.

Кнопка "Send" (идентификатор IDC_SEND) запускает процесс отправки дейтаграмм.

Подключение библиотек, объявление констант, глобальных переменных и функций

В окне Project Settings на вкладке Link в поле Object/library modules: подключим библиотеку ws2_32.lib.

Подключим заголовок библиотеки Winsock2 в файле SenderDlg.h:

```
// ...
#include <winsock2.h>
////////////////////////////////////
// CSenderDlg dialog
```

В начале файла SenderDlg.cpp добавим необходимые объявления констант, глобальных переменных и объявления глобальных функций:

```
#define DEFAULT_PORT          5150
#define DEFAULT_COUNT        25
#define DEFAULT_BUFFER_LENGTH 64

BOOL  bConnect = FALSE;      // Вызывать connect?
int    iPort = DEFAULT_PORT;  // Номер порта
DWORD dwCount = DEFAULT_COUNT; // Количество дейтаграмм
char  szRecipient[128];      // Имя или адрес получателя

CListBox *pListBox; // Для функции DatagramSend

int DatagramSend();
```

Назначение переменных ясно из комментариев. Глобальная функция DatagramSend() выполняет всю основную работу по отправке данных.

Определение переменных и начальная настройка диалогового окна

Создадим переменную m_ListBox типа CListBox для управления списком IDC_LISTBOX.

В коде функции CSenderDlg::OnInitDialog() настроим главное окно диалога перед его первым появлением:

```
// ...
// TODO: Add extra initialization here
char Str[128];

GetDlgItem(IDC_RECEIVER) ->SetWindowText("localhost");
sprintf (Str, "%d", iPort);
GetDlgItem(IDC_PORT) ->SetWindowText(Str);
```

```

    sprintf (Str, "%d", dwCount);
    GetDlgItem(IDC_COUNT) ->SetWindowText (Str);

    if (!bConnect)
        ((CButton *)GetDlgItem(IDC_CONNECTION)) ->SetCheck(0);
    else
        ((CButton *)GetDlgItem(IDC_CONNECTION)) ->SetCheck(1);
// ...

```

Определение и реализация обработчиков событий

Нам потребуются обработчики двух событий: одно – для оперативного отслеживания состояния флажка (идентификатор IDC_CONNECTION), второе для обработки события, происходящего при нажатии кнопки "Send" (идентификатор IDC_SEND).

Код первого достаточно очевиден:

```

void CSenderDlg::OnConnection()
{
    if (((CButton *)GetDlgItem(IDC_CONNECTION)) ->
                                                GetCheck() == 1)
        bConnect = TRUE;
    else
        bConnect = FALSE;
}

```

Обработчик OnSend() запоминает в глобальных переменных всю информацию, необходимую для процесса отправки данных, а затем вызывает функцию DatagramSend(), выполняющую всю работу.

```

void CSenderDlg::OnSend()
{
    char Str[81];

    GetDlgItem(IDC_PORT) ->GetWindowText(Str, sizeof(Str));
    iPort=atoi(Str);
    if (iPort<=0 || iPort>=0x10000)
    {
        AfxMessageBox("Incorrect Port number");
        return;
    }

    GetDlgItem(IDC_COUNT) ->GetWindowText(Str, sizeof(Str));
    dwCount=atoi(Str);
    if (dwCount<=0)
    {
        AfxMessageBox("Incorrect number of datagrams");
        return;
    }
}

```

```

        GetDlgItem(IDC_RECEIVER) ->
            GetWindowText(szRecipient, sizeof(szRecipient));

        pListBox = &m_ListBox;
        DatagramSend();
    }

```

Глобальная функция отправки дейтаграмм DatagramSend

Функция DatagramSend() инициализирует библиотеку Winsock, затем создает сокет для отправки данных и, если задана соответствующая опция, подключает его к удаленному компьютеру. После этого она начинает отправку дейтаграмм.

Разберем основные участки кода. Сначала мы описываем необходимые локальные переменные и загружаем библиотеку Winsock:

```

int DatagramSend()
{
    WSADATA wsd;
    SOCKET s;
    char *sendbuf = NULL;
    int ret,
        i;
    SOCKADDR_IN recipient;
    char Str[200];

    if (WSAStartup(MAKEWORD(2, 2), &wsd) != 0)
    {
        sprintf(Str, "WSAStartup failed!");
        pListBox->AddString(Str);
        return 1;
    }

```

Затем создаем сокет для отправки дейтаграмм и осуществляем разрешение адреса получателя:

```

    s = socket(AF_INET, SOCK_DGRAM, 0);
    if (s == INVALID_SOCKET)
    {
        sprintf(Str, "socket() failed; %d",
            WSAGetLastError());
        pListBox->AddString(Str);
        return 1;
    }
    // Разрешить имя или IP-адрес приемника
    //
    recipient.sin_family = AF_INET;
    recipient.sin_port = htons((short)iPort);

```

```

if ((recipient.sin_addr.s_addr = inet_addr(szRecipient))
    == INADDR_NONE)
{
    struct hostent *host=NULL;
    host = gethostbyname(szRecipient);
    if (host)
        CopyMemory(&recipient.sin_addr,
                    host->h_addr_list[0], host->h_length);
    else
    {
        sprintf(Str, "gethostbyname() failed: %d",
                WSAGetLastError());
        pListBox->AddString(Str);
        WSACleanup();
        return 1;
    }
}

```

Выделяем буфер для отправки данных:

```

sendbuf = (char *)GlobalAlloc(GMEM_FIXED,
    DEFAULT_BUFFER_LENGTH);
if (!sendbuf)
{
    sprintf(Str, "GlobalAlloc() failed: %d",
            GetLastError());
    pListBox->AddString(Str);
    return 1;
}

```

Наконец, в цикле отправляем дейтаграммы. По завершении освобождаем захваченные ресурсы:

```

// Если задана опция connect, выполняется "подключение"
// и отправка с использованием функции send()
if (bConnect)
{
    if (connect(s, (SOCKADDR *)&recipient,
                sizeof(recipient)) == SOCKET_ERROR)
    {
        sprintf(Str, "connect() failed: %d",
                WSAGetLastError());
        pListBox->AddString(Str);
        GlobalFree(sendbuf);
        WSACleanup();
        return 1;
    }
}

```

```

for(i = 0; i < dwCount; i++)
{
    sprintf (sendbuf,
        "Test datagram (with connection) #%d", i);
    ret = send(s, sendbuf, strlen(sendbuf), 0);
    if (ret == SOCKET_ERROR)
    {
        sprintf(Str, "send() failed: %d",
            WSAGetLastError());
        pListBox->AddString(Str);
        break;
    }
    else if (ret == 0)
        break;
    // Отправка успешна!
}
}
else
{
    // Иначе используем функцию sendto()
    //
    for(i = 0; i < dwCount; i++)
    {
        sprintf (sendbuf, "Test datagram #%d", i);
        ret = sendto(s, sendbuf, strlen(sendbuf), 0,
            (SOCKADDR *)&recipient, sizeof(recipient));
        if (ret == SOCKET_ERROR)
        {
            sprintf(Str, "sendto() failed; %d",
                WSAGetLastError());
            pListBox->AddString(Str);
            break;
        }
        else if (ret == 0)
            break;
        // Отправка успешна!
    }
}
}
closesocket(s);

GlobalFree(sendbuf);
WSACleanup();
return 0;
}

```

Ввод-вывод в Winsock

Основные цели

Изучение данной темы очень важно для написания качественных сетевых приложений. Ранее мы разбирали только основные операции для установления связи и осуществления операций передачи информации. При этом функциональность наших приложений была весьма примитивна: предполагалось, что все участники обмена информацией действуют по заранее написанному жесткому сценарию.

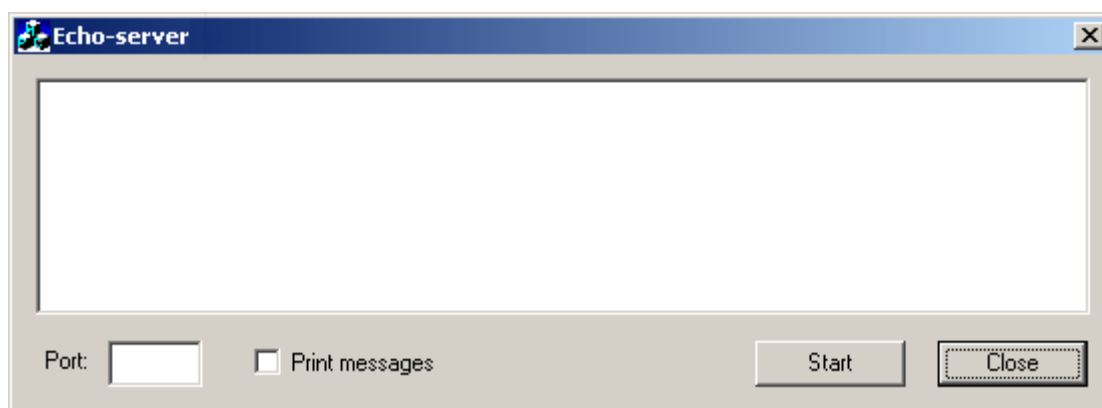
Здесь разбираются примеры организации значительно более гибкого взаимодействия, достигаемого за счет использования различных *моделей ввода-вывода*. При этом приложение в любой момент может одновременно заниматься и приемом, и передачей данных (обычно асинхронно) с несколькими партнерами.

Все модели ввода-вывода рассматриваются на примере разработки соответствующего эхо-сервера (для серверов вопросы эффективной организации обмена информацией особенно важны). Поскольку для тестирования необходимо иметь и клиентское приложение, в качестве такового можно взять клиента из предыдущей темы.

Поскольку целью настоящего пособия является разбор именно практической реализации тех или иных приемов и средств разработки сетевых приложений, теоретические вопросы здесь не рассматриваются, и перед выполнением работы настоятельно рекомендуется освежить в памяти идеологию рассматриваемой модели ввода-вывода, а также соответствующие библиотечные средства. В достаточно компактном виде необходимая информация содержится в учебном пособии [7].

Главное диалоговое окно стартового проекта

Все стартовые проекты практически идентичны и представляют собой диалоговые приложения с главным окном следующего вида (отличаются они друг от друга только надписью в заголовке окна).



На главном диалоговом окне расположены следующие элементы управления.

Список в верхней части окна (идентификатор IDC_LISTBOX) предназначен для вывода сообщений. Поле ввода (идентификатор IDC_PORT) служит для ввода номера порта, на котором мы ждем соединений с клиентом. Флажок (идентификатор IDC_PRINT) разрешает вывод получаемых от клиента сообщений в окне диалога.

Кнопка "Start" (идентификатор IDC_START) предназначена для запуска сервера.

Подключение библиотек, объявление констант и переменных, реализованная функциональность

В стартовом проекте уже подключен файл winsock2.h и библиотека ws2_32.lib.

Также добавлен обработчик изменения состояния флажка IDC_PRINT и настройка его начального состояния в функции CServerDlg::OnInitDialog().

В предложенном стартовом проекте уже создана переменная m_ListBox типа CListBox для управления списком IDC_LISTBOX.

Кроме того, в файле ServerDlg.cpp сделаны следующие объявления констант и глобальных переменных:

```
#define PORT 5150                // Порт по умолчанию
#define DATA_BUFSIZE 8192      // Размер буфера по умолчанию

int  iPort = PORT;              // Порт для прослушивания подключений
bool bPrint = false;           // Выводить ли сообщения клиентов
```

Пример 8.1. Эхо-сервер на основе модели select()

Сервер работает по следующему сценарию. Он создает сокет, привязывает его к локальному IP-интерфейсу и порту, переводит в неблокирующий режим и слушает соединения клиентов. Для проверки наличия обращения клиента используется функция select.

После приема от клиента запроса на соединение новый сокет, созданный функцией accept, переводится в неблокирующий режим. Для проверки возможности осуществления чтения или записи данных также используется функция select.

Открытие проекта

Откройте файл рабочего пространства заготовки проекта Server.dsw, расположенный в директории 08_InputOutput\Select\Starter.

Объявление констант и глобальных переменных

В стартовом проекте уже подключен файл winsock2.h и библиотека ws2_32.lib.

В начале файла ServerDlg.cpp добавим необходимые объявления типов, глобальных переменных и объявления глобальных функций:

```
// ...
bool bPrint = false; // Выводить ли сообщения клиентов

typedef struct _SOCKET_INFORMATION {
    CHAR Buffer[DATA_BUFSIZE];
    WSABUF DataBuf;
    SOCKET Socket;
    OVERLAPPED Overlapped;
    DWORD BytesSEND;
    DWORD BytesRECV;
} SOCKET_INFORMATION, * LPSOCKET_INFORMATION;

DWORD TotalSockets = 0; // Всего сокетов
LPSOCKET_INFORMATION SocketArray[FD_SETSIZE];

HWND hWnd_LB; // Для передачи потокам

BOOL CreateSocketInformation(SOCKET s, CListBox * pLB);
void FreeSocketInformation(DWORD Index, CListBox * pLB);
UINT ListenThread(PVOID lpParam);
```

Здесь ListenThread() – главная функция рабочего потока. Вспомогательные глобальные функции CreateSocketInformation() и FreeSocketInformation() предназначены для работы с массивом SocketArray: размещения в памяти и заполнения его элементов информацией о сокетах, а также их удаления из массива и освобождения памяти.

Определение и реализация обработчиков событий

Из обработчиков нам потребуется только обработчик события, происходящего при нажатии кнопки "Start" (идентификатор IDC_START).

Обработчик OnStart() запоминает дескриптор окна списка для вывода информации, сохраняет введенный пользователем номер порта, затем запускает основной рабочий поток. В завершение он делает неактивной кнопку "Start", чтобы избежать повторного запуска.

```
void CServerDlg::OnStart()
{
    char Str[81];

    hWnd_LB = m_ListBox.m_hWnd; // Для ListenThread
    GetDlgItem(IDC_PORT) -> GetWindowText(Str, sizeof(Str));
    iPort=atoi(Str);
```



```

if (iPort<=0 || iPort>=0x10000)
{
    AfxMessageBox("Incorrect Port number");
    return;
}

AfxBeginThread(ListenThread, NULL);

GetDlgItem(IDC_START)->EnableWindow(false);
}

```

Главная функция рабочего потока

Функция ListenThread() инициализирует библиотеку Winsock, создает сокет для прослушивания, привязывает его к локальному адресу и ждет подключений клиентов. Для проверки наличия запроса клиента на соединение, а также возможности чтения или записи информации используется соответствующая настройка сокетов и далее функция select().

Разберем основные участки кода. Сначала мы описываем необходимые локальные переменные и загружаем библиотеку Winsock:

```

UINT ListenThread(PVOID lpParam)
{
    SOCKET    ListenSocket;
    SOCKET    AcceptSocket;
    SOCKADDR_IN InternetAddr;
    WSADATA    wsaData;
    INT        Ret;
    FD_SET    WriteSet;
    FD_SET    ReadSet;
    DWORD    i;
    DWORD    Total;
    ULONG    NonBlock;
    DWORD    Flags;
    DWORD    SendBytes;
    DWORD    RecvBytes;

    char    Str[200];
    CListBox * pLB =
        (CListBox *) (CListBox::FromHandle(hWnd_LB));

    if ((Ret = WSASStartup(0x0202,&wsaData)) != 0)
    {
        sprintf(Str, "WSASStartup() failed with error %d",
            Ret);
        pLB->AddString(Str);
        WSACleanup();
        return 1;
    }
}

```

Затем создаем сокет, привязываем к интерфейсу и выставяем на прослушивание соединений. Второй параметр функции listen() задает максимальную длину очереди ожидающих соединения клиентов.

```
if ((ListenSocket = WSASocket(AF_INET, SOCK_STREAM, 0,
    NULL, 0, WSA_FLAG_OVERLAPPED)) == INVALID_SOCKET)
{
    sprintf(Str, "WSASocket() failed with error %d",
        WSAGetLastError());
    pLB->AddString(Str);
    return 1;
}

InternetAddr.sin_family = AF_INET;
InternetAddr.sin_addr.s_addr = htonl(INADDR_ANY);
InternetAddr.sin_port = htons(iPort);

if (bind(ListenSocket, (PSOCKADDR) &InternetAddr,
    sizeof(InternetAddr)) == SOCKET_ERROR)
{
    sprintf(Str, "bind() failed with error %d",
        WSAGetLastError());
    pLB->AddString(Str);
    return 1;
}

if (listen(ListenSocket, 5))
{
    sprintf(Str, "listen() failed with error %d",
        WSAGetLastError());
    pLB->AddString(Str);
    return 1;
}
```

Переводим сокет в неблокирующий режим, начинаем бесконечный цикл обслуживания подключений клиентов, чтения и записи данных:

```
NonBlock = 1;
if (ioctlsocket(ListenSocket, FIONBIO, &NonBlock)
    == SOCKET_ERROR)
{
    sprintf(Str, "ioctlsocket() failed with error %d",
        WSAGetLastError());
    pLB->AddString(Str);
    return 1;
}

while(TRUE)
{
```

Разберем последовательность действий внутри этого цикла.

Подготовим множества сокетов для уведомления о возможности операций ввода-вывода. Сначала мы их обнулим, потом добавим к множеству ReadSet прослушивающий сокет, чтобы получать уведомления о попытках подключения клиентов. Затем включим каждый из созданных на данный момент сокетов из массива SocketArray в множество WriteSet или ReadSet в зависимости от его состояния. Если в буфере остались какие-то данные, мы помещаем его в множество WriteSet, в противном случае – в ReadSet.

```
FD_ZERO(&ReadSet);
FD_ZERO(&WriteSet);

FD_SET(ListenSocket, &ReadSet);

for (i = 0; i < TotalSockets; i++)
    if (SocketArray[i]->BytesRECV >
        SocketArray[i]->BytesSEND)
        FD_SET(SocketArray[i]->Socket, &WriteSet);
    else
        FD_SET(SocketArray[i]->Socket, &ReadSet);
```

Вызовем функцию select(), чтобы определить возможность осуществления какой-либо операции:

```
if ((Total = select(0, &ReadSet, &WriteSet, NULL,
                    NULL)) == SOCKET_ERROR)
{
    sprintf(Str, "select() returned with error %d",
            WSAGetLastError());
    pLB->AddString(Str);
    return 1;
}
```

Проверим наличие запросов клиентов на соединение. При наличии запроса примем его и переведем сокет, созданный функцией accept в неблокирующий режим работы:

```
if (FD_ISSET(ListenSocket, &ReadSet))
{
    Total--;
    if ((AcceptSocket = accept(ListenSocket, NULL,
                              NULL)) != INVALID_SOCKET)
    {
        // Перевод сокета в неблокирующий режим
        NonBlock = 1;
    }
}
```

```

        if (ioctlsocket(AcceptSocket, FIONBIO,
                        &NonBlock) == SOCKET_ERROR)
        {
            sprintf(Str,
                    "ioctlsocket() failed with error %d",
                    WSAGetLastError());
            pLB->AddString(Str);
            return 1;
        }
        // Добавление сокет в массив SocketArray
        if (CreateSocketInformation(AcceptSocket, pLB)
            == FALSE)

            return 1;
    }
    else
    {
        if (WSAGetLastError() != WSAEWOULDBLOCK)
        {
            sprintf(Str, "accept() failed with error %d",
                    WSAGetLastError());
            pLB->AddString(Str);
            return 1;
        }
    }
}

```

Далее в цикле для каждого сокета проверяем возможность осуществления операции чтения или записи и, если таковая существует, выполняем соответствующую операцию.

```

for (i = 0; Total > 0 && i < TotalSockets; i++)
{
    LPSOCKET_INFORMATION SocketInfo = SocketArray[i];

    // Если сокет попадает в множество ReadSet,
    // выполняем чтение данных.

    if (FD_ISSET(SocketInfo->Socket, &ReadSet))
    {
        Total--;

        SocketInfo->DataBuf.buf = SocketInfo->Buffer;
        SocketInfo->DataBuf.len = DATA_BUFSIZE;

        Flags = 0;
    }
}

```

```

if (WSARecv(SocketInfo->Socket,
    &(SocketInfo->DataBuf), 1, &RecvBytes,
    &Flags, NULL, NULL) == SOCKET_ERROR)
{
    if (WSAGetLastError() != WSAEWOULDBLOCK)
    {
        sprintf(Str,
            "WSARecv() failed with error %d",
            WSAGetLastError());
        pLB->AddString(Str);

        FreeSocketInformation(i, pLB);
    }
    continue;
}

else // чтение успешно
{
    SocketInfo->BytesRECV = RecvBytes;

    // Если получено 0 байтов, значит клиент
    // закрыл соединение.

    if (RecvBytes == 0)
    {
        FreeSocketInformation(i, pLB);
        continue;
    }

    // Распечатка сообщения, если нужно

    if (bPrint)
    {
        unsigned l = sizeof(Str)-1;
        if (l > RecvBytes)
            l = RecvBytes;
        strncpy(Str, SocketInfo->Buffer, l);
        Str[l]=0;
        pLB->AddString(Str);
    }
}
}

```

```

// Если сокет попадает в множество WriteSet,
// выполняем отправку данных.

if (FD_ISSET(SocketInfo->Socket, &WriteSet))
{
    Total--;

    SocketInfo->DataBuf.buf =
        SocketInfo->Buffer + SocketInfo->BytesSEND;
    SocketInfo->DataBuf.len =
        SocketInfo->BytesRECV - SocketInfo->BytesSEND;

    if (WSASend(SocketInfo->Socket,
        &(SocketInfo->DataBuf), 1, &SendBytes, 0,
        NULL, NULL) == SOCKET_ERROR)
    {
        if (WSAGetLastError() != WSAEWOULDBLOCK)
        {
            sprintf(Str,
                "WSASend() failed with error %d",
                WSAGetLastError());
            pLB->AddString(Str);

            FreeSocketInformation(i, pLB);
        }
        continue;
    }
    else
    {
        SocketInfo->BytesSEND += SendBytes;

        if (SocketInfo->BytesSEND ==
            SocketInfo->BytesRECV)
        {
            SocketInfo->BytesSEND = 0;
            SocketInfo->BytesRECV = 0;
        }
    }
}

return 0;
}

```

Другие глобальные функции

В коде функции ListenThread() присутствуют вызовы вспомогательных глобальных функций CreateSocketInformation() и FreeSocketInformation().

Первая отвечает за добавление нового сокета в массив SocketArray. Она выделяет область памяти для структуры SOCKET_INFORMATION, заполняет ее необходимой информацией и добавляет в массив:

```
BOOL CreateSocketInformation(SOCKET s, CListBox *pLB)
{
    LPSOCKET_INFORMATION SI;
    char Str[81];

    sprintf(Str, "Accepted socket number %d", s);
    pLB->AddString(Str);

    if ((SI = (LPSOCKET_INFORMATION) GlobalAlloc(GPTR,
                                                sizeof(SOCKET_INFORMATION))) == NULL)
    {
        sprintf(Str, "GlobalAlloc() failed with error %d",
                GetLastError());
        pLB->AddString(Str);
        return FALSE;
    }

    // Подготовка структуры SocketInfo для использования.
    SI->Socket = s;
    SI->BytesSEND = 0;
    SI->BytesRECV = 0;

    SocketArray[TotalSockets] = SI;

    TotalSockets++;

    return(TRUE);
}
```

Функция FreeSocketInformation() удаляет эту структуру из массива и освобождает область памяти, отведенную под нее.

```
void FreeSocketInformation(DWORD Index, CListBox *pLB)
{
    LPSOCKET_INFORMATION SI = SocketArray[Index];
    DWORD i;
    char Str[81];

    closesocket(SI->Socket);
}
```

```

    sprintf(Str, "Closing socket number %d", SI->Socket);
    pLB->AddString(Str);

    GlobalFree(SI);

    // Сдвиг массива сокетов
    for (i = Index; i < TotalSockets; i++)
    {
        SocketArray[i] = SocketArray[i + 1];
    }
    TotalSockets--;
}

```

Пример 8.2. Эхо-сервер на основе модели AsyncSelect()

Сервер работает по следующему сценарию. Он создает сокет, привязывает его к локальному IP-интерфейсу и порту и слушает соединения клиентов. Для проверки наличия обращения клиента, а также готовности сокета для отправки и приема данных используется функция `WSAAsyncSelect()`.

Эта функция предполагает использование механизма сообщений Windows, а значит, наличие окна, которому эти сообщения направляются на обработку. Для этой цели мы могли бы использовать имеющееся главное диалоговое окно программы. Однако автору хотелось бы привести пример кода, в котором создается окно специально для этой цели и организуется в отдельном потоке цикл трансляции и обработки сообщений Windows. Созданное рабочее окно остается невидимым и служит только в качестве приемника и обработчика этих сообщений. Предлагаемый исходный код, таким образом, может быть использован и для организации обработки сообщений Windows в консольных приложениях.

Открытие проекта

Откройте файл рабочего пространства заготовки проекта `Server.dsw`, расположенный в директории `08_InputOutput\AsyncSelect\Starter`.

Подключение библиотек, объявление констант и глобальных переменных

В стартовом проекте уже подключен файл `winsock2.h` и библиотека `ws2_32.lib`. Если бы мы разрабатывали консольное приложение, дополнительно следовало бы подключить библиотеки `user32.lib` и `gdi32.lib`. В нашем случае это было сделано мастером `AppWizard` при генерации заготовки приложения.

В начале файла `ServerDlg.cpp` добавим необходимые объявления типов, констант, глобальных переменных и глобальных функций:


```

// ...
bool bPrint = false; // Выводить ли сообщения клиентов

typedef struct _SOCKET_INFORMATION {
    BOOL RecvPosted;
    CHAR Buffer[DATA_BUFSIZE];
    WSABUF DataBuf;
    SOCKET Socket;
    DWORD BytesSEND;
    DWORD BytesRECV;
    _SOCKET_INFORMATION *Next;
} SOCKET_INFORMATION, *LPSOCKET_INFORMATION;

#define WM_SOCKET (WM_USER + 1) // Сообщение о событии

void CreateSocketInformation(SOCKET s, char *Str,
                           CListBox *pLB);
LPSOCKET_INFORMATION GetSocketInformation(SOCKET s,
                                          char *Str, CListBox *pLB);
void FreeSocketInformation(SOCKET s, char *Str,
                          CListBox *pLB);

HWND MakeWorkerWindow(char *Str, CListBox *pLB);
LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg,
                            WPARAM wParam, LPARAM lParam);

LPSOCKET_INFORMATION SocketInfoList;

HWND hWnd_LB; // Для вывода в других потоках

UINT ListenThread(PVOID lpParam);

```

Здесь ListenThread() – главная функция рабочего потока. Функция MakeWorkerWindow() создает упомянутое выше невидимое рабочее окно для приема сообщений Windows, а функция WindowProc() осуществляет обработку этих сообщений.

Объявлены также три вспомогательные глобальные функции CreateSocketInformation(), GetSocketInformation() и FreeSocketInformation(), предназначенные для работы со списком SocketInfoList. Первая функция размещает в памяти структуру SOCKET_INFORMATION, связывает ее с сокетом, предназначенным для обмена данными с клиентом, и добавляет в список. Вторая находит в списке и возвращает элемент (структура SOCKET_INFORMATION), связанный с заданным сокетом. Наконец, последняя функция удаляет элемент из списка и освобождает выделенную память.

Определение и реализация обработчиков событий

Из обработчиков нам потребуется только обработчик события, происходящего при нажатии кнопки "Start" (идентификатор IDC_START).

Обработчик OnStart() запоминает дескриптор окна списка для вывода информации, сохраняет введенный пользователем номер порта, затем запускает основной рабочий поток. В завершение он делает неактивной кнопку "Start", чтобы избежать повторного запуска.

```
void CServerDlg::OnStart()
{
    char Str[81];

    hWnd_LB = m_ListBox.m_hWnd; // Для ListenThread
    GetDlgItem(IDC_PORT) ->GetWindowText(Str, sizeof(Str));
    iPort=atoi(Str);
    if (iPort<=0 || iPort>=0x10000)
    {
        AfxMessageBox("Incorrect Port number");
        return;
    }

    AfxBeginThread(ListenThread, NULL);

    GetDlgItem(IDC_START) ->EnableWindow(false);
}
```

Главная функция рабочего потока

Функция ListenThread() сначала создает окно для приема сообщений Windows и инициализирует библиотеку Winsock. Затем она создает сокет для прослушивания и посредством вызова функции WSAsyncSelect() назначает вновь созданное окно приемником сообщений о событиях на соquete, привязывает его к локальному адресу и выставляет на прослушивание.

После этого организуется цикл по трансляции и диспетчеризации сообщений.

Разберем основные участки кода. Сначала мы описываем необходимые локальные переменные, создаем рабочее окно и загружаем библиотеку Winsock:

```
UINT ListenThread(PVOID lpParam)
{
    MSG msg;
    DWORD Ret;
    SOCKET Listen;
    SOCKADDR_IN InternetAddr;
    HWND Window;
    WSADATA wsaData;
```

```

char Str[200];
CListBox *pLB =
    (CListBox *) (CListBox::FromHandle(hWnd_LB));

if ((Window = MakeWorkerWindow(Str, pLB)) == NULL)
    return 1;

if ((Ret = WSASStartup(0x0202, &wsaData)) != 0)
{
    sprintf(Str, "WSASStartup failed with error %d", Ret);
    pLB->AddString(Str);
    return 1;
}

```

Затем создаем сокет, назначаем окно с дескриптором Window в качестве приемника сообщений, вызванных событиями на сокете, привязываем сокет к интерфейсу и выставляем его на прослушивание соединений. Вторым параметром функции listen() задает максимальную длину очереди ожидающих соединения клиентов.

```

if ((Listen = socket (PF_INET, SOCK_STREAM, 0)) ==
    INVALID_SOCKET)
{
    sprintf(Str, "socket() failed with error %d",
        WSAGetLastError());
    pLB->AddString(Str);
    return 1;
}

WSAAsyncSelect(Listen, Window, WM_SOCKET,
    FD_ACCEPT|FD_CLOSE);

InternetAddr.sin_family = AF_INET;
InternetAddr.sin_addr.s_addr = htonl(INADDR_ANY);
InternetAddr.sin_port = htons(iPort);

if (bind(Listen, (PSOCKADDR) &InternetAddr,
    sizeof(InternetAddr)) == SOCKET_ERROR)
{
    sprintf(Str, "bind() failed with error %d",
        WSAGetLastError());
    pLB->AddString(Str);
    return 1;
}

```

```

if (listen(Listen, 5))
{
    sprintf(Str, "listen() failed with error %d",
            WSAGetLastError());
    pLB->AddString(Str);
    return 1;
}

Наконец, организуем в цикле распределение сообщений:
while(Ret = GetMessage(&msg, NULL, 0, 0))
{
    if (Ret == -1)
    {
        sprintf(Str, "GetMessage() failed with error %d",
                GetLastError());
        pLB->AddString(Str);
        return 1;
    }
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
return 0;
}

```

Функция обработки сообщений

Из всех сообщений Windows функция WindowProc() будет обрабатывать только одно, предназначенное для нее, а именно сообщение WM_SOCKET (передается как значение аргумента uMsg). Прочие сообщения она перенаправляет на обработку системе. При этом через параметр wParam функция получает дескриптор сокета, а через параметр lParam – дополнительную информацию о типе события. Мы предусмотрим четыре возможных значения этого параметра:

- значение FD_ACCEPT соответствует наличию запроса клиента на соединение;
- значения FD_READ и FD_WRITE означают готовность сокета к операциям получения и отправки данных соответственно;
- значение, равное FD_CLOSE, соответствует отключению клиента.

В качестве реакции на первое событие мы создаем сокет для обслуживания клиента, добавляем его в список сокетов и передаем функции WSAAsyncSelect() для того, чтобы получать уведомления о событиях на данном сокете. При получении уведомлений о возможности чтения или записи данных мы выполняем эту операцию, а при отсоединении клиента закрываем сокет и удаляем его из списка.

Далее приводится исходный код, реализующий описанные действия:

```

LRESULT CALLBACK WindowProc (HWND hwnd, UINT uMsg,
                             WPARAM wParam, LPARAM lParam)
{
    SOCKET Accept;
    LPSOCKET_INFORMATION SocketInfo;
    DWORD RecvBytes, SendBytes;
    DWORD Flags;

    char Str[200];
    CListBox *pLB =
        (CListBox *) (CListBox::FromHandle(hwnd_LB));

    if (uMsg == WM_SOCKET)
    {
        if (WSAGETSELECTERROR(lParam))
        {
            sprintf(Str, "Socket failed with error %d",
                    WSAGETSELECTERROR(lParam));
            pLB->AddString(Str);
            FreeSocketInformation(wParam, Str, pLB);
        }
        else
        {
            switch(WSAGETSELECTEVENT(lParam))
            {
                case FD_ACCEPT:
                    if ((Accept = accept(wParam, NULL, NULL)) ==
                        INVALID_SOCKET)
                    {
                        sprintf(Str,
                                "accept() failed with error %d",
                                WSAGetLastError());
                        pLB->AddString(Str);
                        break;
                    }

                    // Создание структуры с информацией о сокете
                    // и занесение ее в список
                    CreateSocketInformation(Accept, Str, pLB);
                    sprintf(Str, "Socket number %d connected",
                            Accept);
                    pLB->AddString(Str);
                    WSAAsyncSelect(Accept, hwnd, WM_SOCKET,
                                   FD_READ|FD_WRITE|FD_CLOSE);
                    break;
            }
        }
    }
}

```

```

case FD_READ:

    SocketInfo =
        GetSocketInformation(wParam, Str, pLB);

    // Читаем только если буфер приема пуст
    if (SocketInfo->BytesRECV != 0)
    {
        SocketInfo->RecvPosted = TRUE;
        return 0;
    }
    else
    {
        SocketInfo->DataBuf.buf =
            SocketInfo->Buffer;
        SocketInfo->DataBuf.len = DATA_BUFSIZE;

        Flags = 0;
        if (WSARecv(SocketInfo->Socket,
                    &(SocketInfo->DataBuf), 1,
                    &RecvBytes,
                    &Flags, NULL, NULL) ==
            SOCKET_ERROR)
        {
            if (WSAGetLastError() !=
                WSAEWOULDBLOCK)
            {
                sprintf(Str, "WSARecv() failed with"
                        " error %d",
                        WSAGetLastError());
                pLB->AddString(Str);
                FreeSocketInformation(wParam,
                                    Str, pLB);
                return 0;
            }
        }
    }
    else // ОК, изменяем счетчик байтов
    {
        SocketInfo->BytesRECV = RecvBytes;
        // Вывод сообщения
        if (bPrint)
        {
            unsigned l = sizeof(Str)-1;
            if (l > RecvBytes)
                l = RecvBytes;

```

```

        strncpy(Str, SocketInfo->Buffer, 1);
        Str[1]=0;
        pLB->AddString(Str);
    }
}
// ЗДЕСЬ НЕТ ОПЕРАТОРА break, поскольку мы
// получили данные. Продолжаем работу и
// начинаем их отправку клиенту.
case FD_WRITE:
    SocketInfo = GetSocketInformation(
        wParam, Str, pLB);
    if (SocketInfo->BytesRECV >
        SocketInfo->BytesSEND)
    {
        SocketInfo->DataBuf.buf =
            SocketInfo->Buffer +
            SocketInfo->BytesSEND;
        SocketInfo->DataBuf.len =
            SocketInfo->BytesRECV -
            SocketInfo->BytesSEND;
        if (WSASend(SocketInfo->Socket,
            &(SocketInfo->DataBuf), 1,
            &SendBytes, 0, NULL, NULL) ==
            SOCKET_ERROR)
        {
            if (WSAGetLastError() !=
                WSAEWOULDBLOCK)
            {
                sprintf(Str, "WSASend() failed"
                    " with error %d",
                    WSAGetLastError());
                pLB->AddString(Str);
                FreeSocketInformation(wParam,
                    Str, pLB);
                return 0;
            }
        }
        else // OK, изменяем счетчик байтов
        {
            SocketInfo->BytesSEND += SendBytes;
        }
    }
}

```

```

        if (SocketInfo->BytesSEND ==
            SocketInfo->BytesRECV)
        {
            SocketInfo->BytesSEND = 0;
            SocketInfo->BytesRECV = 0;

            // Если пришли данные пока мы занимались
            // отправкой, надо послать на сокет
            // уведомление FD_READ.

            if (SocketInfo->RecvPosted == TRUE)
            {
                SocketInfo->RecvPosted = FALSE;
                PostMessage(hwnd, WM_SOCKET,
                    wParam, FD_READ);
            }
        }

        break;

    case FD_CLOSE:

        sprintf(Str, "Closing socket %d", wParam);
        pLB->AddString(Str);
        FreeSocketInformation(wParam, Str, pLB);

        break;
    }
}
return 0;
}

return DefWindowProc(hwnd, uMsg, wParam, lParam);
}

```

Вспомогательные глобальные функции

Добавим в наше приложение еще четыре функции вспомогательного характера.

Функция `CreateSocketInformation()` создает в памяти структуру `SOCKET_INFORMATION`, записывает туда информацию о сокете и его текущем состоянии, после чего добавляет структуру в начало списка `SocketInfoList`.


```

void CreateSocketInformation(SOCKET s, char *Str,
                           CListBox *pLB)
{
    LPSOCKET_INFORMATION SI;

    if ((SI = (LPSOCKET_INFORMATION) GlobalAlloc(GPTR,
        sizeof(SOCKET_INFORMATION))) == NULL)
    {
        sprintf(Str, "GlobalAlloc() failed with error %d",
            GetLastError());
        pLB->AddString(Str);
        return;
    }

    // Подготовка структуры для использования.

    SI->Socket = s;
    SI->RecvPosted = FALSE;
    SI->BytesSEND = 0;
    SI->BytesRECV = 0;

    SI->Next = SocketInfoList;

    SocketInfoList = SI;
}

```

Функция GetSocketInformation() по дескриптору сокета находит его в списке и возвращает указатель на структуру SOCKET_INFORMATION с описанием состояния сокета:

```

LPSOCKET_INFORMATION GetSocketInformation(SOCKET s,
                                           char *Str, CListBox *pLB)
{
    SOCKET_INFORMATION *SI = SocketInfoList;
    while(SI)
    {
        if (SI->Socket == s)
            return SI;
        SI = SI->Next;
    }
    return NULL;
}

```

Функция FreeSocketInformation() находит по дескриптору сокета соответствующий ему элемент в списке SocketInfoList, удаляет его из списка, закрывает сокет и освобождает выделенную под элемент списка память:

```

void FreeSocketInformation(SOCKET s, char *Str,
                          CListBox *pLB)
{
    SOCKET_INFORMATION *SI = SocketInfoList;
    SOCKET_INFORMATION *PrevSI = NULL;

    while(SI)
    {
        if (SI->Socket == s)
        {
            if (PrevSI)
                PrevSI->Next = SI->Next;
            else
                SocketInfoList = SI->Next;

            closesocket(SI->Socket);
            GlobalFree(SI);
            return;
        }
        PrevSI = SI;
        SI = SI->Next;
    }
}

```

Функция MakeWorkerWindow() создает рабочее окно, которому будут направляться на обработку сообщения об изменении состояния сокета, при этом в качестве функции для обработки этих сообщений указывается функция WindowProc(). В предлагаемом варианте аргументы функции MakeWorkerWindow() нужны только для того, чтобы обеспечить ей возможность вывода сообщений на главном диалоговом окне нашего приложения.

```

HWND MakeWorkerWindow(char *Str, CListBox *pLB)
{
    WNDCLASS wndclass;
    CHAR *ProviderClass = "AsyncSelect";
    HWND Window;

    wndclass.style = CS_HREDRAW | CS_VREDRAW;
    wndclass.lpfnWndProc = (WNDPROC)WindowProc;
    wndclass.cbClsExtra = 0;
    wndclass.cbWndExtra = 0;
    wndclass.hInstance = NULL;
    wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);
    wndclass.hbrBackground =
        (HBRUSH) GetStockObject(WHITE_BRUSH);
}

```

```

wndclass.lpszMenuName = NULL;
wndclass.lpszClassName = ProviderClass;

if (RegisterClass(&wndclass) == 0)
{
    sprintf(Str, "RegisterClass() failed with error %d",
            GetLastError());
    pLB->AddString(Str);
    return NULL;
}

// Собственно создание окна.
if ((Window = CreateWindow(
                                ProviderClass,
                                "",
                                WS_OVERLAPPEDWINDOW,
                                CW_USEDEFAULT,
                                CW_USEDEFAULT,
                                CW_USEDEFAULT,
                                CW_USEDEFAULT,
                                NULL,
                                NULL,
                                NULL,
                                NULL)) == NULL)
{
    sprintf(Str, "CreateWindow() failed with error %d",
            GetLastError());
    pLB->AddString(Str);
    return NULL;
}

return Window;
}

```

Пример 8.3. Эхо-сервер на основе модели EventSelect()

Сервер работает по следующему сценарию. Он создает сокет для ожидания подключений клиентов. Сокет выставляется на прослушивание обращений клиентов. Для каждого подключения создается (точнее возвращается функцией `accept()`) сокет для обмена информацией. Сервер считывает присланные данные и просто отправляет их назад клиенту.

Для связи сокета с интересующими нас типами событий (обращение клиента, разрыв связи, возможности выполнения операций ввода и вывода данных) используется функция `WSAEventSelect()`. Для ожидания уведомлений о происходящих событиях мы будем использовать функцию `WSAWaitForMultipleEvents()`. Более подробные пояснения даются далее по тексту.

Данная модель организации работы сервера сама по себе не требует использования многопоточности. В нашем случае мы создаем один рабочий поток с главной функцией ListenThread() только для того, чтобы вызовы функции ожидания WSAWaitForMultipleEvents() не блокировали пользовательский интерфейс приложения.

Открытие проекта

Откройте файл рабочего пространства заготовки проекта Server.dsw, расположенный в директории 08_InputOutput\EventSelect\Starter.

Объявление констант и глобальных переменных

В начале файла ServerDlg.cpp добавим необходимые объявления типов, глобальных переменных и объявления глобальных функций:

```
// ...
bool bPrint = false; // Выводить ли сообщения клиентов

typedef struct _SOCKET_INFORMATION {
    CHAR Buffer[DATA_BUFSIZE];
    WSABUF DataBuf;
    SOCKET Socket;
    DWORD BytesSEND;
    DWORD BytesRECV;
} SOCKET_INFORMATION, *LPSOCKET_INFORMATION;

BOOL CreateSocketInformation(SOCKET s, char *Str,
                           CListBox *pLB);
void FreeSocketInformation(DWORD Event, char *Str,
                           CListBox *pLB);

DWORD EventTotal = 0;
WSAEVENT EventArray[WSA_MAXIMUM_WAIT_EVENTS];
LPSOCKET_INFORMATION SocketArray[WSA_MAXIMUM_WAIT_EVENTS];

HWND hWnd_LB; // Для вывода в других потоках

UINT ListenThread(PVOID lpParam);
```

Здесь ListenThread() – главная функция рабочего потока, в которой и реализуется вся основная функциональность нашего приложения.

Вспомогательные глобальные функции CreateSocketInformation() и FreeSocketInformation() предназначены для работы с массивом сокетов SocketArray и массивом событий EventArray.

Первая функция создает объект типа событие, помещает его в массив EventArray, размещает в памяти структуру SOCKET_INFORMATION,

связывает ее с сокетом, полученным через параметры функции, и добавляет в массив SocketArray. Функция FreeSocketInformation() удаляет указанные элементы из этих массивов и освобождает захваченные ими ресурсы.

Определение и реализация обработчиков событий

Из обработчиков нам потребуется только обработчик события, происходящего при нажатии кнопки "Start" (идентификатор IDC_START).

Обработчик OnStart() запоминает дескриптор окна списка для вывода информации, сохраняет введенный пользователем номер порта, затем запускает основной рабочий поток. В завершение он делает неактивной кнопку "Start", чтобы избежать повторного запуска.

```
void CServerDlg::OnStart()
{
    char Str[81];

    hWnd_LB    = m_ListBox.m_hWnd;    // Для ListenThread
    GetDlgItem(IDC_PORT)->GetWindowText(Str, sizeof(Str));
    iPort=atoi(Str);
    if (iPort<=0 || iPort>=0x10000)
    {
        AfxMessageBox("Incorrect Port number");
        return;
    }

    AfxBeginThread(ListenThread, NULL);

    GetDlgItem(IDC_START)->EnableWindow(false);
}
```

Главная функция рабочего потока

Функция ListenThread() сначала инициализирует библиотеку Winsock. Затем она создает сокет для прослушивания и посредством вызова функции WSAEventSelect() связывает его с интересующими нас сетевыми событиями. Затем она осуществляет привязку сокета и выставляет на прослушивание.

Далее в бесконечном цикле осуществляется обработка происходящих событий в зависимости от типа события.

Разберем основные участки кода. Сначала мы описываем необходимые локальные переменные, создаем рабочее окно и загружаем библиотеку Winsock:

```

UINT ListenThread(PVOID lpParam)
{
    SOCKET Listen;
    SOCKET Accept;
    SOCKADDR_IN InternetAddr;
    DWORD Event;
    WSANETWORKEVENTS NetworkEvents;
    WSADATA wsaData;
    DWORD Ret;
    DWORD Flags;
    DWORD RecvBytes;
    DWORD SendBytes;
    char Str[200];
    CListBox *pLB =
        (CListBox *) (CListBox::FromHandle(hWnd_LB));

    if ((Ret = WSASStartup(0x0202, &wsaData)) != 0)
    {
        sprintf(Str, "WSASStartup() failed with error %d",
            Ret);
        pLB->AddString(Str);
        return 1;
    }

```

Затем создаем сокет, через вызов функции CreateSocketInformation() создаем для него событие и структуру для описания его состояния и записываем в соответствующие массивы. Затем посредством вызова функции WSAEventSelect() связываем его с интересующими нас событиями FD_ACCEPT и FD_CLOSE.

```

    if ((Listen = socket (AF_INET, SOCK_STREAM, 0)) ==
        INVALID_SOCKET)
    {
        sprintf(Str, "socket() failed with error %d",
            WSAGetLastError());
        pLB->AddString(Str);
        return 1;
    }
    CreateSocketInformation(Listen, Str, pLB);
    if (WSAEventSelect(Listen, EventArray[EventTotal - 1],
        FD_ACCEPT|FD_CLOSE) == SOCKET_ERROR)
    {
        sprintf(Str, "WSAEventSelect() failed with error %d",
            WSAGetLastError());
        pLB->AddString(Str);
        return 1;
    }

```

Привязываем сокет к интерфейсу и выставляем его на прослушивание соединений. Второй параметр функции listen() задает максимальную длину очереди ожидающих соединения клиентов.

```
InternetAddr.sin_family = AF_INET;
InternetAddr.sin_addr.s_addr = htonl(INADDR_ANY);
InternetAddr.sin_port = htons(iPort);

if (bind(Listen, (PSOCKADDR) &InternetAddr,
        sizeof(InternetAddr)) == SOCKET_ERROR)
{
    sprintf(Str, "bind() failed with error %d",
            WSAGetLastError());
    pLB->AddString(Str);
    return 1;
}

if (listen(Listen, 5))
{
    sprintf(Str, "listen() failed with error %d",
            WSAGetLastError());
    pLB->AddString(Str);
    return 1;
}
```

Начинаем в цикле обработку происходящих событий. Характер обработки, разумеется, зависит от типа события. Сначалаждемся уведомления о событии посредством функции WSAWaitForMultipleEvents(). После этого информация о событии находится в структуре NetworkEvents.

```
while(TRUE)
{
    // Ждем уведомления о событии на любом сокете
    if ((Event = WSAWaitForMultipleEvents(EventTotal,
            EventArray, FALSE, WSA_INFINITE, FALSE)) ==
            WSA_WAIT_FAILED)
    {
        sprintf(Str, "WSAWaitForMultipleEvents failed"
                " with error %d", WSAGetLastError());
        pLB->AddString(Str);
        return 1;
    }

    if (WSAEnumNetworkEvents(
            SocketArray[Event - WSA_WAIT_EVENT_0] ->Socket,
            EventArray[Event - WSA_WAIT_EVENT_0],
            &NetworkEvents) == SOCKET_ERROR)
```

```

{
    sprintf(Str, "WSAEnumNetworkEvents failed"
              " with error %d", WSAGetLastError());
    pLB->AddString(Str);
    return 1;
}

```

Далее мы рассматриваем возможные типы событий на сокете. Если произошло событие типа `FD_ACCEPT`, мы вызываем функцию установления соединения с клиентом `accept()`. В данной ситуации она не вызовет блокировки и возвратит дескриптор вновь созданного сокета для связи с клиентом. Так же, как и ранее, мы через вызов функции `CreateSocketInformation()` создаем для него объект "событие" структуру для описания состояния сокета, добавляя их в соответствующие массивы. Далее, вызвав функцию `WSAEventSelect()`, мы связываем с сокетом события `FD_READ`, `FD_WRITE` и `FD_CLOSE`.

```

if (NetworkEvents.lNetworkEvents & FD_ACCEPT)
{
    if (NetworkEvents.iErrorCode[FD_ACCEPT_BIT] != 0)
    {
        sprintf(Str, "FD_ACCEPT failed with error %d",
                  NetworkEvents.iErrorCode[FD_ACCEPT_BIT]);
        pLB->AddString(Str);
        break;
    }
    // Прием нового соединения и добавление его
    // в списки сокетов и событий
    if ((Accept = accept(
        SocketArray[Event - WSA_WAIT_EVENT_0]->Socket,
        NULL, NULL)) == INVALID_SOCKET)
    {
        sprintf(Str, "accept() failed with error %d",
                  WSAGetLastError());
        pLB->AddString(Str);
        break;
    }
    // Слишком много сокетов. Закрываем соединение.
    if (EventTotal > WSA_MAXIMUM_WAIT_EVENTS)
    {
        sprintf(Str,
                "Too many connections - closing socket.");
        pLB->AddString(Str);
        closesocket(Accept);
        break;
    }
}

```



```

CreateSocketInformation(Accept, Str, pLB);

if (WSAEventSelect(Accept,
                   EventArray[EventTotal - 1],
                   FD_READ|FD_WRITE|FD_CLOSE) ==
    SOCKET_ERROR)
{
    sprintf(Str, "WSAEventSelect() failed"
              " with error %d", WSAGetLastError());
    pLB->AddString(Str);
    return 1;
}

sprintf(Str, "Socket %d connected", Accept);
pLB->AddString(Str);
}

```

Далее, если произошло событие типа FD_READ или FD_WRITE, выполняем соответственно прием или отправку данных. Детали обработки поясняются комментариями в предлагаемом исходном коде.

```

// Пытаемся читать или писать данные,
// если произошло соответствующее событие

if (NetworkEvents.lNetworkEvents & FD_READ ||
    NetworkEvents.lNetworkEvents & FD_WRITE)
{
    if (NetworkEvents.lNetworkEvents & FD_READ &&
        NetworkEvents.iErrorCode[FD_READ_BIT] != 0)
    {
        sprintf(Str, "FD_READ failed with error %d",
                  NetworkEvents.iErrorCode[FD_READ_BIT]);
        pLB->AddString(Str);
        break;
    }

    if (NetworkEvents.lNetworkEvents & FD_WRITE &&
        NetworkEvents.iErrorCode[FD_WRITE_BIT] != 0)
    {
        sprintf(Str, "FD_WRITE failed with error %d",
                  NetworkEvents.iErrorCode[FD_WRITE_BIT]);
        pLB->AddString(Str);
        break;
    }
}

LPSOCKET_INFORMATION SocketInfo =
    SocketArray[Event - WSA_WAIT_EVENT_0];

```

```

// Читаем данные, если приемный буфер пуст
if (SocketInfo->BytesRECV == 0)
{
    SocketInfo->DataBuf.buf = SocketInfo->Buffer;
    SocketInfo->DataBuf.len = DATA_BUFSIZE;

    Flags = 0;
    if (WSARecv(SocketInfo->Socket,
                &(SocketInfo->DataBuf), 1, &RecvBytes,
                &Flags, NULL, NULL) == SOCKET_ERROR)
    {
        if (WSAGetLastError() != WSAEWOULDBLOCK)
        {
            sprintf(Str, "WSARecv() failed with "
                        " error %d", WSAGetLastError());
            pLB->AddString(Str);
            FreeSocketInformation(
                Event - WSA_WAIT_EVENT_0, Str, pLB);
            return 1;
        }
    }
    else
    {
        SocketInfo->BytesRECV = RecvBytes;
        // Вывод сообщения, если требуется
        if (bPrint)
        {
            unsigned l = sizeof(Str)-1;
            if (l > RecvBytes) l = RecvBytes;
            strncpy(Str, SocketInfo->Buffer, l);
            Str[l]=0;
            pLB->AddString(Str);
        }
    }
}

// Отправка данных, если это возможно
if (SocketInfo->BytesRECV > SocketInfo->BytesSEND)
{
    SocketInfo->DataBuf.buf =
        SocketInfo->Buffer + SocketInfo->BytesSEND;
    SocketInfo->DataBuf.len =
        SocketInfo->BytesRECV - SocketInfo->BytesSEND;
}

```

```

        if (WSASend(SocketInfo->Socket,
                    &(SocketInfo->DataBuf), 1,
                    &SendBytes, 0, NULL, NULL) ==
                    SOCKET_ERROR)
        {
            if (WSAGetLastError() != WSAEWOULDBLOCK)
            {
                sprintf(Str, "WSASend() failed with "
                        "error %d", WSAGetLastError());
                pLB->AddString(Str);
                FreeSocketInformation(
                    Event - WSA_WAIT_EVENT_0, Str, pLB);
                return 1;
            }

            // Произошла ошибка WSAEWOULDBLOCK.
            // Событие FD_WRITE будет отправлено, когда
            // в буфере будет больше свободного места
        }
        else
        {
            SocketInfo->BytesSEND += SendBytes;

            if (SocketInfo->BytesSEND ==
                SocketInfo->BytesRECV)
            {
                SocketInfo->BytesSEND = 0;
                SocketInfo->BytesRECV = 0;
            }
        }
    }
}

```

Наконец, если соединение разорвано, выполняем завершающую очистку:

```

if (NetworkEvents.lNetworkEvents & FD_CLOSE)
{
    if (NetworkEvents.iErrorCode[FD_CLOSE_BIT] != 0)
    {
        sprintf(Str, "FD_CLOSE failed with error %d",
                NetworkEvents.iErrorCode[FD_CLOSE_BIT]);
        pLB->AddString(Str);
        break;
    }
}

```

```

        sprintf(Str, "Closing socket information %d",
            SocketArray[Event - WSA_WAIT_EVENT_0]->Socket);
        pLB->AddString(Str);

        FreeSocketInformation(Event - WSA_WAIT_EVENT_0,
            Str, pLB);
    }
} // while
return 0;
}

```

Вспомогательные глобальные функции

В нашем приложении имеются еще две функции вспомогательного характера.

Функция CreateSocketInformation() через вызов WSACreateEvent() создает объект "событие" и добавляет его в массив EventArray. Она также выделяет память для хранения структуры SOCKET_INFORMATION, записывает туда информацию о сокете и его текущем состоянии, после чего добавляет структуру в массив SocketArray.

```

BOOL CreateSocketInformation(SOCKET s, char *Str,
                            CListBox *pLB)
{
    LPSOCKET_INFORMATION SI;

    if ((EventArray[EventTotal] = WSACreateEvent()) ==
        WSA_INVALID_EVENT)
    {
        sprintf(Str, "WSACreateEvent() failed with error %d",
            WSAGetLastError());
        pLB->AddString(Str);
        return FALSE;
    }

    if ((SI = (LPSOCKET_INFORMATION) GlobalAlloc(GPTR,
        sizeof(SOCKET_INFORMATION))) == NULL)
    {
        sprintf(Str, "GlobalAlloc() failed with error %d",
            GetLastError());
        pLB->AddString(Str);
        return FALSE;
    }

    // Подготовка структуры SocketInfo для использования.
    SI->Socket = s;
}

```

```

SI->BytesSEND = 0;
SI->BytesRECV = 0;

SocketArray[EventTotal] = SI;
EventTotal++;
return (TRUE);
}

```

Функция FreeSocketInformation() получает через первый аргумент номер удаляемого объекта и удаляет соответствующие элементы из массивов EventArray и SocketArray. Она также закрывает сокет и освобождает захваченные ресурсы.

```

void FreeSocketInformation(DWORD Event, char *Str,
                           CListBox *pLB)
{
    LPSOCKET_INFORMATION SI = SocketArray[Event];
    DWORD i;

    closesocket(SI->Socket);
    GlobalFree(SI);
    WSACloseEvent(EventArray[Event]);

    // Сжатие массивов сокетов и событий

    for (i = Event; i < EventTotal; i++)
    {
        EventArray[i] = EventArray[i + 1];
        SocketArray[i] = SocketArray[i + 1];
    }

    EventTotal--;
}

```

Пример 8.4. Эхо-сервер на основе модели перекрытого ввода-вывода

Эта модель интерфейса Winsock более эффективна, чем рассмотренные ранее. С ее помощью приложение может асинхронно выдать несколько запросов ввода-вывода, а затем обслужить принятые запросы по мере их завершения.

Сервер работает следующим образом. Он создает сокет для ожидания подключений клиентов. Сокет выставляется на прослушивание обращений клиентов. Для каждого подключения создается (точнее возвращается функцией ассерт()) сокет для обмена информацией. Сервер считывает присланные данные и просто отправляет их назад клиенту.

Приложение создает два рабочих потока. Первый, с главной функцией ListenThread(), осуществляет прослушивание сети и прием входящих соединений клиентов. Второй, основанный на функции ProcessIO(), осуществляет собственно процесс асинхронного приема и передачи данных. Для синхронизации работы двух этих потоков используются механизмы событий и критических секций.

Открытие проекта

Откройте файл рабочего пространства заготовки проекта Server.dsw, расположенный в директории 08_InputOutput\Overlap\Starter.

Объявление констант и глобальных переменных

В начале файла ServerDlg.cpp добавим необходимые объявления типов, глобальных переменных и объявления глобальных функций:

```
// ...
bool bPrint = false; // Выводить ли сообщения клиентов

typedef struct _SOCKET_INFORMATION {
    CHAR Buffer[DATA_BUFSIZE];
    WSABUF DataBuf;
    SOCKET Socket;
    WSAOVERLAPPED Overlapped;
    DWORD BytesSEND;
    DWORD BytesRECV;
} SOCKET_INFORMATION, *LPSOCKET_INFORMATION;

UINT ProcessIO(PVOID lpParameter);

DWORD EventTotal = 0;
WSAEVENT EventArray[WSA_MAXIMUM_WAIT_EVENTS];
LPSOCKET_INFORMATION SocketArray[WSA_MAXIMUM_WAIT_EVENTS];
CRITICAL_SECTION CriticalSection;

HWND hWnd_LB; // Для вывода в других потоках

UINT ListenThread(PVOID lpParam);
```

Здесь ListenThread() – главная функция рабочего потока, в которой организуется процесс прослушивания сети, прием входящих соединений, а также запуск потока для обслуживания перекрытых операций ввода-вывода с функцией ProcessIO() в качестве главной функции этого потока.

Также здесь описаны массив сокетов SocketArray и массив событий EventArray. Переменная EventTotal предназначена для хранения текущего количества элементов в этих массивах, а критическая секция CriticalSection позволяет синхронизировать работу упомянутых выше потоков с глобальными переменными.

Определение и реализация обработчиков событий

Из обработчиков нам потребуется только обработчик события, происходящего при нажатии кнопки "Start" (идентификатор IDC_START).

Обработчик OnStart() запоминает дескриптор окна списка для вывода информации, сохраняет введенный пользователем номер порта, затем запускает основной рабочий поток. В завершение он делает неактивной кнопку "Start", чтобы избежать повторного запуска.

```
void CServerDlg::OnStart()
{
    char Str[81];

    hWnd_LB    = m_ListBox.m_hWnd;    // Для ListenThread
    GetDlgItem(IDC_PORT)->GetWindowText(Str, sizeof(Str));
    iPort=atoi(Str);
    if (iPort<=0 || iPort>=0x10000)
    {
        AfxMessageBox("Incorrect Port number");
        return;
    }

    AfxBeginThread(ListenThread, NULL);

    GetDlgItem(IDC_START)->EnableWindow(false);
}
```

Главная функция рабочего потока

Функция ListenThread() создает объект "критическая секция" для синхронизации совместного использования глобальных переменных двумя потоками, затем инициализирует библиотеку Winsock.

После этого она создает сокет для прослушивания входящих соединений, осуществляет его привязку к интерфейсу и переводит в режим прослушивания.

Затем создается объект типа событие и записывается в начало массива EventArray. Это событие будет использоваться для уведомления потока обслуживания операций ввода-вывода о необходимости обработать дополнительное событие в массиве событий.

После этого запускается поток с главной функцией ProcessIO(), предназначенный для выполнения перекрытых операций ввода-вывода.

Далее в бесконечном цикле осуществляется прием обращений клиентов и инициирование процесса приема данных от них.

Разберем основные участки кода. Сначала мы описываем необходимые локальные переменные, создаем объект "критическая секция" и загружаем библиотеку Winsock:

```

UINT ListenThread(PVOID lpParam)
{
    WSADATA wsaData;
    SOCKET ListenSocket, AcceptSocket;
    SOCKADDR_IN InternetAddr;
    DWORD Flags;
    DWORD RecvBytes;
    INT Ret;

    char Str[200];
    CListBox * pLB =
        (CListBox *) (CListBox::FromHandle(hWnd_LB));

    InitializeCriticalSection(&CriticalSection);

    if ((Ret = WSASStartup(0x0202, &wsaData)) != 0)
    {
        sprintf(Str, "WSASStartup failed with error %d", Ret);
        pLB->AddString(Str);
        WSACleanup();
        return 1;
    }

    Затем создаем сокет, осуществляем его привязку и выставляем на
    прослушивание входящих соединений.

    if ((ListenSocket = WSASocket(AF_INET, SOCK_STREAM, 0,
        NULL, 0, WSA_FLAG_OVERLAPPED)) == INVALID_SOCKET)
    {
        sprintf(Str, "Failed to get a socket %d",
            WSAGetLastError());
        pLB->AddString(Str);
        return 1;
    }

    InternetAddr.sin_family = AF_INET;
    InternetAddr.sin_addr.s_addr = htonl(INADDR_ANY);
    InternetAddr.sin_port = htons(PORT);

    if (bind(ListenSocket, (PSOCKADDR) &InternetAddr,
        sizeof(InternetAddr)) == SOCKET_ERROR)
    {
        sprintf(Str, "bind() failed with error %d",
            WSAGetLastError());
        pLB->AddString(Str);
        return 1;
    }
}

```



```

if (listen(ListenSocket, 5))
{
    sprintf(Str, "listen() failed with error %d",
              WSAGetLastError());
    pLB->AddString(Str);
    return 1;
}

```

Создаем объект типа событие для сообщения рабочему потоку о необходимости обслуживания дополнительных событий, а также сам этот поток для обслуживания перекрытых операций ввода-вывода:

```

if ((EventArray[0] = WSACreateEvent()) ==
    WSA_INVALID_EVENT)
{
    sprintf(Str, "WSACreateEvent failed with error %d",
              WSAGetLastError());
    pLB->AddString(Str);
    return 1;
}

// Создание потока обслуживания перекрытых операций
if (AfxBeginThread(ProcessIO, NULL) == NULL)
{
    sprintf(Str, "AfxBeginThread failed with error %d",
              GetLastError());
    pLB->AddString(Str);
    return 1;
}

```

```
EventTotal = 1;
```

Начинаем в цикле обработку входящих соединений. Получаем сокет для обмена информацией с клиентом:

```

while(TRUE)
{
    // Принимаем входящее соединение
    if ((AcceptSocket = accept(ListenSocket, NULL, NULL))
        == INVALID_SOCKET)
    {
        sprintf(Str, "accept failed with error %d",
                  WSAGetLastError());
        pLB->AddString(Str);
        return 1;
    }
}

```

```

else // успешное соединение
{
    sprintf(Str, "Socket %d connected", AcceptSocket);
    pLB->AddString(Str);
}

```

Далее создаем и записываем в массив SocketArray структуру с информацией о вновь созданном сокете. Поскольку работа с упомянутым массивом осуществляется в разных потоках, используем для синхронизации критическую секцию.

```

EnterCriticalSection(&CriticalSection);

// Создание структуры, содержащей информацию
// о созданном сокете
if ((SocketArray[EventTotal] =
    (LPSOCKET_INFORMATION) GlobalAlloc(GPTR,
        sizeof(SOCKET_INFORMATION))) == NULL)
{
    sprintf(Str, "GlobalAlloc() failed with error %d",
        GetLastError());
    pLB->AddString(Str);
    return 1;
}

// Заполнение структуры данными о сокете
SocketArray[EventTotal]->Socket = AcceptSocket;
ZeroMemory(&(SocketArray[EventTotal]->Overlapped),
    sizeof(OVERLAPPED));
SocketArray[EventTotal]->BytesSEND = 0;
SocketArray[EventTotal]->BytesRECV = 0;
SocketArray[EventTotal]->DataBuf.len = DATA_BUFSIZE;
SocketArray[EventTotal]->DataBuf.buf =
    SocketArray[EventTotal]->Buffer;

if ((SocketArray[EventTotal]->Overlapped.hEvent =
    EventArray[EventTotal] = WSACreateEvent()) ==
    WSA_INVALID_EVENT)
{
    sprintf(Str, "WSACreateEvent() failed with "
        "error %d", WSAGetLastError());
    pLB->AddString(Str);
    return 1;
}

```

Для вновь установленного соединения иницилируем процесс получения данных и выходим из критической секции.

```

// Посылка запроса WSARecv для начала приема на сожете
Flags = 0;
if (WSARecv(SocketArray[EventTotal]->Socket,
            &(SocketArray[EventTotal]->DataBuf), 1,
            &RecvBytes, &Flags,
            &(SocketArray[EventTotal]->Overlapped), NULL)
            == SOCKET_ERROR)
{
    if (WSAGetLastError() != ERROR_IO_PENDING)
    {
        sprintf(Str, "WSARecv() failed with error %d",
                WSAGetLastError());
        pLB->AddString(Str);
        return 1;
    }
}
EventTotal++;
LeaveCriticalSection(&CriticalSection);

```

Наконец, возбуждаем событие EventArray[0], чтобы рабочий поток обслужил новое событие в массиве EventArray:

```

// Установка события EventArray[0] в свободное
// состояние, чтобы заставить рабочий поток обслужить
// дополнительное событие в массиве событий
if (WSASetEvent(EventArray[0]) == FALSE)
{
    sprintf(Str, "WSASetEvent failed with error %d",
            WSAGetLastError());
    pLB->AddString(Str);
    return 1;
}
}
return 0;
}

```

Главная функция потока обслуживания ввода-вывода

Функция ProcessIO() в цикле обслуживает асинхронные запросы по приему и передаче данных. Разберем этот процесс поэтапно. Сначала опишем и инициализируем необходимые переменные:

```

UINT ProcessIO(PVOID lpParameter)
{
    DWORD Index;
    DWORD Flags;
    LPSOCKET_INFORMATION SI;
    DWORD BytesTransferred;

```

```

DWORD i;
DWORD RecvBytes, SendBytes;

char Str[200];
CListBox *pLB =
    (CListBox *) (CListBox::FromHandle(hWnd_LB));

```

Начинаем цикл по обработке операций ввода-вывода. Определяем индекс произошедшего события и сбрасываем его. Если это событие с нулевым индексом, значит, произошло подключение очередного клиента, изменилось количество событий в массиве EventArray, поэтому мы переходим к следующей итерации цикла, чтобы вызвать заново функцию WSAWaitForMultipleEvents(). В противном случае определяем дескриптор связанного с событием сокета.

```

// Обслуживание асинхронных запросов WSASend, WSARecv
while(TRUE)
{
    if ((Index = WSAWaitForMultipleEvents(EventTotal,
        EventArray, FALSE, WSA_INFINITE, FALSE)) ==
        WSA_WAIT_FAILED)
    {
        sprintf(Str, "WSAWaitForMultipleEvents failed %d",
            WSAGetLastError());
        pLB->AddString(Str);
        return 0;
    }

    // Если произошло событие с индексом 0, значит
    // была попытка подключения к нашему слушающему сокету
    if ((Index - WSA_WAIT_EVENT_0) == 0)
    {
        WSAResetEvent(EventArray[0]);
        continue;
    }

    SI = SocketArray[Index - WSA_WAIT_EVENT_0];
    WSAResetEvent(EventArray[Index - WSA_WAIT_EVENT_0]);
}

```

Если асинхронная операция завершилась неуспешно или партнер разорвал соединение, удаляем ненужные более объекты из массивов SocketArray и EventArray и освобождаем выделенные ресурсы.

```

if (WSAGetOverlappedResult(SI->Socket,
    &(SI->Overlapped), &BytesTransferred, FALSE,
    &Flags) == FALSE || BytesTransferred == 0)
{
    sprintf(Str, "Closing socket %d", SI->Socket);
    pLB->AddString(Str);
}

```

```

if (closesocket(SI->Socket) == SOCKET_ERROR)
{
    sprintf(Str, "closesocket() failed with "
               "error %d", WSAGetLastError());
    pLB->AddString(Str);
}

GlobalFree(SI);
WSACloseEvent(
    EventArray[Index - WSA_WAIT_EVENT_0]);

// Очистка массивов SocketArray и EventArray
// путем удаления дескриптора события и
// структуры описания сокета, если только они
// не в хвосте наших массивов
EnterCriticalSection(&CriticalSection);

if ((Index - WSA_WAIT_EVENT_0) + 1 != EventTotal)
    for (i = Index - WSA_WAIT_EVENT_0;
         i < EventTotal; i++)
    {
        EventArray[i] = EventArray[i + 1];
        SocketArray[i] = SocketArray[i + 1];
    }

EventTotal--;
LeaveCriticalSection(&CriticalSection);

continue;
}

```

Далее в зависимости от состояния полей структуры с информацией о сокете, определяем, требуется отправка или прием данных, и выполняем необходимую операцию. Детали ясны из комментариев.

```

// Проверим, что поле BytesRECV равно нулю.
// Это означает, вызов WSAREcv только что завершен.
// В этом случае заполняем BytesRECV значением
// BytesTransferred, полученным через вызов
// WSAREcv.

if (SI->BytesRECV == 0)
{
    SI->BytesRECV = BytesTransferred;
    SI->BytesSEND = 0;
}

```

```

// Вывод сообщения, если требуется
if (bPrint)
{
    unsigned l = sizeof(Str)-1;
    if (l > BytesTransferred)
        l = BytesTransferred;
    strncpy(Str, SI->Buffer, l);
    Str[l]=0;
    pLB->AddString(Str);
}
}
else
{
    SI->BytesSEND += BytesTransferred;
}

if (SI->BytesRECV > SI->BytesSEND)
{
    // Посылка еще одного запроса WSASend().
    // Поскольку WSASend() не гарантирует отправку
    // полностью, продолжим посылать запросы WSASend(),
    // пока вся информация не будет отправлена.

    ZeroMemory(&(SI->Overlapped),
                sizeof(WSAOVERLAPPED));
    SI->Overlapped.hEvent =
        EventArray[Index - WSA_WAIT_EVENT_0];
    SI->DataBuf.buf = SI->Buffer + SI->BytesSEND;
    SI->DataBuf.len = SI->BytesRECV - SI->BytesSEND;
    if (WSASend(SI->Socket, &(SI->DataBuf), 1,
                &SendBytes, 0, &(SI->Overlapped), NULL)
        == SOCKET_ERROR)
    {
        if (WSAGetLastError() != ERROR_IO_PENDING)
        {
            sprintf(Str, "WSASend() failed with "
                        "error %d", WSAGetLastError());
            pLB->AddString(Str);
            return 0;
        }
    }
}
else
{
    SI->BytesRECV = 0;
}

```

```

// Больше посылать нечего.
// Поэтому выдаем запрос WSARecv().

Flags = 0;
ZeroMemory(&(SI->Overlapped),
           sizeof(WSAOVERLAPPED));
SI->Overlapped.hEvent =
    EventArray[Index - WSA_WAIT_EVENT_0];
SI->DataBuf.len = DATA_BUFSIZE;
SI->DataBuf.buf = SI->Buffer;

if (WSARecv(SI->Socket, &(SI->DataBuf), 1,
            &RecvBytes, &Flags, &(SI->Overlapped), NULL)
    == SOCKET_ERROR)
{
    if (WSAGetLastError() != ERROR_IO_PENDING)
    {
        sprintf(Str, "WSARecv() failed with "
                    "error %d", WSAGetLastError());
        pLB->AddString(Str);
        return 0;
    }
}
}
}
}
}
}

```

Пример 8.5. Эхо-сервер на основе модели перекрытого ввода-вывода с использованием процедур завершения

Процедуры завершения – это еще один способ управлять завершенными запросами перекрытого ввода-вывода. Они представляют собой функции, которые можно передать запросу перекрытого ввода-вывода и которые вызываются системой по его завершении.

Сервер работает по следующему сценарию. Он создает сокет для ожидания подключений клиентов. Сокет выставляется на прослушивание обращений клиентов. Для каждого подключения создается (точнее возвращается функцией `ассерт()`) сокет для обмена информацией. Сервер считывает присланные данные и просто отправляет их назад клиенту.

Приложение использует два рабочих потока. Первый, с главной функцией `ListenThread()`, осуществляет прослушивание сети и прием входящих соединений клиентов.

Второй, основанный на функции `WorkerThread()`, осуществляет собственно процесс асинхронного приема и передачи данных, основанного

на использовании функции обратного вызова (callback) WorkerRoutine(). Для синхронизации работы двух этих потоков используются события и глобальные переменные.

Открытие проекта

Откройте файл рабочего пространства заготовки проекта Server.dsw, расположенный в директории 08_InputOutput\Callback\Starter.

Объявление констант и глобальных переменных

В начале файла ServerDlg.cpp добавим необходимые объявления типов, глобальных переменных и объявления глобальных функций:

```
// ...
bool bPrint = false; // Выводить ли сообщения клиентов

typedef struct _SOCKET_INFORMATION {
    OVERLAPPED Overlapped;
    SOCKET Socket;
    CHAR Buffer[DATA_BUFSIZE];
    WSABUF DataBuf;
    DWORD BytesSEND;
    DWORD BytesRECV;
} SOCKET_INFORMATION, * LPSOCKET_INFORMATION;

void CALLBACK WorkerRoutine(DWORD Error,
    DWORD BytesTransferred, LPWSAOVERLAPPED Overlapped,
    DWORD InFlags);

UINT WorkerThread(LPVOID lpParameter);

SOCKET AcceptSocket;
HWND hWnd_LB; // Для вывода в других потоках

UINT ListenThread(PVOID lpParam);

bool fAcceptedSocketProcessed;
```

Здесь ListenThread() – главная функция рабочего потока, в которой организуется процесс прослушивания сети, прием входящих соединений, а также запуск рабочего потока для обслуживания перекрытых операций ввода-вывода с функцией WorkerThread() в качестве главной функции этого потока. В качестве параметра эта функция получает указатель на объект типа событие, который используется для сообщения рабочему потоку о том, что успешно было установлено соединение с клиентом.

Переменная AcceptSocket содержит в этом случае дескриптор сокета, созданного функцией accept() при подключении клиента. Функция WorkerRoutine() играет роль процедуры завершения для операций перекрытого ввода-вывода. Установка флага fAcceptedSocketProcessed означает, что сокет, созданный предыдущим вызовом функции accept(), уже запомнен в рабочем потоке с главной функцией WorkerThread(), и можно принимать новые соединения от клиентов.

Определение и реализация обработчиков событий

Из обработчиков нам потребуется только обработчик события, происходящего при нажатии кнопки "Start" (идентификатор IDC_START).

Обработчик OnStart() запоминает дескриптор окна списка для вывода информации, сохраняет введенный пользователем номер порта, затем запускает основной рабочий поток. В завершение он делает неактивной кнопку "Start", чтобы избежать повторного запуска.

```
void CServerDlg::OnStart()
{
    char Str[81];

    hWnd_LB = m_ListBox.m_hWnd; // Для вывода в других потоках
    GetDlgItem(IDC_PORT) ->GetWindowText(Str, sizeof(Str));
    iPort=atoi(Str);
    if (iPort<=0 || iPort>=0x10000)
    {
        AfxMessageBox("Incorrect Port number");
        return;
    }

    AfxBeginThread(ListenThread, NULL);

    GetDlgItem(IDC_START) ->EnableWindow(false);
}
```

Главная функция рабочего потока

Функция ListenThread() инициализирует библиотеку Winsock. После этого она создает сокет для прослушивания входящих соединений, осуществляет его привязку к интерфейсу и переводит в режим прослушивания.

Затем создается объект типа событие, который будет использоваться для уведомления потока обслуживания операций ввода-вывода о наличии вновь подключившегося клиента и необходимости начать процесс его обслуживания.

После этого запускается поток с главной функцией WorkerThread(), предназначенный для выполнения перекрытых операций ввода-вывода.

Далее в бесконечном цикле осуществляется прием обращений клиентов и инициирование процесса их обработки путем установки события AcceptEvent.

Разберем основные участки кода. Сначала мы описываем необходимые локальные переменные и загружаем библиотеку Winsock:

```
UINT ListenThread(PVOID lpParam)
{
    WSADATA wsaData;
    SOCKET ListenSocket;
    SOCKADDR_IN InternetAddr;
    INT Ret;
    WSAEVENT AcceptEvent;

    char Str[200];
    CListBox *pLB =
        (CListBox *) (CListBox::FromHandle(hWnd_LB));

    if ((Ret = WSASStartup(0x0202, &wsaData)) != 0)
    {
        sprintf(Str, "WSASStartup failed with error %d", Ret);
        pLB->AddString(Str);
        WSACleanup();
        return 1;
    }

    Затем создаем сокет, осуществляем его привязку и выставляем на
    прослушивание входящих соединений.

    if ((ListenSocket = WSASocket(AF_INET, SOCK_STREAM, 0,
        NULL, 0, WSA_FLAG_OVERLAPPED)) == INVALID_SOCKET)
    {
        sprintf(Str, "Failed to get a socket %d",
            WSAGetLastError());
        pLB->AddString(Str);
        return 1;
    }

    InternetAddr.sin_family = AF_INET;
    InternetAddr.sin_addr.s_addr = htonl(INADDR_ANY);
    InternetAddr.sin_port = htons(iPort);

    if (bind(ListenSocket, (PSOCKADDR) &InternetAddr,
        sizeof(InternetAddr)) == SOCKET_ERROR)
    {
        sprintf(Str, "bind() failed with error %d",
            WSAGetLastError());
    }
}
```

```

    pLB->AddString(Str);
    return 1;
}

if (listen(ListenSocket, 5))
{
    sprintf(Str, "listen() failed with error %d",
            WSAGetLastError());
    pLB->AddString(Str);
    return 1;
}

```

Создаем объект типа событие для сообщения рабочему потоку о необходимости обслуживания вновь установленного соединения, а также сам этот поток для выполнения операций ввода-вывода. Дескриптор события AcceptEvent передается потоку через аргумент его главной функции.

```

if ((AcceptEvent = WSACreateEvent()) ==
    WSA_INVALID_EVENT)
{
    sprintf(Str, "WSACreateEvent() failed with error %d",
            WSAGetLastError());
    pLB->AddString(Str);
    return 1;
}

// Создание рабочего потока для обслуживания
// завершенных запросов ввода/вывода

if (AfxBeginThread(WorkerThread, (LPVOID) AcceptEvent) ==
    NULL)
{
    sprintf(Str, "AfxBeginThread failed with error %d",
            GetLastError());
    pLB->AddString(Str);
    return 1;
}

```

Наконец, организуем цикл для обработки входящих соединений. Если предыдущее соединение еще не обработано потоком выполнения операций ввода-вывода, мы ждем, пока это будет сделано. После этого выполняем блокирующий вызов функции accept() и устанавливаем в свободное состояние событие AcceptSocket().

```

fAcceptedSocketProcessed = true;

while(TRUE)
{
    // Обработано ли предыдущее соединение
    if (!fAcceptedSocketProcessed)
    {
        Sleep(0); // Переключение на другой поток
        continue;
    }

    AcceptSocket = accept(ListenSocket, NULL, NULL);

    if (WSASetEvent(AcceptEvent) == FALSE)
    {
        sprintf(Str, "WSASetEvent failed with error %d",
                WSAGetLastError());
        pLB->AddString(Str);
        return 1;
    }
}
return 0;
}

```

Главная функция потока обслуживания ввода-вывода

Функция WorkerThread() предназначена для инициирования процесса обслуживания вновь принятого клиента. Разберем этот процесс поэтапно. Сначала опишем и инициализируем необходимые переменные:

```

UINT WorkerThread(LPVOID lpParameter)
{
    DWORD Flags;
    LPSOCKET_INFORMATION SocketInfo;
    WSAEVENT EventArray[1];
    DWORD Index;
    DWORD RecvBytes;

    char Str[200];
    CListBox *pLB =
        (CListBox *) (CListBox::FromHandle(hWnd_LB));

    // Сохранение события асепт в массиве
    EventArray[0] = (WSAEVENT) lpParameter;

```

Далее организуется цикл для инициирования процесса обслуживания вновь принятых клиентов. Сначала дожидаемся установки события AcceptSocket (его дескриптор хранится в нулевом элементе массива EventArray).

```
while(TRUE)
{
    // Ждем, пока сработает событие для accept()
    while(TRUE)
    {
        Index = WSAWaitForMultipleEvents(1, EventArray,
                                         FALSE, WSA_INFINITE, TRUE);

        if (Index == WSA_WAIT_FAILED)
        {
            sprintf(Str, "WSAWaitForMultipleEvents failed "
                        "with error %d", WSAGetLastError());
            pLB->AddString(Str);
            return FALSE;
        }

        if (Index != WAIT_IO_COMPLETION)
        {
            // Событие для accept() произошло -
            // прерываем цикл ожидания
            break;
        }
    }
}
```

Далее мы сбрасываем это событие, создаем и заполняем структуру с необходимой информацией о сокете. Существенно, что первое поле этой структуры – поле Overlapped, так как это позволяет функции обратного вызова WorkerRoutine() получить доступ ко всей структуре, соответствующим образом преобразовав указатель на это поле, полученный как один из аргументов. Для вновь установленного соединения делается первый вызов WSARecv() с указанием функции WorkerRoutine() в качестве функции обратного вызова. Дальнейшая работа с клиентом осуществляется уже этой функцией.

```
WSAResetEvent(EventArray[Index - WSA_WAIT_EVENT_0]);

// Создаем структуру с информацией о сокете
// и связываем ее с сокетом, созданным через
// вызов accept()
```

```

if ((SocketInfo = (LPSOCKET_INFORMATION)
    GlobalAlloc(GPTR, sizeof(SOCKET_INFORMATION)))
    == NULL)
{
    sprintf(Str, "GlobalAlloc() failed with error %d",
        GetLastError());
    pLB->AddString(Str);
    return FALSE;
}

// Заполняем структуру информацией о сокете

SocketInfo->Socket = AcceptSocket;
ZeroMemory(&(SocketInfo->Overlapped),
    sizeof(WSAOVERLAPPED));
SocketInfo->BytesSEND = 0;
SocketInfo->BytesRECV = 0;
SocketInfo->DataBuf.len = DATA_BUFSIZE;
SocketInfo->DataBuf.buf = SocketInfo->Buffer;

Flags = 0;
if (WSARecv(SocketInfo->Socket,
    &(SocketInfo->DataBuf), 1, &RecvBytes, &Flags,
    &(SocketInfo->Overlapped), WorkerRoutine) ==
    SOCKET_ERROR)
{
    if (WSAGetLastError() != WSA_IO_PENDING)
    {
        sprintf(Str, "WSARecv() failed with error %d",
            GetLastError());
        pLB->AddString(Str);
        return FALSE;
    }
}

sprintf(Str, "Socket %d connected", AcceptSocket);
pLB->AddString(Str);

// Информация из переменной AcceptSocket запомнена
fAcceptedSocketProcessed = true;
}
return TRUE;
}

```

Функция обратного вызова для запросов ввода-вывода

Функция обратного вызова WorkerRoutine() получает управление по завершении очередной операции приема или отправки данных. В нашем примере она все принятые данные отправляет назад клиенту и снова начинает процесс приема.

Ниже поэтапно разбирается ее код. Сначала описываются и инициализируются необходимые переменные. Далее мы получаем ссылку на структуру с описанием сокета для дальнейшей работы с ним. При этом мы пользуемся тем, что структура перекрытого ввода-вывода, полученная функцией WorkerRoutine() через третий параметр, является первым полем структуры SOCKET_INFORMATION, а значит, их адреса совпадают.

```
void CALLBACK WorkerRoutine(DWORD Error,
                             DWORD BytesTransferred, LPWSAOVERLAPPED Overlapped,
                             DWORD InFlags)
{
    DWORD SendBytes, RecvBytes;
    DWORD Flags;
    char Str[200];
    static CListBox *pLB =
        (CListBox *) (CListBox::FromHandle(hWnd_LB));
    // Ссылка на структуру WSAOVERLAPPED как
    // на структуру SOCKET_INFORMATION
    LPSOCKET_INFORMATION SI =
        (LPSOCKET_INFORMATION) Overlapped;
```

В случае ошибки операции ввода-вывода либо разрыва соединения со стороны клиента закрываем сокет и освобождаем ресурсы.

```
    if (Error != 0)
    {
        sprintf(Str, "I/O operation failed with error %d",
                Error);
        pLB->AddString(Str);
    }
    if (BytesTransferred == 0)
    {
        sprintf(Str, "Closing socket %d", SI->Socket);
        pLB->AddString(Str);
    }
    if (Error != 0 || BytesTransferred == 0)
    {
        closesocket(SI->Socket);
        GlobalFree(SI);
        return;
    }
}
```

Проверяем, какой процесс завершился: прием данных или отправка. В зависимости от этого корректируем содержимое полей структуры SOCKET_INFORMATION с информацией о количестве принятых и отправленных байтов. Если установлена соответствующая опция, полученная информация выводится на экран.

```
// Проверим, что поле BytesRECV равно нулю.
// Это означает, вызов WSAREcv только что завершен.
// В этом случае заполняем BytesRECV значением
// BytesTransferred, полученным через вызов WSAREcv.

if (SI->BytesRECV == 0)
{
    SI->BytesRECV = BytesTransferred;
    SI->BytesSEND = 0;

    // Печать сообщения, если требуется
    if (bPrint)
    {
        unsigned l = sizeof(Str)-1;
        if (l > BytesTransferred) l = BytesTransferred;
        strncpy(Str, SI->Buffer, l);
        Str[l]=0;
        pLB->AddString(Str);
    }
}
else
{
    SI->BytesSEND += BytesTransferred;
}
```

Если есть данные, которые еще не отправлены обратно клиенту, делаем это через асинхронный вызов функции WSASend(). В качестве функции обратного вызова указываем WorkerRoutine().

```
if (SI->BytesRECV > SI->BytesSEND)
{
    // Посылка еще одного запроса WSASend().
    // Поскольку WSASend() не гарантирует отправку
    // полностью, продолжим посылать запросы WSASend(),
    // пока вся информация не будет отправлена.

    ZeroMemory(&(SI->Overlapped), sizeof(WSAOVERLAPPED));

    SI->DataBuf.buf = SI->Buffer + SI->BytesSEND;
    SI->DataBuf.len = SI->BytesRECV - SI->BytesSEND;
```



```

if (WSASend(SI->Socket, &(SI->DataBuf), 1, &SendBytes,
            0, &(SI->Overlapped), WorkerRoutine) ==
            SOCKET_ERROR)
{
    if (WSAGetLastError() != WSA_IO_PENDING)
    {
        sprintf(Str, "WSASend() failed with error %d",
                WSAGetLastError());
        pLB->AddString(Str);
        return;
    }
}
}

```

В противном случае начинаем процесс приема очередной порции данных от клиента.

```

else
{
    SI->BytesRECV = 0;

    // Больше посылать нечего.
    // Поэтому выдаем запрос WSARecv().

    Flags = 0;
    ZeroMemory(&(SI->Overlapped), sizeof(WSAOVERLAPPED));

    SI->DataBuf.len = DATA_BUFSIZE;
    SI->DataBuf.buf = SI->Buffer;

    if (WSARecv(SI->Socket, &(SI->DataBuf), 1, &RecvBytes,
                &Flags, &(SI->Overlapped), WorkerRoutine)
        == SOCKET_ERROR)
    {
        if (WSAGetLastError() != WSA_IO_PENDING )
        {
            sprintf(Str, "WSARecv() failed with error %d",
                    WSAGetLastError());
            pLB->AddString(Str);
            return;
        }
    }
}
}
}

```

Пример 8.6. Эхо-сервер на основе модели портов завершения

Данная модель организации ввода-вывода является наиболее сложной из рассматриваемых. В то же время она позволяет достичь наибольшей производительности, если требуется одновременно управлять сотнями и тысячами сокетов одновременно, причем требуется обеспечить хорошую масштабируемость при добавлении новых процессоров в систему. Модель предназначена для работы только под управлением Windows NT, Windows 2000 и, разумеется, более свежих версий этой операционной системы.

Сервер работает по следующему сценарию. В отдельном потоке с главной функцией ListenThread() он создает сокет для ожидания подключений клиентов и организует прослушивание сети. В этом же потоке создается объект типа порт завершения, который используется в процессе обслуживания клиентов. Каждый сокет для обмена данных (созданный функцией приема соединения) привязывается к нему. Здесь же запускается необходимое количество рабочих потоков (главная функция ServerWorkerThread()) по обслуживанию завершенных операций ввода-вывода. Эта функция считывает присланные данные и просто отправляет их назад клиенту.

Открытие проекта

Откройте файл рабочего пространства заготовки проекта Server.dsw, расположенный в директории 08_InputOutput\IOcimplt\Starter.

Объявление констант и глобальных переменных

В начале файла ServerDlg.cpp добавим необходимые объявления типов, глобальных переменных, типов данных и объявления глобальных функций:

```
// ...
bool bPrint = false; // Выводить ли сообщения клиентов

typedef struct
{
    OVERLAPPED Overlapped;
    WSABUF DataBuf;
    CHAR Buffer[DATA_BUFSIZE];
    DWORD BytesSEND;
    DWORD BytesRECV;
} PER_IO_OPERATION_DATA, * LPPER_IO_OPERATION_DATA;

typedef struct
{
    SOCKET Socket;
} PER_HANDLE_DATA, *LPPER_HANDLE_DATA;
```

```
UINT    ServerWorkerThread(LPVOID CompletionPortID);
```

```
HWND    hWnd_LB;    // Для вывода в других потоках
```

```
UINT    ListenThread(PVOID lpParam);
```

Здесь ListenThread() – главная функция рабочего потока, в которой создается объект "порт завершения", запускается необходимое количество рабочих потоков обслуживания ввода-вывода, а также организуется процесс прослушивания сети и прием входящих соединений. Сокет для каждого принятого соединения привязывается к порту завершения, и на нем инициируется процесс приема данных от клиента.

ServerWorkerThread() – главная функция потока, обслуживающего завершённые операции ввода-вывода.

Структуры предназначены для хранения необходимой информации о сокете и операции ввода-вывода. Набор их полей зависит от алгоритма работы сервера. Например, обычная практика – использование в структуре PER_IO_OPERATION_DATA поля, содержащего информацию о типе выполняемой операции (в нашем примере мы обходимся без него).

Определение и реализация обработчиков событий

Из обработчиков нам потребуется только обработчик события, происходящего при нажатии кнопки "Start" (идентификатор IDC_START).

Обработчик OnStart() запоминает дескриптор окна списка для вывода информации, сохраняет введенный пользователем номер порта, затем запускает основной рабочий поток. В завершение он делает неактивной кнопку "Start", чтобы избежать повторного запуска.

```
void CServerDlg::OnStart()
{
    char Str[81];

    hWnd_LB = m_ListBox.m_hWnd; // Для вывода в других потоках
    GetDlgItem(IDC_PORT) ->GetWindowText(Str, sizeof(Str));
    iPort=atoi(Str);
    if (iPort<=0 || iPort>=0x10000)
    {
        AfxMessageBox("Incorrect Port number");
        return;
    }

    AfxBeginThread(ListenThread, NULL);

    GetDlgItem(IDC_START) ->EnableWindow(false);
}
```

Главная функция рабочего потока

Функция ListenThread() инициализирует библиотеку Winsock, создает порт завершения и запускает необходимое количество рабочих потоков для обслуживания завершенных операций ввода-вывода. После этого она создает сокет для прослушивания входящих соединений, осуществляет его привязку к интерфейсу и переводит в режим прослушивания.

Далее в бесконечном цикле осуществляется прием обращений клиентов и инициирование процесса их обслуживания.

Разберем основные участки кода. Сначала мы описываем необходимые локальные переменные и загружаем библиотеку Winsock:

```
UINT ListenThread(PVOID lpParam)
{
    SOCKADDR_IN InternetAddr;
    SOCKET Listen;
    SOCKET Accept;
    HANDLE CompletionPort;
    SYSTEM_INFO SystemInfo;
    LPPER_HANDLE_DATA PerHandleData;
    LPPER_IO_OPERATION_DATA PerIoData;
    int i;
    DWORD RecvBytes;
    DWORD Flags;
    WSADATA wsaData;
    DWORD Ret;

    char Str[200];
    CListBox *pLB =
        (CListBox *) (CListBox::FromHandle(hWnd_LB));

    if ((Ret = WSASStartup(0x0202, &wsaData)) != 0)
    {
        sprintf(Str, "WSASStartup failed with error %d", Ret);
        pLB->AddString(Str);
        return 1;
    }
}
```

Далее создаем порт завершения и запускаем необходимое количество рабочих потоков (в нашем примере – по два для каждого процессора в системе). Главной функции этих потоков ServerWorkerThread() в качестве аргумента передается дескриптор созданного порта завершения.

```
// Создание порта завершения
if ((CompletionPort = CreateIoCompletionPort(
    INVALID_HANDLE_VALUE, NULL, 0, 0)) == NULL)
{
    sprintf(Str, "CreateIoCompletionPort failed with "
        "error: %d", GetLastError());
}
```

```

    pLB->AddString(Str);
    return 1;
}

// Определяем количество процессоров в системе
GetSystemInfo(&SystemInfo);

// Создаем рабочие потоки,
// по два для каждого процессора в системе
for(i = 0; i < SystemInfo.dwNumberOfProcessors * 2; i++)
{
    // Создание серверного рабочего потока
    // с передачей ему порта завершения
    if (AfxBeginThread(ServerWorkerThread, CompletionPort)
        == NULL)
    {
        sprintf(Str, "AfxBeginThread() failed with "
            "error %d", GetLastError());
        pLB->AddString(Str);
        return 1;
    }
}

```

Затем создаем сокет, осуществляем его привязку и выставляем на прослушивание входящих соединений.

```

// Создание сокета для прослушивания сети
if ((Listen = WSASocket(AF_INET, SOCK_STREAM, 0, NULL, 0,
    WSA_FLAG_OVERLAPPED)) == INVALID_SOCKET)
{
    sprintf(Str, "WSASocket() failed with error %d",
        WSAGetLastError());
    pLB->AddString(Str);
    return 1;
}

InternetAddr.sin_family = AF_INET;
InternetAddr.sin_addr.s_addr = htonl(INADDR_ANY);
InternetAddr.sin_port = htons(iPort);
if (bind(Listen, (PSOCKADDR) &InternetAddr,
    sizeof(InternetAddr)) == SOCKET_ERROR)
{
    sprintf(Str, "bind() failed with error %d",
        WSAGetLastError());
    pLB->AddString(Str);
    return 1;
}

```

```
// Перевод сокета в режим прослушивания
if (listen(Listen, 5) == SOCKET_ERROR)
{
    sprintf(Str, "listen() failed with error %d",
            WSAGetLastError());
    pLB->AddString(Str);
    return 1;
}
```

Наконец, организуем цикл для обработки входящих соединений. Внутри цикла принимаем очередное подключение клиента, создаем структуру описания соответствующего сокета, привязываем его к порту завершения и инициализируем первую операцию по обмену данными, а именно, прием информации от клиента (вызов WSARecv()).

```
// Прием соединений и их привязка к порту завершения
while(TRUE)
{
    if ((Accept = WSAAccept(Listen, NULL, NULL, NULL, 0))
        == SOCKET_ERROR)
    {
        sprintf(Str, "WSAAccept() failed with error %d",
                WSAGetLastError());
        pLB->AddString(Str);
        return 1;
    }

    // Создание структуры с информацией о сокете
    if ((PerHandleData = (LPPER_HANDLE_DATA)
        GlobalAlloc(GPTR, sizeof(PER_HANDLE_DATA)))
        == NULL)
    {
        sprintf(Str, "GlobalAlloc() failed with error %d",
                GetLastError());
        pLB->AddString(Str);
        return 1;
    }

    // Привязка сокета, возвращенного функцией WSAAccept,
    // к порту завершения
    sprintf(Str, "Socket number %d connected", Accept);
    pLB->AddString(Str);
    PerHandleData->Socket = Accept;
}
```

```

if (CreateIoCompletionPort((HANDLE) Accept,
                           CompletionPort, (DWORD) PerHandleData, 0)
    == NULL)
{
    sprintf(Str, "CreateIoCompletionPort failed "
               "with error %d", GetLastError());
    pLB->AddString(Str);
    return 1;
}

// Создание структуры данных описателя сокета,
// которая будет связана с вызовом WSAREcv
if ((PerIoData = (LPPER_IO_OPERATION_DATA)
    GlobalAlloc(GPTR, sizeof(PER_IO_OPERATION_DATA)))
    == NULL)
{
    sprintf(Str, "GlobalAlloc() failed with error %d",
            GetLastError());
    pLB->AddString(Str);
    return 1;
}

ZeroMemory(&(PerIoData->Overlapped),
           sizeof(OVERLAPPED));
PerIoData->BytesSEND = 0;
PerIoData->BytesRECV = 0;
PerIoData->DataBuf.len = DATA_BUFSIZE;
PerIoData->DataBuf.buf = PerIoData->Buffer;

Flags = 0;
if (WSAREcv(Accept, &(PerIoData->DataBuf), 1,
            &RecvBytes, &Flags, &(PerIoData->Overlapped),
            NULL) == SOCKET_ERROR)
{
    if (WSAGetLastError() != ERROR_IO_PENDING)
    {
        sprintf(Str, "WSAREcv() failed with error %d",
                WSAGetLastError());
        pLB->AddString(Str);
        return 1;
    }
}
}
return 0;
}

```

Главная функция потока обслуживания ввода-вывода

Функция `ServerWorkerThread()` получает в качестве параметра дескриптор порта завершения, используемого в процессе обмена данными с клиентом. Далее в цикле осуществляется прием данных и отправка их обратно клиенту.

Разберем этот процесс поэтапно. Сначала опишем и инициализируем необходимые переменные:

```
UINT ServerWorkerThread(LPVOID CompletionPortID)
{
    HANDLE CompletionPort = (HANDLE) CompletionPortID;
    DWORD BytesTransferred;
    LPPER_HANDLE_DATA PerHandleData;
    LPPER_IO_OPERATION_DATA PerIoData;
    DWORD SendBytes, RecvBytes;
    DWORD Flags;

    char Str[200];
    CListBox *pLB =
        (CListBox *) (CListBox::FromHandle(hWnd_LB));
```

Далее организуется цикл для обслуживания запросов ввода-вывода и ожидаем завершения какой-либо операции ввода-вывода. В случае ошибки или разрыва соединения прекращаем обслуживание, закрываем сокет и освобождаем ресурсы.

```
while(TRUE)
{
    // Ожидание завершения ввода-вывода на любом
    // из сокетов, связанных с портом завершения
    if (GetQueuedCompletionStatus(CompletionPort,
        &BytesTransferred, (LPDWORD)&PerHandleData,
        (LPOVERLAPPED *) &PerIoData, INFINITE) == 0)
    {
        sprintf(Str, "GetQueuedCompletionStatus failed "
            "with error %d", GetLastError());
        pLB->AddString(Str);
        return 0;
    }

    // Сначала проверим, не было ли ошибки на сокете;
    // если была, закрываем сокет и очищаем
    // данные описателя и данные операции
    if (BytesTransferred == 0)
    {
        sprintf(Str, "Closing socket %d",
            PerHandleData->Socket);
        pLB->AddString(Str);
```



```

// Отсутствие перемещенных байт (BytesTransferred)
// означает, что сокет закрыт партнером по соединению
// и нам тоже нужно закрыть сокет. Примечание:
// для ссылки на сокет, связанный с операцией
// ввода-вывода, использовались данные описателя.
if (closesocket(PerHandleData->Socket) ==
                                SOCKET_ERROR)
{
    sprintf(Str, "closesocket() failed with "
                "error %d", WSAGetLastError());
    pLB->AddString(Str);
    return 0;
}

GlobalFree(PerHandleData);
GlobalFree(PerIoData);
continue;
}

```

Проверяем, какой процесс завершился: прием данных или отправка. В зависимости от этого корректируем содержимое полей структуры PER_IO_OPERATION_DATA с информацией о количестве принятых и отправленных байтов. Если установлена соответствующая опция, полученная информация выводится на экран.

```

// Проверим, что поле BytesRECV равно нулю.
// Это означает, вызов WSARECV только что завершен.
// В этом случае заполняем BytesRECV значением
// BytesTransferred, полученным через вызов WSARECV().

if (PerIoData->BytesRECV == 0)
{
    PerIoData->BytesRECV = BytesTransferred;
    PerIoData->BytesSEND = 0;

    // Вывод сообщения, если требуется
    if (bPrint)
    {
        unsigned l = sizeof(Str)-1;
        if (l > BytesTransferred)
            l = BytesTransferred;
        strncpy(Str, PerIoData->Buffer, l);
        Str[l]=0;
        pLB->AddString(Str);
    }
}

```

```

else
{
    PerIoData->BytesSEND += BytesTransferred;
}

```

Если есть данные, которые еще не отправлены обратно клиенту, делаем это через асинхронный вызов функции WSA Send.

```

if (PerIoData->BytesRECV > PerIoData->BytesSEND)
{
    // Посылка еще одного запроса WSA Send().
    // Поскольку WSA Send() не гарантирует отправку
    // полностью, продолжим посылать запросы WSA Send(),
    // пока вся информация не будет отправлена.

    ZeroMemory(&(PerIoData->Overlapped),
                sizeof(OVERLAPPED));

    PerIoData->DataBuf.buf =
        PerIoData->Buffer + PerIoData->BytesSEND;
    PerIoData->DataBuf.len =
        PerIoData->BytesRECV - PerIoData->BytesSEND;

    if (WSASend(PerHandleData->Socket,
                &(PerIoData->DataBuf), 1, &SendBytes, 0,
                &(PerIoData->Overlapped), NULL) ==
        SOCKET_ERROR)
    {
        if (WSAGetLastError() != ERROR_IO_PENDING)
        {
            sprintf(Str, "WSASend() failed with "
                        "error %d", WSAGetLastError());
            pLB->AddString(Str);
            return 0;
        }
    }
}

```

В противном случае начинаем процесс приема очередной порции данных от клиента.

```

else
{
    PerIoData->BytesRECV = 0;

    // Больше посылать нечего.
    // Поэтому выдаем запрос WSA Recv().

```

```

Flags = 0;
ZeroMemory(&(PerIoData->Overlapped),
           sizeof(OVERLAPPED));

PerIoData->DataBuf.len = DATA_BUFSIZE;
PerIoData->DataBuf.buf = PerIoData->Buffer;

if (WSARecv(PerHandleData->Socket,
            &(PerIoData->DataBuf), 1, &RecvBytes, &Flags,
            &(PerIoData->Overlapped), NULL) ==
    SOCKET_ERROR)
{
    if (WSAGetLastError() != ERROR_IO_PENDING)
    {
        sprintf(Str, "WSARecv() failed with "
                    "error %d", WSAGetLastError());
        pLB->AddString(Str);
        return 0;
    }
}
}
}
}
}

```

Пример 8.7. Эхо-сервер на основе модели перекрытого ввода-вывода с использованием AcceptEx()

Использование функции AcceptEx() позволяет осуществлять асинхронный прием входящих соединений от клиентов. В сочетании с использованием асинхронных операций приема и отправки информации это дает нам возможность в одном рабочем потоке осуществлять как установление соединения, так и обмен информацией с клиентами.

Сервер работает следующим образом. Он создает сокет для ожидания подключений клиентов. Сокет выставляется на прослушивание обращений клиентов. Сокеты для обслуживания этих соединений создаются по мере необходимости в цикле обработки событий. Первый такой сокет создается еще до цикла.

Приложение создает всего один рабочий поток с главной функцией ListenThread(), где в бесконечном цикле осуществляет прослушивание сети, прием входящих соединений клиентов и их обслуживание. Обслуживание клиента заключается в том, что сервер считывает присланные данные и просто отправляет их назад клиенту. Синхронизация основана на использовании структур перекрытого ввода-вывода, в качестве синхронизирующих объектов используются события.

Открытие проекта

Откройте файл рабочего пространства заготовки проекта Server.dsw, расположенный в директории 08_InputOutput\Callback using AcceptEx\Starter.

Подключение библиотек, объявление констант и глобальных переменных

В стартовом проекте уже подключен файл winsock2.h и библиотека ws2_32.lib.

Для использования данного варианта организации ввода-вывода необходимо дополнительно подключить файл Mswsock.h и библиотеку Mswsock.lib

В начале файла ServerDlg.cpp добавим необходимые объявления типов, глобальных переменных и объявления глобальных функций:

```
// ...  
bool bPrint = false; // Выводить ли сообщения клиентов
```

```
typedef struct _SOCKET_INFORMATION {  
    CHAR Buffer[DATA_BUFSIZE];  
    WSABUF DataBuf;  
    SOCKET Socket;  
    WSAOVERLAPPED Overlapped;  
    DWORD BytesSEND;  
    DWORD BytesRECV;  
} SOCKET_INFORMATION, * LPSOCKET_INFORMATION;
```

```
HWND hWnd_LB; // Для вывода в других потоках
```

```
UINT ListenThread(PVOID lpParam);
```

Здесь ListenThread() – главная функция рабочего потока, в которой организуется процесс прослушивания сети, создается сокет для приема входящих соединений, а также запускается цикл для асинхронного установления связи с клиентами (посредством вызова функции AcceptEx()), а также асинхронного получения и отправки данных.

Также здесь описан шаблон структуры SOCKET_INFORMATION, используемой для хранения информации о сокетах, участвующих в обмене информацией.

Определение и реализация обработчиков событий

Из обработчиков нам потребуется добавить только обработчик события, происходящего при нажатии кнопки "Start" (идентификатор IDC_START).

Обработчик OnStart() запоминает дескриптор окна списка для вывода информации, сохраняет введенный пользователем номер порта, затем запускает основной рабочий поток. В завершение он делает неактивной кнопку "Start", чтобы избежать повторного запуска.

```
void CServerDlg::OnStart()
{
    char Str[81];

    hWnd_LB = m_ListBox.m_hWnd; // Для ListenThread
    GetDlgItem(IDC_PORT)->GetWindowText(Str, sizeof(Str));
    iPort=atoi(Str);
    if (iPort<=0 || iPort>=0x10000)
    {
        AfxMessageBox("Incorrect Port number");
        return;
    }

    AfxBeginThread(ListenThread, NULL);

    GetDlgItem(IDC_START)->EnableWindow(false);
}
```

Главная функция рабочего потока

Функция ListenThread() создает сокет для прослушивания входящих соединений, осуществляет его привязку к интерфейсу и переводит в режим прослушивания. Также создается сокет AcceptSocket для приема входящих соединений.

Затем создается объект типа событие и записывается в начало массива EventArray. Это событие будет использоваться для уведомления потока обслуживания операций ввода-вывода о наличии обращения клиента с целью установления соединения. Для асинхронного приема входящих соединений заполняется структура ListenOverlapped и процесс приема первого соединения инициализируется через вызов функции AcceptEx().

После этого в бесконечном цикле осуществляется прием обращений клиентов и обслуживание асинхронных операций приема и отправки данных.

Разберем основные участки кода. Сначала мы описываем необходимые локальные переменные и массивы, затем загружаем библиотеку Winsock:

```
UINT ListenThread(PVOID lpParam)
{
    DWORD EventTotal = 0;
    WSAEVENT EventArray[WSA_MAXIMUM_WAIT_EVENTS];
    LPSOCKET_INFORMATION SocketArray[WSA_MAXIMUM_WAIT_EVENTS];
    CHAR AcceptBuffer[2 * (sizeof(SOCKADDR_IN) + 16)];
    WSAOVERLAPPED ListenOverlapped;
```

```

DWORD Bytes;
DWORD Index;
DWORD Flags;
DWORD BytesTransferred;
LPSOCKET_INFORMATION SI;
WSADATA wsaData;
SOCKET ListenSocket, AcceptSocket;
SOCKADDR_IN InternetAddr;
DWORD RecvBytes, SendBytes;
DWORD i;
INT Ret;

char Str[200];
CListBox *pLB =
    (CListBox *) (CListBox::FromHandle(hWnd_LB));

if ((Ret = WSASStartup(0x0202,&wsaData)) != 0)
{
    sprintf(Str, "WSAStartup failed with error %d", Ret);
    pLB->AddString(Str);
    WSACleanup();
    return 1;
}

```

Затем создаем сокет, осуществляем его привязку и выставляем на прослушивание входящих соединений.

```

// Создание сокета для прослушивания входящих соединений.
if ((ListenSocket = WSASocket(AF_INET, SOCK_STREAM, 0,
    NULL, 0, WSA_FLAG_OVERLAPPED)) == INVALID_SOCKET)
{
    sprintf(Str, "Failed to get a socket %d",
        WSAGetLastError());
    pLB->AddString(Str);
    return 1;
}

InternetAddr.sin_family = AF_INET;
InternetAddr.sin_addr.s_addr = htonl(INADDR_ANY);
InternetAddr.sin_port = htons(PORT);

if (bind(ListenSocket, (PSOCKADDR) &InternetAddr,
    sizeof(InternetAddr)) == SOCKET_ERROR)
{
    sprintf(Str, "bind() failed with error %d",
        WSAGetLastError());
    pLB->AddString(Str);
    return 1;
}

```

```

if (listen(ListenSocket, 5))
{
    sprintf(Str, "listen() failed with error %d",
        WSAGetLastError());
    pLB->AddString(Str);
    return 1;
}

```

Далее создаем сокет для приема входящих соединений и объект типа событие, предназначенный для сообщения о необходимости приема очередного соединения от клиента. Потом заполняем структуру ListenOverlapped для описания перекрытой операции приема соединения от клиента. Также в счетчике ожидаемых событий отмечаем наличие только что созданного события.

```

// Создание сокета для приема входящих соединений.
if ((AcceptSocket = WSASocket(AF_INET, SOCK_STREAM, 0,
    NULL, 0, WSA_FLAG_OVERLAPPED)) == INVALID_SOCKET)
{
    sprintf(Str, "Failed to get a socket %d",
        WSAGetLastError());
    pLB->AddString(Str);
    return 1;
}
ZeroMemory(&ListenOverlapped, sizeof(OVERLAPPED));
if ((EventArray[0] = ListenOverlapped.hEvent =
    WSACreateEvent()) == WSA_INVALID_EVENT)
{
    sprintf(Str, "WSACreateEvent failed with error %d",
        WSAGetLastError());
    pLB->AddString(Str);
    return 1;
}
EventTotal = 1;

```

Вызываем функцию AcceptEx() для асинхронного приема первого соединения, передавая ей информацию о слушающем сокете, сокете для приема соединения и о структуре ListenOverlapped, содержащей описание данной асинхронной операции.

```

if (AcceptEx(ListenSocket, AcceptSocket, (PVOID)
    AcceptBuffer, 0, sizeof(SOCKADDR_IN) + 16,
    sizeof(SOCKADDR_IN) + 16, &Bytes, &ListenOverlapped)
    == FALSE)
if (WSAGetLastError() != ERROR_IO_PENDING)
{
    sprintf(Str, "AcceptEx failed with error %d",
        WSAGetLastError());
}

```

```

        pLB->AddString(Str);
        return 1;
    }

```

Начинаем в цикле обработку происходящих событий. В качестве таковых выступают события, связанные с обращениями клиентов (результат работы `AcceptEx`), а также события, вызванные необходимостью обслуживать запросы по приему и передаче информации (результат вызовов `WSASend` и `WSARecv`). Сначала дожидаемся, пока произойдет какое-либо событие.

```

while(TRUE)
{
    if ((Index = WSAWaitForMultipleEvents(EventTotal,
                                           EventArray, FALSE, WSA_INFINITE, FALSE))
        == WSA_WAIT_FAILED)
    {
        sprintf(Str, "WSAWaitForMultipleEvents failed %d",
                WSAGetLastError());
        pLB->AddString(Str);
        return 1;
    }
}

```

Если это было событие на слушающем сокете, то проверяем, успешно ли произошло подключение;

```

// Если произошло событие с индексом 0, значит
// была попытка подключения к нашему слушающему сокету

if ((Index - WSA_WAIT_EVENT_0) == 0)
{
    // Проверяем результат перекрытой операции
    // ввода-вывода на прослушивающем сокете

    if (WSAGetOverlappedResult(ListenSocket,
                                &(ListenOverlapped), &BytesTransferred,
                                FALSE, &Flags) == FALSE)
    {
        sprintf(Str, "WSAGetOverlappedResult failed "
                    "with error %d", WSAGetLastError());
        pLB->AddString(Str);
        return 1;
    }

    sprintf(Str, "Socket %d connected", AcceptSocket);
    pLB->AddString(Str);
}

```


Если количество соединений превышает максимально допустимое, разрываем его и продолжаем работу. В противном случае создаем и записываем в массив SocketArray структуру с информацией о вновь созданном сокете.

```
if (EventTotal > WSA_MAXIMUM_WAIT_EVENTS)
{
    sprintf(Str, "Too many connections - "
               "closing socket.");
    pLB->AddString(Str);
    closesocket(AcceptSocket);
    continue;
}
else
{
    // Создание структуры, содержащей информацию
    // о созданном сокете
    if ((SocketArray[EventTotal] =
         (LPSOCKET_INFORMATION) GlobalAlloc(GPTR,
         sizeof(SOCKET_INFORMATION))) == NULL)
    {
        sprintf(Str, "GlobalAlloc() failed with ",
                "error %d ", GetLastError());
        pLB->AddString(Str);
        return 1;
    }
    // Заполнение структуры данными о сокете.
    SocketArray[EventTotal]->Socket = AcceptSocket;
    ZeroMemory(
        &(SocketArray[EventTotal]->Overlapped),
        sizeof(OVERLAPPED));
    SocketArray[EventTotal]->BytesSEND = 0;
    SocketArray[EventTotal]->BytesRECV = 0;
    SocketArray[EventTotal]->DataBuf.len =
        DATA_BUFSIZE;
    SocketArray[EventTotal]->DataBuf.buf =
        SocketArray[EventTotal]->Buffer;
    if ((SocketArray[EventTotal]->Overlapped.hEvent =
         EventArray[EventTotal] = WSACreateEvent())
        == WSA_INVALID_EVENT)
    {
        sprintf(Str, "WSACreateEvent() failed with "
                "error %d", WSAGetLastError());
        pLB->AddString(Str);
        return 1;
    }
}
```

Далее для вновь установленного соединения иницилируем процесс получения данных через асинхронный вызов функции WSARecv:

```
// Посылка запроса WSARecv
// для начала приема на сокете

if (WSARecv(SocketArray[EventTotal]->Socket,
    &(SocketArray[EventTotal]->DataBuf), 1,
    &RecvBytes, &Flags,
    &(SocketArray[EventTotal]->Overlapped), NULL)
    == SOCKET_ERROR)
{
    if (WSAGetLastError() != ERROR_IO_PENDING)
    {
        sprintf(Str, "WSARecv() failed with ",
            "error %d ", WSAGetLastError());
        pLB->AddString(Str);
        return 1;
    }
}

EventTotal++;
}
```

Для приема следующего соединения создадим еще один сокет, новый объект типа событие, а также заполним соответствующим образом структуру ListenOverlapped и вызовем AcceptEx() для запуска новой асинхронной операции приема входящего соединения:

```
// Создание нового сокета для приема будущих
// соединений и еще один вызов AcceptEx.

if ((AcceptSocket = WSASocket(AF_INET, SOCK_STREAM,
    0, NULL, 0, WSA_FLAG_OVERLAPPED))
    == INVALID_SOCKET)
{
    sprintf(Str, "Failed to get a socket %d",
        WSAGetLastError());
    pLB->AddString(Str);
    return 1;
}

WSAResetEvent(EventArray[0]);

ZeroMemory(&ListenOverlapped, sizeof(OVERLAPPED));
ListenOverlapped.hEvent = EventArray[0];
```

```

        if (AcceptEx(ListenSocket, AcceptSocket,
                      (PVOID) AcceptBuffer, 0,
                      sizeof(SOCKADDR_IN) + 16,
                      sizeof(SOCKADDR_IN) + 16, &Bytes,
                      &ListenOverlapped)
                == FALSE)
        {
            if (WSAGetLastError() != ERROR_IO_PENDING)
            {
                sprintf(Str, "AcceptEx failed with error %d",
                        WSAGetLastError());
                pLB->AddString(Str);
                return 1;
            }
        }
        continue;
    }
}

```

Оставшаяся часть цикла посвящена обработке операций ввода-вывода. Определяем индекс произошедшего события и сбрасываем его. Поскольку это не событие с нулевым индексом, значит, произошло событие, связанное с приемом или отправкой информации. Определяем дескриптор связанного с событием сокета и сбрасываем соответствующее событие.

```

SI = SocketArray[Index - WSA_WAIT_EVENT_0];
WSAResetEvent(EventArray[Index - WSA_WAIT_EVENT_0]);

if (WSAGetOverlappedResult(SI->Socket,
                            &(SI->Overlapped), &BytesTransferred,
                            FALSE, &Flags) == FALSE)
{
    sprintf(Str, "WSAGetOverlappedResult failed with "
                "error %d", WSAGetLastError());
    pLB->AddString(Str);
    return 1;
}

```

Если асинхронная операция завершилась неуспешно или партнер разорвал соединение (определяем это по нулевому количеству перемещенных байтов), удаляем ненужные более объекты из массивов SocketArray и EventArray и освобождаем выделенные ресурсы.

```

// Сначала проверим, не закрыл ли партнер соединение,
// и если это так, то закроем сокет и освободим
// структуру SOCKET_INFORMATION, с ним связанную.
if (BytesTransferred == 0)
{
    sprintf(Str, "Closing socket %d", SI->Socket);
    pLB->AddString(Str);
}

```

```

if (closesocket(SI->Socket) == SOCKET_ERROR)
{
    sprintf(Str, "closesocket() failed with ",
            "error %d", WSAGetLastError());
    pLB->AddString(Str);
}

GlobalFree(SI);
WSACloseEvent(EventArray[Index - WSA_WAIT_EVENT_0]);

// Очистка массивов SocketArray и EventArray
// путем удаления дескриптора события и
// структуры описания сокета, если только они
// не в хвосте наших массивов

if ((Index - WSA_WAIT_EVENT_0) + 1 != EventTotal)
    for (i = Index - WSA_WAIT_EVENT_0;
         i < EventTotal; i++)
    {
        EventArray[i] = EventArray[i + 1];
        SocketArray[i] = SocketArray[i + 1];
    }

EventTotal--;

continue;
}

```

Далее в зависимости от состояния полей структуры с информацией о сокете, определяем, требуется отправка или прием данных, и выполняем необходимую операцию. Детали ясны из комментариев.

```

// Проверим, что поле BytesRECV равно нулю.
// Это означает, вызов WSAREcv только что завершен.
// В этом случае заполняем BytesRECV значением
// BytesTransferred, полученным через вызов WSAREcv().

if (SI->BytesRECV == 0)
{
    SI->BytesRECV = BytesTransferred;
    SI->BytesSEND = 0;

    // Вывод сообщения, если требуется
    if (bPrint)
    {
        unsigned l = sizeof(Str)-1;

```

```

        if (l > BytesTransferred)
            l = BytesTransferred;
        strncpy(Str, SI->Buffer, l);
        Str[l]=0;
        pLB->AddString(Str);
    }
}
else
{
    SI->BytesSEND += BytesTransferred;
}

if (SI->BytesRECV > SI->BytesSEND)
{
    // Посылка еще одного запроса WSASend().
    // Поскольку WSASend() не гарантирует отправку
    // полностью, продолжим посылать запросы WSASend(),
    // пока вся информация не будет отправлена.

    ZeroMemory(&(SI->Overlapped), sizeof(WSAOVERLAPPED));
    SI->Overlapped.hEvent =
        EventArray[Index - WSA_WAIT_EVENT_0];

    SI->DataBuf.buf = SI->Buffer + SI->BytesSEND;
    SI->DataBuf.len = SI->BytesRECV - SI->BytesSEND;

    if (WSASend(SI->Socket, &(SI->DataBuf),
                1, &SendBytes, 0, &(SI->Overlapped),
                NULL) == SOCKET_ERROR)
    {
        if (WSAGetLastError() != ERROR_IO_PENDING)
        {
            sprintf(Str, "WSASend() failed with ",
                    "error %d ", WSAGetLastError());
            pLB->AddString(Str);
            return 1;
        }
    }
}
else
{
    SI->BytesRECV = 0;

    // Больше посылать нечего.
    // Поэтому выдаем запрос WSAREcv().

```

```

Flags = 0;
ZeroMemory(&(SI->Overlapped), sizeof(WSAOVERLAPPED));
SI->Overlapped.hEvent =
    EventArray[Index - WSA_WAIT_EVENT_0];

SI->DataBuf.len = DATA_BUFSIZE;
SI->DataBuf.buf = SI->Buffer;

if (WSARecv(SI->Socket, &(SI->DataBuf), 1,
    &RecvBytes, &Flags, &(SI->Overlapped),
    NULL) == SOCKET_ERROR)
{
    if (WSAGetLastError() != ERROR_IO_PENDING)
    {
        sprintf(Str, "WSARecv() failed with ",
            "error %d", WSAGetLastError());
        pLB->AddString(Str);
        return 1;
    }
}
}
}

return 0;
}

```

Пример 8.8. Эхо-сервер на основе модели перекрытого ввода-вывода с использованием процедур завершения и функции `AcceptEx()`

Использование процедур завершения или функций обратного вызова для организации работы сервера было рассмотрено в примере 8.5. В данном случае нам хотелось бы продемонстрировать его в сочетании с асинхронным установлением соединения через вызов функции `AcceptEx`. Это позволяет в одном рабочем потоке осуществлять как установление соединения, так и обмен информацией с клиентами.

Сервер работает следующим образом. Он создает сокет для ожидания подключений клиентов. Сокет выставляется на прослушивание обращений клиентов. Сокеты для обслуживания этих соединений создаются по мере необходимости в цикле обработки событий. После приема очередного соединения инициируется процесс обмена данными через начальный асинхронный запрос на получение информации (вызов `WSARecv()`).

Приложение создает всего один рабочий поток с главной функцией `ListenThread()`, где в бесконечном цикле осуществляет прослушивание сети, прием входящих соединений клиентов и их обслуживание.

Обслуживание клиента заключается в том, что сервер считывает присланные данные и просто отправляет их назад клиенту. Синхронизация основана на использовании структур перекрытого ввода-вывода, в качестве синхронизирующих объектов при приеме соединения используются события, а при обмене данными – функции обратного вызова, точнее одна-единственная функция `WorkerRoutine()`.

Открытие проекта

Откройте файл рабочего пространства заготовки проекта `Server.dsw`, расположенный в директории `08_InputOutput\Callback` using `AcceptEx\Starter`.

Подключение библиотек, объявление констант и глобальных переменных

В стартовом проекте уже подключен файл `winsock2.h` и библиотека `ws2_32.lib`.

Для использования данного варианта организации ввода-вывода необходимо дополнительно подключить файл `Mswsock.h` и библиотеку `Mswsock.lib`

В начале файла `ServerDlg.cpp` добавим необходимые объявления типов, глобальных переменных и объявления глобальных функций:

```
// ...
bool bPrint = false; // Выводить ли сообщения клиентов

typedef struct _SOCKET_INFORMATION {
    OVERLAPPED Overlapped;
    SOCKET Socket;
    CHAR Buffer[DATA_BUFSIZE];
    WSABUF DataBuf;
    DWORD BytesSEND;
    DWORD BytesRECV;
} SOCKET_INFORMATION, *LPSOCKET_INFORMATION;

void CALLBACK WorkerRoutine(DWORD Error,
    DWORD BytesTransferred, LPWSAOVERLAPPED Overlapped,
    DWORD InFlags);

HWND    hWnd_LB;    // Для вывода в других потоках

UINT    ListenThread(PVOID lpParam);
```

Здесь `ListenThread()` – главная функция рабочего потока, в которой организуется процесс прослушивания сети, прием входящих соединений, а также первый вызов `WSARecv()` для начала обмена информацией с

клиентом. Дальнейшее обслуживание осуществляется функцией обратного вызова WorkerRoutine().

Также здесь описан шаблон структуры SOCKET_INFORMATION, используемой для хранения информации о сокетах, участвующих в обмене информацией. Эта структура передается функции WorkerRoutine() в качестве одного из входных параметров.

Определение и реализация обработчиков событий

Из обработчиков нам потребуется только обработчик события, происходящего при нажатии кнопки "Start" (идентификатор IDC_START).

Обработчик OnStart() запоминает дескриптор окна списка для вывода информации, сохраняет введенный пользователем номер порта, затем запускает основной рабочий поток. В завершение он делает неактивной кнопку "Start", чтобы избежать повторного запуска.

```
void CServerDlg::OnStart()
{
    char Str[81];

    hWnd_LB = m_ListBox.m_hWnd; // Для вывода в других потоках
    GetDlgItem(IDC_PORT)->GetWindowText(Str, sizeof(Str));
    iPort=atoi(Str);
    if (iPort<=0 || iPort>=0x10000)
    {
        AfxMessageBox("Incorrect Port number");
        return;
    }

    AfxBeginThread(ListenThread, NULL);

    GetDlgItem(IDC_START)->EnableWindow(false);
}
```

Главная функция рабочего потока

Функция ListenThread() инициализирует библиотеку Winsock. После этого она создает сокет для прослушивания входящих соединений, осуществляет его привязку к интерфейсу и переводит в режим прослушивания. Затем создается объект типа событие, который будет использоваться для уведомления о наличии попытки подключения клиента.

Далее в бесконечном цикле происходит прием подключений от клиентов. Для установления соединения используется функция AcceptEx(). В случае успешного подключения создается новый поток с главной функцией WorkerRoutine(), предназначенный для выполнения перекрытых операций обмена информацией с клиентом.

Разберем основные участки кода функции ListenThread(). Сначала мы описываем необходимые локальные переменные и загружаем библиотеку Winsock. Объявление массива EventArray из одного элемента выглядит несколько необычно, но это объясняется использованием в дальнейшем функции WSAWaitForMultipleEvents(), которая в качестве одного из аргументов принимает именно массив событий (точнее константный указатель на объект типа событие).

```

UINT ListenThread(PVOID lpParam)
{
    DWORD Flags;
    WSADATA wsaData;
    SOCKET ListenSocket, AcceptSocket;
    SOCKADDR_IN InternetAddr;
    DWORD RecvBytes;
    LPSOCKET_INFORMATION SocketInfo;
    WSAEVENT EventArray[1];
    DWORD Index;
    CHAR AcceptBuffer[2 * (sizeof(SOCKADDR_IN) + 16)];
    OVERLAPPED ListenOverlapped;
    DWORD Bytes;
    INT Ret;

    char    Str[200];
    CListBox * pLB =
        (CListBox *) (CListBox::FromHandle(hWnd_LB));

    if ((Ret = WSASStartup(0x0202, &wsaData)) != 0)
    {
        sprintf(Str, "WSASStartup failed with error %d", Ret);
        pLB->AddString(Str);
        WSACleanup();
        return 1;
    }

    Затем создаем сокет, осуществляем его привязку и начинаем прослу-
    шивание входящих соединений.

    if ((ListenSocket = WSASocket(AF_INET, SOCK_STREAM, 0,
        NULL, 0, WSA_FLAG_OVERLAPPED)) == INVALID_SOCKET)
    {
        sprintf(Str, "Failed to get a socket %d",
            WSAGetLastError());
        pLB->AddString(Str);
        return 1;
    }

    InternetAddr.sin_family = AF_INET;
    InternetAddr.sin_addr.s_addr = htonl(INADDR_ANY);

```

```

InternetAddr.sin_port = htons(iPort);

if (bind(ListenSocket, (PSOCKADDR) &InternetAddr,
        sizeof(InternetAddr)) == SOCKET_ERROR)
{
    sprintf(Str, "bind() failed with error %d",
            WSAGetLastError());
    pLB->AddString(Str);
    return 1;
}

if (listen(ListenSocket, 5))
{
    sprintf(Str, "listen() failed with error %d",
            WSAGetLastError());
    pLB->AddString(Str);
    return 1;
}

```

Создаем объект типа событие для определения момента обращения клиента для установления соединения.

```

if ((EventArray[0] = WSACreateEvent()) ==
    WSA_INVALID_EVENT)
{
    sprintf(Str, "WSACreateEvent() failed with error %d",
            WSAGetLastError());
    pLB->AddString(Str);
    return 1;
}

```

Организуем цикл для обработки входящих соединений. Создаем новый сокет для обслуживания клиента, сбрасывает событие EventArray[0], подготавливаем структуру ListenOverlapped и иницилируем асинхронную операцию приема входящего соединения путем вызова функции AcceptEx(). Последняя, в частности, получает через свои аргументы дескрипторы слушающего и принимающего сокетов, а также структуры для описания перекрытой операции (в данном случае для определения момента завершения операции используется механизм событий).

```

while(TRUE)
{
    if ((AcceptSocket = WSASocket(AF_INET, SOCK_STREAM, 0,
        NULL, 0, WSA_FLAG_OVERLAPPED)) == INVALID_SOCKET)
    {
        sprintf(Str, "Failed to get a socket %d",
            WSAGetLastError());
    }
}

```

```

    pLB->AddString(Str);
    return 1;
}

// Подготавливаем структуры ListenOverlapped
ZeroMemory(&ListenOverlapped, sizeof(WSAOVERLAPPED));
WSAResetEvent(EventArray[0]);
ListenOverlapped.hEvent = EventArray[0];

if (AcceptEx(ListenSocket, AcceptSocket,
    (PVOID) AcceptBuffer, 0, sizeof(SOCKADDR_IN)+16,
    sizeof(SOCKADDR_IN)+16, &Bytes,
    &ListenOverlapped) == FALSE)

    if (WSAGetLastError() != ERROR_IO_PENDING)
    {
        sprintf(Str, "AcceptEx failed with error %d",
            WSAGetLastError());
        pLB->AddString(Str);
        return 1;
    }

```

Далее в данном потоке просто ждем, пока подключится очередной клиент:

```

// Ожидаем событий AcceptEx
while(TRUE)
{
    Index = WSAWaitForMultipleEvents(1, EventArray,
        FALSE, WSA_INFINITE, TRUE);

    if (Index == WSA_WAIT_FAILED)
    {
        sprintf(Str, "WSAWaitForMultipleEvents failed "
            "with error %d", WSAGetLastError());
        pLB->AddString(Str);
        return 1;
    }

    if (Index != WAIT_IO_COMPLETION)
    {
        // Событие AcceptEx произошло. Прерываем цикл.
        break;
    }
}

```

После успешного подключения очередного клиента мы начинаем его обслуживание с запуска асинхронной операции приема данных `WSARecv()`. В качестве способа синхронизации здесь используется механизм обратных (callback) функций (у нас это `WorkerRoutine()`). Ее формат предполагает передачу большей части входной информации через указатель на структуру типа `OVERLAPPED`. Однако поскольку требуется передать еще и дополнительную информацию, нами была объявлена структура типа `SOCKET_INFORMATION`, в которой первым полем является структура `OVERLAPPED`. Таким образом, передавая указатель на `OVERLAPPED`, мы фактически передаем указатель на `SOCKET_INFORMATION`, остается только в самой функции обратного вызова произвести соответствующее приведение типа.

Итак, сначала мы создаем структуру `SOCKET_INFORMATION` и заполняем ее поля. После этого происходит вызов `WSARecv()` с указанием `WorkerRoutine()` в качестве функции обратного вызова и указателя на первое поле созданной структуры в качестве аргумента этой функции.

```
// Создаем структуру с информацией о сокете,
// обслуживающем принятое соединение.
if ((SocketInfo = (LPSOCKET_INFORMATION) GlobalAlloc(
    GPTR, sizeof(SOCKET_INFORMATION))) == NULL)
{
    sprintf(Str, "GlobalAlloc() failed with error %d",
        GetLastError());
    pLB->AddString(Str);
    return 1;
}

// Заполнение структуры данными о сокете
SocketInfo->Socket = AcceptSocket;
ZeroMemory(&(SocketInfo->Overlapped),
    sizeof(WSAOVERLAPPED));
SocketInfo->BytesSEND = 0;
SocketInfo->BytesRECV = 0;
SocketInfo->DataBuf.len = DATA_BUFSIZE;
SocketInfo->DataBuf.buf = SocketInfo->Buffer;

Flags = 0;
if (WSARecv(SocketInfo->Socket,
    &(SocketInfo->DataBuf), 1, &RecvBytes,
    &Flags, &(SocketInfo->Overlapped),
    WorkerRoutine) == SOCKET_ERROR)
{
    if (WSAGetLastError() != WSA_IO_PENDING)
    {
        sprintf(Str, "WSARecv() failed with error %d",
            GetLastError());
    }
}
```

```

        pLB->AddString(Str);
        return 1;
    }
}

    sprintf(Str, "Socket %d connected", AcceptSocket);
    pLB->AddString(Str);
}
}

```

Функция обратного вызова для запросов ввода-вывода

Функция WorkerRoutine() вызывается после завершения приема очередной порции данных от клиента.

Разберем этот процесс поэтапно. Сначала опишем и инициализируем необходимые переменные, а также произведем упомянутое выше приведение типа указателя, полученного через третий аргумент, чтобы получить доступ ко всей информации, хранящейся в структуре описания сокета SOCKET_INFORMATION.

```

void CALLBACK WorkerRoutine(DWORD Error,
                            DWORD BytesTransferred,
                            LPWSAOVERLAPPED Overlapped,
                            DWORD InFlags)
{
    DWORD SendBytes, RecvBytes;
    DWORD Flags;

    char Str[200];
    static CListBox *pLB =
        (CListBox *) (CListBox::FromHandle(hWnd_LB));

    // Преобразуем ссылку на структуру WSAOVERLAPPED
    // в ссылку на структуру SOCKET_INFORMATION
    LPSOCKET_INFORMATION SI =
        (LPSOCKET_INFORMATION) Overlapped;

```

Далее проверим код завершения операции и количество перемещенных байтов. Если последнее равно нулю, значит, партнер разорвал соединение. В этом случае закрываем сокет и освобождаем связанные с ним ресурсы (память).

```

    if (Error != 0)
    {
        sprintf(Str, "I/O operation failed with error %d",
                Error);
        pLB->AddString(Str);
    }

```

```

if (BytesTransferred == 0)
{
    sprintf(Str, "Closing socket %d", SI->Socket);
    pLB->AddString(Str);
}

if (Error != 0 || BytesTransferred == 0)
{
    closesocket(SI->Socket);
    GlobalFree(SI);
    return;
}

```

Проверяем, какой процесс завершился: прием данных или отправка. В зависимости от этого корректируем содержимое полей структуры SOCKET_INFORMATION с информацией о количестве принятых и отправленных байтов. Если установлена соответствующая опция, полученная информация выводится на экран.

```

// Проверим, что поле BytesRECV равно нулю.
// Это означает, вызов WSAREcv только что завершен.
// В этом случае заполняем BytesRECV значением
// BytesTransferred, полученным через вызов WSAREcv().

if (SI->BytesRECV == 0)
{
    SI->BytesRECV = BytesTransferred;
    SI->BytesSEND = 0;

    // Вывод сообщения, если требуется
    if (bPrint)
    {
        unsigned l = sizeof(Str)-1;
        if (l > BytesTransferred) l = BytesTransferred;
        strncpy(Str, SI->Buffer, l);
        Str[l]=0;
        pLB->AddString(Str);
    }
}
else
{
    SI->BytesSEND += BytesTransferred;
}

```

Если есть данные, которые еще не отправлены обратно клиенту, делаем это через асинхронный вызов функции WSASend. В качестве функции обратного вызова указываем WorkerRoutine.

```

if (SI->BytesRECV > SI->BytesSEND)
{
    // Посылка еще одного запроса WSASend().
    // Поскольку WSASend() не гарантирует отправку
    // полностью, продолжим посылать запросы WSASend(),
    // пока вся информация не будет отправлена.

    ZeroMemory(&(SI->Overlapped), sizeof(WSAOVERLAPPED));

    SI->DataBuf.buf = SI->Buffer + SI->BytesSEND;
    SI->DataBuf.len = SI->BytesRECV - SI->BytesSEND;

    if (WSASend(SI->Socket, &(SI->DataBuf), 1, &SendBytes,
                0, &(SI->Overlapped), WorkerRoutine)
        == SOCKET_ERROR)
    {
        if (WSAGetLastError() != WSA_IO_PENDING)
        {
            sprintf(Str, "WSASend() failed with error %d",
                    WSAGetLastError());
            pLB->AddString(Str);
            return;
        }
    }
}

```

В противном случае начинаем процесс приема очередной порции данных от клиента.

```

else
{
    SI->BytesRECV = 0;

    // Больше посылать нечего.
    // Поэтому выдаем запрос WSAREcv().

    Flags = 0;
    ZeroMemory(&(SI->Overlapped), sizeof(WSAOVERLAPPED));

    SI->DataBuf.len = DATA_BUFSIZE;
    SI->DataBuf.buf = SI->Buffer;

    if (WSAREcv(SI->Socket, &(SI->DataBuf), 1, &RecvBytes,
                &Flags, &(SI->Overlapped), WorkerRoutine) ==
        SOCKET_ERROR)
    {
        if (WSAGetLastError() != WSA_IO_PENDING)
        {
            sprintf(Str, "WSAREcv() failed with error %d",
                    WSAGetLastError());

```

```
        pLB->AddString(Str);  
        return;  
    }  
}  
}
```


Многоадресная рассылка в сетях IP

Целью рассматриваемых в рамках данной темы примеров является изучение способов организации многоадресной рассылки информации в сетях IP с использованием сетевого интерфейса Winsock как первой, так и второй версии. Перед выполнением предлагаемых практических заданий рекомендуется вспомнить соответствующий теоретический материал, например, перечитав главу "Многоадресная рассылка" из учебного пособия [7].

Многоадресная рассылка (multicasting) – это технология, позволяющая отправлять данные от одного узла в сети, а затем тиражировать их множеству других, не создавая при этом большой нагрузки на сеть. Она является альтернативой широковещанию (broadcasting), которое при активном использовании может существенно снизить пропускную способность сети. При многоадресной рассылке данные передаются в сеть, только если процессы, выполняемые на других рабочих станциях в сети, запрашивают их.

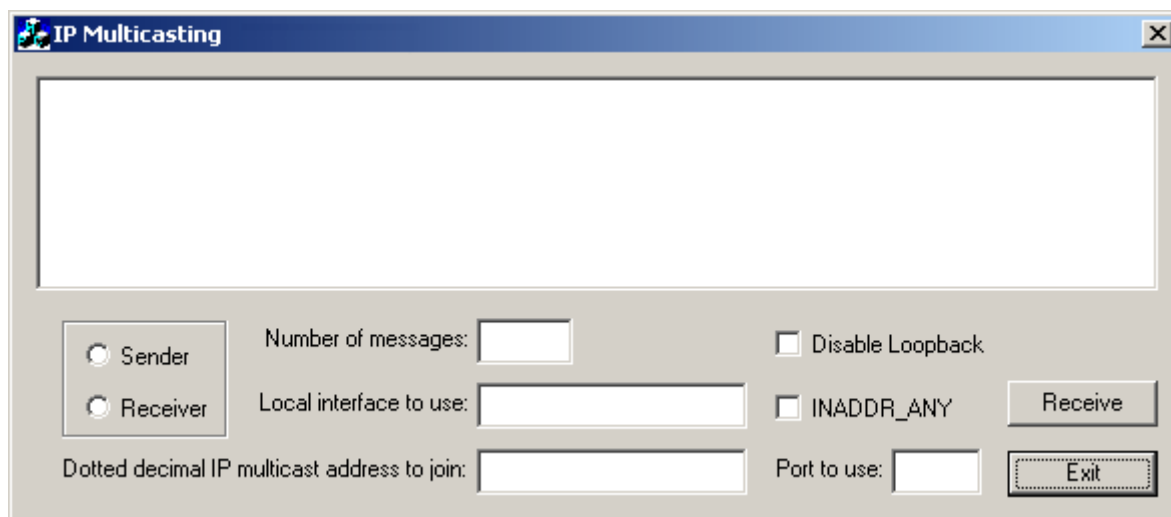
Мы разберем примеры организации многоадресной рассылки на базе интерфейса Winsock 1 и на базе интерфейса Winsock 2.

Поскольку нашей целью является изучение именно моментов, специфичных для каждой версии интерфейса Winsock, стартовые проекты практически идентичны (за исключением заголовка главного окна). Кроме того, значительная часть работы по настройке элементов управления в зависимости от текущего состояния приложения также уже сделана.

Ниже описываются основные элементы этого стартового проекта. Отметим сразу, что стартовый проект *не компилируется*. Это сделано намеренно, поскольку для его компиляции и сборки в разных моделях Winsock требуется подключить разные заголовки и библиотеки. При тестировании также следует учесть, что многоадресная рассылка не надежна и, следовательно, *не гарантирует* доставки всех сообщений всем адресатам.

Главное диалоговое окно

Предлагаемая заготовка стартового проекта для обоих рассматриваемых примеров представляет собой диалоговое приложение с главным окном следующего вида



Список в верхней части окна (идентификатор IDC_LISTBOX) предназначен для вывода информации о ходе работы. Радиокнопки (IDC_SENDER и IDC_RECEIVER) служат для переключения из режима передачи в режим приема. Поле ввода IDC_NUMBER позволяет ввести количество отправляемых сообщений (только в режиме Sender). Отметив флажок IDC_INADDR_ANY, можно разрешить выбрать локальный интерфейс по умолчанию, в противном случае потребуется ввести его IP-адрес в поле ввода IDC_LOCAL.

В поле IDC_ADDRESS нужно ввести IP-адрес для группы, к которой мы присоединяемся. Напомним, что этот адрес должен лежать в диапазоне 224.0.0.0 – 235.255.255.255. Поле ввода IDC_PORT предназначено для выбора порта.

Флажок IDC_LOOPBACK используется для запрета петли (в этом случае отправитель не будет сам получать свои сообщения).

Кнопка с идентификатором IDC_START (на рисунке имеет надпись "Receive") запускает процесс посылки или приема. Надпись на кнопке оперативно изменяется в зависимости от состояния приложения. Наконец, кнопка "Exit" (идентификатор IDCANCEL) завершает работу программы.

Что уже сделано в стартовом проекте

В начале файла McastDlg.cpp имеется объявление следующих констант, задающих значения настроечных параметров по умолчанию:

```
#define MCASTADDR      "234.5.6.7"
#define MCASTPORT      25000
#define BUFSIZE        1024
#define DEFAULT_COUNT  10
#define MIN_COUNT      1
#define MAX_COUNT      50
```

Также произведено описание и инициализация глобальных (для простоты) переменных, хранящих текущие значения этих настроечных параметров

```

BOOL  bSender = FALSE,      // Действовать как отправитель?
      bLoopBack = FALSE,    // Запретить петли?
      bInterface = FALSE,   // Local interface != INADDR_ANY
      bStopRecv = TRUE;     // Остановить прием?

DWORD dwInterface,          // Локальный интерфейс
      dwMulticastGroup,     // Многоадресная группа
      dwCount;              // Кол-во сообщ. для отправки

short iPort = MCASTPORT;    // Номер порта

```

В стартовом проекте в класс CMcastDlg добавлены в качестве членов две переменные: m_ListBox типа CListBox для управления окном вывода IDC_LISTBOX, а также m_Sender типа int для работы с переключателем Sender/Receiver.

Добавлены и реализованы обработчики событий, связанных с изменением состояния флажка IDC_INADDR_ANY и переключателя Sender/Receiver.

Добавлены в класс CMcastDlg и реализованы функции RefreshValues() и SetUI() для оперативного изменения состояния элементов управления в зависимости от ситуации. Также в функции CMcastDlg::OnInitDialog() добавлен код для начальной настройки диалога. Сам код здесь не приводится, с ним можно ознакомиться, открыв стартовый проект.

Пример 11.1. Многоадресная рассылка в сетях IP с использованием Winsock 1

Пример демонстрирует, каким образом можно подключиться к многоадресной группе и осуществить передачу и прием сообщений на основе сетевого интерфейса Winsock 1. Метод подключения к многоадресной группе в этом случае заключается в соответствующем изменении параметров сокета через вызов функции setsockopt().

Открытие проекта

Откройте файл рабочего пространства заготовки проекта Mcast.dsw, расположенный в директории 11_MultiAddr\IP_WS1\Starter.

Подключение библиотек, объявление глобальных переменных и функций

Подключим заголовок winsock.h в начале файла McastDlg.cpp. Обратите внимание на то, что в случае Winsock 1 подключать следует именно этот файл. Также обратите внимание на место расположения соответствующей директивы #include. Его изменение может вызвать ошибки компиляции с не слишком вразумительными сообщениями.

```
// ...
#include "McastDlg.h"
```

```
#include <winsock.h>
```

```
#ifdef _DEBUG
// ...
```

Далее следует подключить к проекту необходимую библиотеку. В случае Winsock 1 – это Wsock32.lib. При использовании среды Microsoft Visual Studio 6 подключение делается через пункт меню Project Settings на вкладке Link в окне "Project/library modules:".

В начале файла McastDlg.cpp добавим дополнительные объявления глобальных переменных и глобальных функций:

```
HWND hWnd_LB;          // Для рабочего потока
HWND hDlg;             // Для рабочего потока
UINT WorkerThread(PVOID lpParam);
DWORD ReceiveData(SOCKET sockM);
```

```
////////////////////////////////////
// CAboutDlg dialog used for App About
```

Здесь переменные hWnd_LB и hDlg нужны для передачи рабочему потоку дескриптора окна вывода IDC_LISTBOX и дескриптора диалога соответственно. WorkerThread() будет использоваться в качестве главной функции рабочего потока, а функция ReceiveData() – для приема данных.

Определение и реализация обработчиков событий

В дополнение к уже реализованным нам потребуется обработчик только одного события, а именно происходящего при нажатии кнопки с идентификатором IDC_START. Эта кнопка в зависимости от ситуации используется для осуществления одного из трех действий:

- отправки заданного количества сообщений,
- приема сообщений в бесконечном цикле,
- прекращения приема (выход из цикла),

Текущее состояние приложения можно определить по значению переключателя Sender/Receiver (переменная m_Sender) и переменной bStopRecv, которая принимает значение false, если идет процесс приема сообщений (бесконечный цикл в рабочем потоке).

Наш обработчик будет проверять значение этой переменной и при необходимости менять его на противоположное, чтобы остановить прием. Если же прием не был запущен, то надо запустить процесс обмена данными (в зависимости от значения переключателя Sender/Receiver). Для этого мы просто запоминаем дескриптор окна вывода и запускаем рабочий поток, который и выполнит всю оставшуюся работу.

```

void CMcastDlg::OnStart()
{
    // TODO: Add your control notification handler code here
    if (!bStopRecv) // Запущен процесс приема
    {
        bStopRecv = TRUE;
        return;
    }
    hWnd_LB = m_ListBox.m_hWnd; // Для рабочего потока
    if (!RefreshValues())
        return;
    AfxBeginThread(WorkerThread, NULL);
}

```

Главная функция рабочего потока

Функция WorkerThread() инициализирует библиотеку Winsock, создает сокет и присоединяет его к многоадресной группе. Далее, если приложение запущено, как отправитель, то осуществляется отправка сообщений, в противном случае происходит их чтение.

Разберем основные участки кода. Сначала мы описываем необходимые локальные переменные и загружаем библиотеку Winsock:

```

// Функция: WorkerThread
// Загрузка библиотеки Winsock, создание сокета,
// присоединение к многоадресной группе.
// Если приложение запущено, как отправитель,
// то осуществляется отправка сообщений, иначе их чтение.

UINT WorkerThread(PVOID lpParam)
{
    WSADATA          wsd;
    struct sockaddr_in local,
                    remote;
    struct ip_mreq    mcast;
    SOCKET            sockM;
    TCHAR             sendbuf[BUFSIZE];

    int              len = sizeof(struct sockaddr_in),
                    optval;
    DWORD            i=0;
    char             Str[200];
    CListBox          *pLB =
                    (CListBox *) (CListBox::FromHandle(hWnd_LB));
    CMcastDlg         *pDlg =
                    (CMcastDlg *) (CMcastDlg::FromHandle(hDlg));

```

```

// Загрузка библиотеки Winsock
//
if (WSAStartup(MAKEWORD(1, 1), &wsd) != 0)
{
    sprintf(Str, "WSAStartup failed");
    pLB->AddString(Str);
    return -1;
}

Далее создаем сокет и привязываем его к локальному интерфейсу

// Создание сокета. В Winsock 1 не требуется
// никаких флагов для указания многоадресности.
//
if ((sockM = socket(AF_INET, SOCK_DGRAM, 0)) ==
                                INVALID_SOCKET)
{
    sprintf(Str, "socket failed with: %d",
            WSAGetLastError());
    pLB->AddString(Str);
    WSACleanup();
    return -1;
}

// Привязка к локальному интерфейсу
// Это требуется для приема данных.
//
local.sin_family = AF_INET;
local.sin_port   = htons(iPort);
local.sin_addr.s_addr = dwInterface;

if (bind(sockM, (struct sockaddr *)&local,
          sizeof(local)) == SOCKET_ERROR)
{
    sprintf(Str, "bind failed with: %d",
            WSAGetLastError());
    pLB->AddString(Str);
    closesocket(sockM);
    WSACleanup();
    return -1;
}

```

Осуществляем подключение сокета к многоадресной группе. В Winsock 1 это делается через изменение параметров сокета путем вызова функции `setsockopt()`. Необходимая информация последней передается через структуру `in_req`:

```

// Настройка интерфейса и структуры im_req
// для подключения к группе
//
remote.sin_family      = AF_INET;
remote.sin_port        = htons(iPort);
remote.sin_addr.s_addr = dwMulticastGroup;

mcast.imr_multiaddr.s_addr = dwMulticastGroup;
mcast.imr_interface.s_addr = dwInterface;

if (setsockopt(sockM, IPPROTO_IP, IP_ADD_MEMBERSHIP,
               (char *)&mcast, sizeof(mcast)) == SOCKET_ERROR)
{
    sprintf(Str, "setsockopt(IP_ADD_MEMBERSHIP) "
               "failed: %d", WSAGetLastError());
    pLB->AddString(Str);
    closesocket(sockM);
    WSACleanup();
    return -1;
}

```

Производим дополнительные настройки, в частности, установку значения TTL (фактически оно означает через сколько маршрутизаторов можно проходить) и запрет петли, если нужно.

```

// Настройка значения TTL, по умолчанию оно равно 1.
//
optval = 8;

if (setsockopt(sockM, IPPROTO_IP, IP_MULTICAST_TTL,
               (char *)&optval, sizeof(int)) == SOCKET_ERROR)
{
    sprintf(Str, "setsockopt(IP_MULTICAST_TTL) "
               "failed: %d", WSAGetLastError());

    pLB->AddString(Str);
    closesocket(sockM);
    WSACleanup();
    return -1;
}

// Запрет петли, если требуется.
// В Windows NT4 и Windows 95 петлю запретить нельзя.
//
if (bLoopBack)
{
    optval = 0;
}

```

```

        if (setsockopt(sockM, IPPROTO_IP, IP_MULTICAST_LOOP,
            (char *)&optval, sizeof(optval))
                == SOCKET_ERROR)
        {
            sprintf(Str, "setsockopt(IP_MULTICAST_LOOP) "
                "failed: %d", WSAGetLastError());
            pLB->AddString(Str);

            closesocket(sockM);
            WSACleanup();
            return -1;
        }
    }
}

```

Далее идет собственно отправка или получение данных. Для удобства отправка осуществляется во вспомогательной функции ReceiveData() в бесконечном цикле. При этом надпись на кнопке IDC_START меняется на "Stop", и ее нажатие приведет к выходу из этого цикла. За настройку пользовательского интерфейса перед вызовом ReceiveData() и после возврата отвечает вызов функции SetUI().

```

    if (!bSender)    // Получатель
    {
        bStopRecv = FALSE;
        pLB->AddString
            ("*** Press Stop to discontinue receiving ***");
        pDlg->SetUI();
        ReceiveData(sockM); // Прием порции данных
        pDlg->SetUI();
    }

```

Если приложение запущено как отправитель, мы отправляем заданное количество тестовых сообщений. В данном примере между отправкой двух последовательных сообщений предусмотрена для наглядности пауза в полсекунды. После отправки, если не была запрещена петля, запускаем еще и прием собственных сообщений (для демонстрации того, что они все-таки пришли). При этом приложение ведет себя как получатель, а после нажатия кнопки "Stop" возвращается в исходное состояние.

```

    else    // Отправитель
    {
        // Отправка порции данных
        //
        for(i = 0; i < dwCount; i++)
        {
            sprintf(sendbuf, "Multicast Sender: "
                "This is a test: %d", i);

```



```

        if (sendto(sockM, (char *)sendbuf,
                    strlen(sendbuf), 0,
                    (struct sockaddr *)&remote,
                    sizeof(remote)) == SOCKET_ERROR)
        {
            sprintf(Str, "sendto failed with: %d",
                    WSAGetLastError());
            pLB->AddString(Str);
            closesocket(sockM);
            WSACleanup();
            return -1;
        }

        sprintf(Str, "*** String sent: %s", sendbuf);
        pLB->AddString(Str);
        Sleep(500);
    }

    if (!bLoopBack)
    {
        bStopRecv = FALSE;
        pDlg->SetUI();
        pLB->AddString
            ("*** Press Stop to discontinue receiving ***");
        ReceiveData(sockM);
        pDlg->SetUI();
    }
}

```

Наконец, выходим из группы и завершаем работу потока путем обычного возврата из его главной функции. В случае Winsock 1 выход из группы осуществляется также через вызов setsockopt().

```

// Выход из группы
//
if (setsockopt(sockM, IPPROTO_IP, IP_DROP_MEMBERSHIP,
               (char *)&mcast, sizeof(mcast)) == SOCKET_ERROR)
{
    sprintf(Str, "setsockopt(IP_DROP_MEMBERSHIP) "
                "failed: %d", WSAGetLastError());
    pLB->AddString(Str);
}
closesocket(sockM);

WSACleanup();
return 0;
}

```

Последней добавим в наше приложение функцию для приема сообщений. Она это делает в бесконечном цикле, проверяя каждый раз значение `bStopRecv`, установка которого в `true` означает приказ завершить прием. Проверка наличия данных на входе осуществляется с использованием функции `select()`, которая вызывается здесь с ограничением времени ожидания до 0.2 секунды.

```
// Функция для получения данных через сокет
DWORD ReceiveData(SOCKET sockM)
{
    TCHAR    recvbuf[BUFSIZE];
    FD_SET   ReadSet;
    struct sockaddr_in  from;
    struct timeval  delay;

    int      len = sizeof(struct sockaddr_in),
            ret,
            i=0;

    char      Str[200];
    CListBox* pLB =
        (CListBox *) (CListBox::FromHandle(hWnd_LB));

    // Получение данных
    //
    while (!bStopRecv)
    {
        FD_ZERO(&ReadSet);
        FD_SET(sockM, &ReadSet);

        delay.tv_sec = 0;
        delay.tv_usec = 200; // 0.2 сек.

        if ((ret = select(0, &ReadSet, NULL, NULL,
                        &delay)) == SOCKET_ERROR)
        {
            sprintf(Str, "select() returned with error "
                        "%d", WSAGetLastError());
            pLB->AddString(Str);
            return 1;
        }

        if (ret == 0) // Данных пока нет
            continue;
    }
}
```

```

        if ((ret = recvfrom(sockM, recvbuf, BUFSIZE, 0,
                           (struct sockaddr *)&from, &len))
            == SOCKET_ERROR)
        {
            sprintf(Str, "recvfrom failed with: %d",
                    WSAGetLastError());
            pLB->AddString(Str);

            closesocket(sockM);
            WSACleanup();
            return -1;
        }

        recvbuf[ret] = 0;
        sprintf(Str, "Receive: '%s' from <%s>", recvbuf,
                inet_ntoa(from.sin_addr));
        pLB->AddString(Str);
    }
    return 0;
}

```

Пример 11.2. Организация многоадресной рассылки в сетях IP средствами Winsock 2

Пример демонстрирует, каким образом можно подключиться к многоадресной группе и осуществить передачу и прием сообщений на основе сетевого интерфейса Winsock 2. Обратите внимание, что метод подключения к многоадресной группе и отключения от нее в этом случае отличается от рассмотренного выше. Поскольку часть работы над примером не отличается от только что сделанной, мы в тексте просто ограничимся ссылками на предыдущий пример.

Открытие проекта

Откройте файл рабочего пространства заготовки проекта Mcast.dsw, расположенный в директории 11_MultiAddr\IP_WS2\Starter.

Подключение библиотек, объявление глобальных переменных и функций

Подключим заголовки winsock2.h и ws2tcpip.h в начале файла McastDlg.cpp. Обратите внимание на то, что в случае Winsock 2 подключать следует именно эти два файла. Как и ранее, обратите внимание на место расположения соответствующих директив #include.

```
// ...
#include "McastDlg.h"
```

```
#include <winsock2.h>
#include <ws2tcpip.h>
```

```
#ifdef _DEBUG
// ...
```

Далее следует подключить к проекту необходимую библиотеку. В случае Winsock 2 – это Ws2_32.lib. При использовании среды Microsoft Visual Studio 6 подключение делается через пункт меню Project Settings на вкладке Link в окне "Project/library modules".

В начале файла McastDlg.cpp добавим дополнительные объявления глобальных переменных и глобальных функций:

```
HWND  hWnd_LB;          // Для рабочего потока
HWND  hDlg;             // Для рабочего потока
UINT  WorkerThread(PVOID lpParam);
DWORD ReceiveData(SOCKET sock, SOCKET sockM);
```

```
////////////////////////////////////
// CAboutDlg dialog used for App About
```

Здесь, как и ранее, переменные hWnd_LB и hDlg нужны для передачи рабочему потоку дескриптора окна вывода IDC_LISTBOX и дескриптора диалога соответственно. WorkerThread() будет использоваться в качестве главной функции рабочего потока, а функция ReceiveData() – для приема данных.

Определение и реализация обработчиков событий

В дополнение к уже реализованным нам потребуется обработчик только одного события, а именно происходящего при нажатии кнопки с идентификатором IDC_START. Смысл действий, которые должен произвести обработчик и предлагаемый код ничем не отличается от рассмотренного выше, поэтому отсылаем читателя к примеру 11.1.

Главная функция рабочего потока

Функция WorkerThread() инициализирует библиотеку Winsock, создает сокет и присоединяет его к многоадресной группе. Далее, если приложение запущено, как отправитель, то осуществляется отправка сообщений, в противном случае происходит их чтение.

Разберем основные участки кода. Сначала мы описываем необходимые локальные переменные и загружаем библиотеку Winsock:

```

// Функция: WorkerThread
// Загрузка библиотеки Winsock, создание сокета,
// присоединение к многоадресной группе.
// Если приложение запущено, как отправитель,
// то осуществляется отправка сообщений, иначе их чтение.

UINT WorkerThread(PVOID lpParam)
{
    WSADATA          wsd;
    struct sockaddr_in local,
                    remote;
    SOCKET           sock,
                    sockM;
    TCHAR            sendbuf[BUFSIZE];

    int              len = sizeof(struct sockaddr_in),
                    optval;
    DWORD            i=0;

    char             Str[200];
    CListBox          *pLB =
                    (CListBox *) (CListBox::FromHandle(hWnd_LB));
    CMcastDlg         *pDlg =
                    (CMcastDlg *) (CMcastDlg::FromHandle(hDlg));

    // Загрузка библиотеки Winsock
    //
    if (WSAStartup(MAKEWORD(2, 2), &wsd) != 0)
    {
        sprintf(Str, "WSAStartup failed");
        pLB->AddString(Str);
        return -1;
    }

    Далее создаем сокет и привязываем его к локальному интерфейсу
    // Создание сокета. В Winsock 2 требуется указать
    // необходимые флаги для поддержки многоадресности.
    //
    if ((sock = WSASocket(AF_INET, SOCK_DGRAM, 0, NULL, 0,
                        WSA_FLAG_MULTIPOINT_C_LEAF
                        | WSA_FLAG_MULTIPOINT_D_LEAF
                        | WSA_FLAG_OVERLAPPED)) == INVALID_SOCKET)
    {
        sprintf(Str, "socket failed with: %d",
                WSAGetLastError());
        pLB->AddString(Str);
    }
}

```

```

    WSACleanup();
    return -1;
}
// Привязка к локальному интерфейсу
// Это требуется для приема данных.
//
local.sin_family = AF_INET;
local.sin_port   = htons(iPort);
local.sin_addr.s_addr = dwInterface;
if (bind(sock, (struct sockaddr *)&local,
        sizeof(local)) == SOCKET_ERROR)
{
    sprintf(Str, "bind failed with: %d",
            WSAGetLastError());
    pLB->AddString(Str);
    closesocket(sock);
    WSACleanup();
    return -1;
}

```

Перед присоединением к многоадресной группе нам потребуется настроить специальную структуру с ее описанием

```

// Настройка структуры SOCKADDR_IN, описывающей
// многоадресную группу для подключения
//
remote.sin_family      = AF_INET;
remote.sin_port        = htons(iPort);
remote.sin_addr.s_addr = dwMulticastGroup;

```

Производим дополнительные настройки, в частности, установку значения TTL и запрет петли, если нужно.

```

// Настройка значения TTL, по умолчанию оно равно 1.
//
optval = 8;

if (setsockopt(sock, IPPROTO_IP, IP_MULTICAST_TTL,
    (char *)&optval, sizeof(int)) == SOCKET_ERROR)
{
    sprintf(Str, "setsockopt(IP_MULTICAST_TTL) "
            "failed: %d", WSAGetLastError());
    pLB->AddString(Str);
    closesocket(sock);
    WSACleanup();
    return -1;
}

```

```

// Запрет петли, если требуется.
// В Windows NT4 и Windows 95 петлю запретить нельзя.
//
if (bLoopBack)
{
    optval = 0;
    if (setsockopt(sock, IPPROTO_IP, IP_MULTICAST_LOOP,
        (char *)&optval, sizeof(optval))
        == SOCKET_ERROR)
    {
        sprintf(Str, "setsockopt(IP_MULTICAST_LOOP) "
            "failed: %d", WSAGetLastError());
        pLB->AddString(Str);
        closesocket(sock);
        WSACleanup();
        return -1;
    }
}

```

Присоединяемся к многоадресной группе. В Winsock 2 для этого используется функция `WSAJoinLeaf()`.

```

// Подсоединение к многоадресной группе.
// Замечание: sockM не используется для
// отправки или получения данных.
// Он используется, когда мы хотим выйти
// из многоадресной группы. Тогда нужно
// просто вызвать closesocket() на нем.
//
if ((sockM = WSAJoinLeaf(sock, (SOCKADDR *)&remote,
    sizeof(remote), NULL, NULL, NULL, NULL,
    JL_BOTH)) == INVALID_SOCKET)
{
    sprintf(Str, "WSAJoinLeaf() failed: %d",
        WSAGetLastError());
    pLB->AddString(Str);
    closesocket(sock);
    WSACleanup();
    return -1;
}

```

Далее идет собственно отправка или получение данных. Для удобства отправка осуществляется во вспомогательной функции `ReceiveData()` в бесконечном цикле. При этом надпись на кнопке `IDC_START` меняется на "Stop", и ее нажатие приведет к выходу из этого цикла. За настройку пользовательского интерфейса перед вызовом `ReceiveData()` и после

возврата отвечает функция SetUI(). Соответствующий код слегка отличается от кода из предыдущего примера, поэтому приводим его полностью

```
if (!bSender)    // Получатель
{
    bStopRecv = FALSE;
    pLB->AddString
        ("*** Press Stop to discontinue receiving ***");
    pDlg->SetUI();
    ReceiveData(sock, sockM); // Прием порции данных
    pDlg->SetUI();
}
```

Если приложение запущено как отправитель, мы отправляем заданное количество тестовых сообщений. В данном примере между отправкой двух последовательных сообщений предусмотрена для наглядности пауза в полсекунды. После отправки, если не была запрещена петля, запускаем еще и прием собственных сообщений (для демонстрации того, что они все-таки пришли). При этом приложение ведет себя как получатель, а после нажатия кнопки "Stop" возвращается в исходное состояние.

```
else    // Отправитель
{
    // Отправка порции данных
    //
    for(i = 0; i < dwCount; i++)
    {
        sprintf(sendbuf, "Multicast Sender: "
                        "This is a test: %d", i);
        if (sendto(sock, (char *)sendbuf,
                    strlen(sendbuf), 0,
                    (struct sockaddr *)&remote,
                    sizeof(remote)) == SOCKET_ERROR)
        {
            sprintf(Str, "sendto failed with: %d",
                    WSAGetLastError());
            pLB->AddString(Str);
            closesocket(sockM);
            closesocket(sock);
            WSACleanup();
            return -1;
        }
        sprintf(Str, "*** String sent: %s", sendbuf);
        pLB->AddString(Str);
        Sleep(500);
    }
}
```



```

        if (!bLoopBack)
        {
            bStopRecv = FALSE;
            pDlg->SetUI();
            pLB->AddString
                ("*** Press Stop to discontinue receiving ***");
            ReceiveData(sock, sockM);
            pDlg->SetUI();
        }
    }
}

```

Наконец, выходим из группы и завершаем работу потока.

```

// Выход из группы путем закрытия sock.
// При использовании немаршрутизируемых
// плоскостей управления и данных WSAJoinLeaf
// возвращает тот же описатель сокета,
// который мы ей передали
//
closesocket(sock);

WSACleanup();
return 0;
}

```

Последней добавим в наше приложение функцию для приема сообщений. Она это делает в бесконечном цикле, проверяя каждый раз значение bStopRecv, установка которого в true означает приказ завершить прием. Проверка наличия данных на входе осуществляется с использованием функции select(), которая вызывается здесь с ограничением времени ожидания до 0.2 секунды. Поскольку ее код также несколько отличается от кода функции ReceiveData() из предыдущего примера, он приводится полностью.

```

// Функция для получения данных через сокет
DWORD ReceiveData(SOCKET sock, SOCKET sockM)
{
    TCHAR    recvbuf[BUFSIZE];
    FD_SET   ReadSet;
    struct sockaddr_in  from;
    struct timeval  delay;

    int      len = sizeof(struct sockaddr_in),
            ret,
            i=0;

    char     Str[200];
    CListBox* pLB =
        (CListBox *) (CListBox::FromHandle(hWnd_LB));
}

```

```

// Получение данных
//
while (!bStopRecv)
{
    FD_ZERO(&ReadSet);
    FD_SET(sock, &ReadSet);

    delay.tv_sec = 0;
    delay.tv_usec = 200; // 0.2 сек.

    if ((ret = select(0, &ReadSet, NULL, NULL,
                     &delay)) == SOCKET_ERROR)
    {
        sprintf(Str, "select() returned with error "
                    "%d", WSAGetLastError());
        pLB->AddString(Str);
        return 1;
    }

    if (ret == 0) // Данных пока нет
        continue;

    if ((ret = recvfrom(sock, recvbuf, BUFSIZE, 0,
                       (struct sockaddr *)&from, &len))
        == SOCKET_ERROR)
    {
        sprintf(Str, "recvfrom failed with: %d",
                WSAGetLastError());
        pLB->AddString(Str);
        closesocket(sockM);
        closesocket(sock);
        WSACleanup();
        return -1;
    }
    recvbuf[ret] = 0;
    sprintf(Str, "Receive: '%s' from <%s>", recvbuf,
            inet_ntoa(from.sin_addr));
    pLB->AddString(Str);
}
return 0;
}

```

Список литературы

1. Джонс, Э. Программирование в сетях Microsoft Windows. Мастер-класс / Э. Джонс, Дж. Оланд. – СПб.: Питер, 2002.
2. Снейдер, Й. Эффективное программирование TCP/IP / Й. Снейдер. – СПб.: Питер, 2002.
3. Олафсен, Ю. MFC и Visual C++ 6. Энциклопедия программиста / Ю. Олафсен, К. Скрайбер, К. Д. Уайт и др. – СПб.: ООО "ДиаСофтЮП", 2004.
4. Круглински, Д. Программирование на Microsoft Visual C++ 6.0 для профессионалов / Д. Круглински, С. Уингоу, Дж. Шеферд. – СПб.: Питер, 2001.
5. Рихтер, Дж. Windows для профессионалов: создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows / Дж. Рихтер. – СПб.: Питер, 2001.
6. Васильчиков, В. В. Программирование в Visual C++ с использованием библиотеки MFC: учебное пособие / В. В. Васильчиков; Яросл. гос. ун-т. – Ярославль: ЯрГУ, 2006.
7. Васильчиков, В. В. Основы разработки сетевых Windows-приложений: учебное пособие / В. В. Васильчиков; Яросл. гос. ун-т. – Ярославль: ЯрГУ, 2007.

Оглавление

Введение	3
Интерфейс NetBIOS	5
Пример 1.1. Библиотека общих функций для приложений NetBIOS	5
Пример 1.2. Эхо-клиент на основе сетевого интерфейса NetBIOS	12
Пример 1.3. Эхо-сервер NetBIOS, использующий функции обратного вызова	20
Пример 1.4. Эхо-сервер NetBIOS, основанный на модели событий	26
Пример 1.5. Приложение для отправки и приема дейтаграмм с использованием интерфейса NetBIOS	35
Перенаправитель	51
Пример 2.1. Пример создания файла по UNC-соединению	51
Почтовые ящики	55
Пример 3.1. Простой сервер почтовых ящиков	55
Пример 3.2. Простой клиент почтовых ящиков	59
Именованные каналы	63
Пример 4.1. Пример многопоточного эхо-сервера именованных каналов	63
Пример 4.2. Простой клиент именованных каналов	67
Пример 4.3. Пример эхо-сервера именованных каналов, работающего в режиме перекрытого ввода-вывода	72
Сетевые протоколы	79
Пример 5.1. Перечисление установленных в системе сетевых протоколов	79
Основы интерфейса Winsock	85
Пример 7.1. Эхо-сервер на основе протокола TCP	85
Пример 7.2. Клиент для эхо-сервера на основе протокола TCP	92
Пример 7.3. Получатель дейтаграмм на основе протокола UDP	98
Пример 7.4. Отправитель дейтаграмм на основе протокола UDP	104

Ввод-вывод в Winsock	110
Пример 8.1. Эхо-сервер на основе модели select().....	111
Пример 8.2. Эхо-сервер на основе модели AsyncSelect().....	120
Пример 8.3. Эхо-сервер на основе модели EventSelect()	131
Пример 8.4. Эхо-сервер на основе модели перекрытого ввода-вывода	141
Пример 8.5. Эхо-сервер на основе модели перекрытого ввода-вывода с использованием процедур завершения	151
Пример 8.6. Эхо-сервер на основе модели портов завершения	162
Пример 8.7. Эхо-сервер на основе модели перекрытого ввода-вывода с использованием AcceptEx().....	171
Пример 8.8. Эхо-сервер на основе модели перекрытого ввода-вывода с использованием процедур завершения и функции AcceptEx()	182
Многоадресная рассылка в сетях IP	193
Пример 11.1. Многоадресная рассылка в сетях IP с использованием Winsock 1	195
Пример 11.2. Организация многоадресной рассылки в сетях IP средствами Winsock 2.....	203
Список литературы	211

Учебное издание

Васильчиков Владимир Васильевич

**Разработка
сетевых приложений
для ОС Windows
(практические примеры)**

Учебное пособие

Редактор, корректор И. В. Бунакова
Компьютерная верстка В. В. Васильчикова

Подписано в печать 22.06.09. Формат 60×84 ¹/₁₆.
Бум. офсетная. Гарнитура "Times New Roman".
Усл. печ. л. 13,5. Уч.-изд. л. 7,45.
Тираж 100 экз. Заказ .

Оригинал-макет подготовлен в редакционно-издательском отделе
Ярославского государственного университета им. П. Г. Демидова.
150000, Ярославль, ул. Советская, 14.

Отпечатано на ризографе.

