Предисловие

В этой статье вашему вниманию предлагаются несколько классов, которые позволяют быстро и безболезненно создавать системные службы (также называемые сервисами) Windows 2000. Так как в классах используются новые функции, появившиеся только в Windows 2000, вы не сможете без некоторых изменений использовать их для написания служб Window NT 4.0. Описанные в статье классы и методы работы со службами основаны на книге Дж. Рихтера и Дж. Кларка "Программирование серверных приложений для Windows 2000". В частности, класс CIOCP является точной копией одноименного класса из этой книги. Предполагается, что читатель уже имеет некоторый опыт создания системных служб и знаком с основными функциями для работы со службами. Более подробную информацию о работе системных служб, их взаимодействии со SCM, а также установке служб можно найти в источниках, приведенных в списке литературы.

Обзор классов

Минимальная рабочая служба

Для начала рассмотрим код, содержащий две минимальные (т.е. ничего не делающие, но корректно работающие) службы. Затем мы изучим используемые в программе классы более подробно.

```
void WINAPI ServiceMain1(DWORD dwArgc, PWSTR* pszArgv)
{
    CServiceContext ctx;                        // Создаем контекст службы
    ctx.fOwnProcess = false;                    // Служб в процессе две
    ctx.fInteractWithDesktop = false;           // С рабочим столом не взаимодействуем
    ctx.m_szServiceName = L"SServ1";           // Имя службы
    CServiceHelper::ServiceMainHelper(&ctx);  // Вызываем функцию, которая делает ВСЕ
}

void WINAPI ServiceMain2(DWORD dwArgc, PWSTR* pszArgv)
{
    CServiceContext ctx;
    ctx.fOwnProcess = false;
    ctx.fInteractWithDesktop = false;
    ctx.m_szServiceName = L"SServ2";
    CServiceHelper::ServiceMainHelper(&ctx);
}

int APIENTRY wWinMain(HINSTANCE hInstance,
                      HINSTANCE hPrevInstance,
                      LPSTR     lpCmdLine,
                      int       nCmdShow)
{
    int nArgc;
    LPWSTR* ppArgv = CommandLineToArgvW(GetCommandLine(), &nArgc);
    if (nArgc < 2)
    {
    // Создаем объект-помошник для запуска и управления службами
    CServiceHelper serv;

    // Добавление службы с указанием ее имени и процедуры
    // ServiceMain
    serv.AddService(L"SServ1", ServiceMain1);
    serv.AddService(L"SServ2", ServiceMain2);

    // Вызов StartServiceCtrlDispatcher()
    serv.Start();

    // Все службы остановлены, выход из программы
    return 0;
    }

    for (int i = 1; i < nArgc; i++)
```

```
{
if ((ppArgv[i][0] == L'-') || (ppArgv[i][0] == L'/'))
{
    wchar_t buf[100];
    if (!lstrcmpi(&ppArgv[i][1], L"install"))
    {
    CSCManager scm;

    // Открываем SCM для создания службы
    if (!scm.Open(SC_MANAGER_CREATE_SERVICE))
    {
        int err = GetLastError();
        swprintf(buf, L"Не могу открыть SCManager\nКод ошибки: %X", err);
        MessageBox(NULL, buf, L"Error", 0);
        return 1;
    }

    // Создаем службу №1
    if (!scm.CreateService(L"SServ1", L"SimpleService1",
                    L"Простой сервис номер 1", false))
    {
        int err = GetLastError();
        swprintf(buf, L"Не могу создать службу %s\nКод ошибки: %X",
                 L"SimpleService1", err);
        MessageBox(NULL, buf, L"Error", 0);
        return 1;
    }

    // Создаем службу №2
    if (!scm.CreateService(L"SServ2", L"SimpleService2",
        L"Простой сервис номер 2",false))
    {
        int err = GetLastError();
        swprintf(buf, L"Не могу создать службу %s\nКод ошибки: %X",
            L"SimpleService2", err);
        MessageBox(NULL, buf, L"Error", 0);
        return 1;
    }

    MessageBox(NULL,L"Служба успешно инсталирована",L"OK",0);
    }
    else if (!lstrcmpi(&ppArgv[i][1], L"remove"))
        {

    CSCManager scm;

            // Открываем SCM для удаления службы
    if (!scm.Open(SC_MANAGER_ALL_ACCESS))
            {
        int err = GetLastError();
        swprintf(buf, L"Не могу открыть SCManager\nКод ошибки: %X", err);
        MessageBox(NULL, buf, L"Error", 0);
        return 1;
    }

            // Удаляем службу №1
    if (!scm.DeleteService(L"SServ1"))
            {
        int err = GetLastError();
        swprintf(buf, L"Не могу удалить службу %s\nКод ошибки: %X",
             L"SimpleService1", err);
        MessageBox(NULL, buf, L"Error", 0);
        return 1;
    }

            // Удаляем службу №2
    if (!scm.DeleteService(L"SServ2"))
```

```
                {
            int err = GetLastError();
            swprintf(buf, L"Не могу удалить службу %s\nКод ошибки: %X",
                 L"SimpleService2", err);
            MessageBox(NULL, buf, L"Error", 0);
            return 1;
        }

        MessageBox(NULL, L"Служба успешно удалена", L"OK", 0);
        }
        else
            {
        MessageBox(NULL, L"Неизвестная команда", L"Error", 0);
        }
    }
    }
    return 0;
}
```

Как видите, все не так и сложно. Большую часть кода занимает обработка ошибок и вывод сообщений на экран. Сам запуск службы занимает четыре(!) строчки. Разберемся, как все это работает.

Class CServiceHelper

Этот класс - один из самых простых, но, тем не менее, и один из самых важных. Рассмотрим ключевую функцию Start(), в которой происходит соединение со SCM (Service Control Manager, менеджер системных служб). Она вызывается один раз при старте процесса, регистрирует имеющиеся в процессе службы и возвращает управление только тогда, когда все службы процесса остановлены.

```
BOOL CServiceHelper::Start()
{
    // m_services - это CSimpleMap, хранит имя службы
    // и адрес ее ServiceMain
    int l = m_services.GetSize();

    // создание массива структур SERVICE_TABLE_ENTRY
    SERVICE_TABLE_ENTRY* pServiceTable =
        new SERVICE_TABLE_ENTRY[l+1];

    if (!pServiceTable) return FALSE;

    // заполнение массива
    for (int i = 0; i < l; i++)
    {
        // ключ - имя службы
        pServiceTable[i].lpServiceName = m_services.GetKeyAt(i);

        // значение - адрес ServiceMain
        pServiceTable[i].lpServiceProc = m_services.GetValueAt(i);
    }

    // последний элемент массива должен быть пустым
    pServiceTable[j].lpServiceName = NULL;
    pServiceTable[j].lpServiceProc = NULL;

    // Запуск служб
    BOOL fOk = StartServiceCtrlDispatcher(pServiceTable);

    delete[] pServiceTable;
    return fOk;
}
```

Функция Start динамически создает и настраивает массив структур SERVICE_TABLE_ENTRY на основе информации, находящейся в ассоциативном массиве m_services, который является стандартным контейнером библиотеки ATL.

ПРЕДУПРЕЖДЕНИЕ

Любая программа, использующая данный класс, должна включать файл atlbase.h (#include <atlbase.h>).

Добавление элементов в m_services производится функцией AddService(LPCTSTR szServiceName, PSERVICEMAIN ServMainProc). Тип PSERVICEMAIN объявлен как

```
typedef void (__stdcall *PSERVICEMAIN)(DWORD, LPTSTR*);
```

Вызовом StartServiceCtrlDispatcher(pServiceTable) процесс сообщает SCM о том, какие службы в нем содержатся, и соответствующие этим службам адреса функций ServiceMain. SCM по имени находит запускаемую службу и вызывает ее функцию ServiceMain.

ПРИМЕЧАНИЕ

ServiceMain запускается в отдельном потоке.

ServiceMain первым делом должна зарегистрировать процедуру обработки сообщений от SCM, которая называется HandlerEx.

ПРИМЕЧАНИЕ

Вообще-то она может называться как угодно, однако в MSDN ее легче искать именно по такому имени. :)

Делается это путем вызова функции RegisterServiceCtrlHandlerEx(), которой передается имя службы, адрес HandlerEx и произвольное значение, определенное пользователем. Зарегистрировав обработчик сообщений, служба должна организовать своеобразный цикл выборки сообщений (message pump), взаимодействуя каким-либо образом с HandlerEx. HandlerEx вызывается всякий раз, когда SCM посылает службе некоторое сообщение. В функцию HandlerEx передается код сообщения, значение, заданное пользователем при регистрации обработчика, и еще пара параметров. HandlerEx, выполняющаяся в главном потоке приложения, должна перенаправлять сообщения в ServiceMain. Приняв сообщение SERVICE_CONTROL_STOP, служба должна выйти из цикла обработки сообщений и завершиться. "Ну и где все это в нашей ServiceMain?" - спросите вы.

```
void WINAPI ServiceMain1(DWORD dwArgc, PWSTR* pszArgv)
{
    CServiceContext ctx;                        // Создаем контекст службы
    ctx.fOwnProcess = false;                    // Служб в процессе две
    ctx.fInteractWithDesktop = false;           // С рабочим столом не взаимодействуем
    ctx.m_szServiceName = L"MWServ1";           // Имя службы
    CServiceHelper::ServiceMainHelper(&ctx);    // Вызываем функцию, которая делает ВСЕ
}
```

Сейчас мы во всем разберемся!

Контекст службы представляет собой объект, хранящий настройки для каждой конкретной службы, такие как имя службы. В процедуре ServiceMain первым делом производится настройка этих параметров. Затем вызывается статическая функция ServiceMainHelper() класса CServiceHelper, которой передается указатель на контекст службы.

Функция ServiceMainHelper() реализует элементарный цикл выборки сообщений, используя порт завершения ввода/вывода.

```
void CServiceHelper::ServiceMainHelper(CServiceContext *pCtx)
{
    // Самое первое сообщение мы посылаем себе сами
    ULONG_PTR CompKey = CServiceContext::CK_SERVICECONTROL;

    // Сообщение SERVICE_CONTROL_CONTINUE аналогично SERVICE_CONTROL_START,
    // однако последней константы просто не определено, поэтому обходимся тем,
    // что есть!
    DWORD dwControl = SERVICE_CONTROL_CONTINUE;

    // Для порта завершения ввода/вывода
    OVERLAPPED* pov;

    //Регистрируем обработчик сообщений от SCM
    if (!pCtx->RegisterService())
        // Если вызов не удачный - выход
        return;

    do
    {
        // Порт завершения ввода/вывода может принимать
        // сообщения от разных объектов
    switch(CompKey)
        {
        //Сообщение пришло от SCM
    case CServiceContext::CK_SERVICECONTROL:
        switch(dwControl)
            {
        case SERVICE_CONTROL_CONTINUE:
                // Вызов виртуальной функции
        pCtx->OnServiceStart();
        break;
        case SERVICE_CONTROL_PAUSE:
                // Вызов виртуальной функции
        pCtx->OnServicePause();
        break;
        case SERVICE_CONTROL_STOP:
        case SERVICE_CONTROL_SHUTDOWN:
                // Вызов виртуальной функции
        pCtx->OnServiceStop();
        break;
        case SERVICE_CONTROL_PARAMCHANGE:
                // Вызов виртуальной функции
        pCtx->OnParamChange();
        break;
        }
            // Сообщаем SCM о своем конечном состоянии
        pCtx->ReportState();
        break;
    default:
                // Вызов виртуальной функции
        pCtx->OnUserEvent(CompKey, dwControl);
    }
    if (pCtx->dwCurrentState != SERVICE_STOPPED)
        {
            // Ожидаем сообщений через порт ввода/вывода
        pCtx->GetStatus(&CompKey, &dwControl, &pov);
    }
    else
            // Если текущее состояние SERVICE_STOPPED -
            // выход из процедуры (завершение службы)
           break;
    }
    while(true);
```

```
}
```

Снова мы видим тривиальный код. Исключение составляет вызов RegisterService() и работа с портом завершения ввода/вывода (I/O completion port, IOCP). Описание работы портов завершения ввода/вывода выходит за рамки данной статьи, подробно она рассмотрена в [4]

RegisterService() делает две полезные вещи:

- Вызывает RegisterServiceCtrlHandlerEx(m_szServiceName, CServiceHelper::HandlerEx, this) (наконец-то!).
- Создает порт завершения ввода/вывода.

Первый параметр при вызове RegisterServiceCtrlHandlerEx() - имя службы, второй - адрес статической функции HandlerEx, которая определена в классе CServiceHelper, и последний параметр - указатель на контекст службы, который передается в HandlerEx при каждом ее вызове SCM.

Итак, последнее, что нам осталось рассмотреть в классе CServiceHelper - это его функцию HandlerEx(). Она, используя контекст службы, обрабатывает сообщения от SCM.

```
DWORD WINAPI CServiceHelper::HandlerEx(DWORD dwControl, DWORD dwEventType,
                                       PVOID pvEventData, PVOID pvContext)
{
    // SCM передает указатель на контекст службы
    CServiceContext* pCtx = (CServiceContext*) pvContext;

    // Так как SCM может запросто послать приостановленной
    // службе запрос на приостановку, мы его пресекаем
    if (pCtx->dwCurrentState == SERVICE_PAUSED &&
    dwControl == SERVICE_CONTROL_PAUSE)
    return ERROR_INVALID_PARAMETER;

    // Так как SCM может запросто послать работающей
    // службе запрос на продолжение работы, мы его пресекаем
    if (pCtx->dwCurrentState == SERVICE_RUNNING &&
    dwControl == SERVICE_CONTROL_CONTINUE)
    return ERROR_INVALID_PARAMETER;

    // Обработка сообщения от SCM
    return pCtx->ProcessControlCode(dwControl);
}
```

Эта функция содержит только две примечательные вещи:

- Мы игнорируем параметр dwEventType и pvEventData.
- Обработкой управляющих кодов занимается контекст службы.

Ну а теперь самое время более подробно рассмотреть, что из себя представляет этот самый контекст службы.

Class CServiceContext

Итак, контекст службы - это объект, описывающий параметры службы. Он занимается обработкой сообщений от SCM и взаимодействием с ServiceMain.

Рассмотрим функцию ProcessControlCode().

```
DWORD CServiceContext::ProcessControlCode(DWORD dwCode)
{
    //Предполагаем худшее
    DWORD dwRetCode = ERROR_CALL_NOT_IMPLEMENTED;
```

```
    //Флаг посылки кода сообщения в ServiceMain
    bool fPost = false;

    switch (dwCode)
    {
    case SERVICE_CONTROL_STOP:
    case SERVICE_CONTROL_SHUTDOWN:
        // Сообщаем SCM, что мы начали остановку службы
    SetState(SERVICE_STOPPED, 2000);
        // Отправляем код в ServiceMain
    fPost = true;
    break;

    case SERVICE_CONTROL_PAUSE:
        //Если служба может приостанавливаться, она должна при
        // создании контекста установить флаг fAllowPauseCont
    if (fAllowPauseCont)
        {
        SetState(SERVICE_PAUSED, 2000);
        fPost = true;
    }
    break;

    case SERVICE_CONTROL_CONTINUE:
    if (fAllowPauseCont)
        {
            // Сообщаем SCM, что мы возобнавляем работу
        SetState(SERVICE_RUNNING, 2000);
        fPost = true;
    }
    break;

    // Специальное сообщение на которое нужно просто ответить
    case SERVICE_CONTROL_INTERROGATE:
        //Сообщаем SCM о своем текущем состоянии
    ReportStatus();
        // Этот код не отправляется в ServiceMain
    break;

    case SERVICE_CONTROL_PARAMCHANGE:
    if (fAllowParamChange)
            fPost = true;
    }

    if (fPost)
    {
        //Отправляем сообщение в ServiceMain
    if (!PostStatus(CK_SERVICECONTROL,dwCode))
        dwRetCode = ERROR_FUNCTION_FAILED;
    else
        dwRetCode = NO_ERROR;
    }
    return dwRetCode;
}
```

ReportStatus() просто вызывает SetServiceStatus(), докладывая SCM о своем текущем состоянии. SetState() изменяет текущее состояние службы. Если в параметре передается, например, SERVICE_PAUSED, то SetState() сообщает SCM о том, что служба начала приостановку, изменяя текущее состояние на SERVICE_PAUSE_PENDING. Затем, с помощью функции PostStatus(), которая вызывает PostQueuedCompletionStatus(), управляющий код ставится в очередь сообщений порта завершения ввода/вывода.

Поток службы тем временем ожидает сообщений, вызывая метод GetStatus, являющийся тонкой оберткой вокруг функции GetQueuedCompletionStatus. Как только управляющий код попадает в очередь сообщений порта ввода/вывода, GetStatus возвращает управление ServiceMainHelper, которая в свою очередь анализирует управляющий код и вызывает один из виртуальных методов

объекта контекста службы, о которых речь пойдет ниже. Обработка управляющего кода завершается вызовом функции ReportState, которая докладывает о новом состоянии службы. Эта функция аналогична SetState, но изменяет состояние службы в "обратную" сторону: например, если текущее состояние службы - SERVICE_STATUS_PENDING, то функция изменит его на SERVICE_PAUSED. После того, как новое состояние службы установлено, ServiceMainHelper переходит к ожиданию следующего сообщения, вновь вызывая GetStatus.

Остались еще две функции данного класса: RegisterService() и AcceptControls(). Рассмотрим их код.

```
BOOL CServiceContext::RegisterService()
{
    m_hss = RegisterServiceCtrlHandlerEx(m_szServiceName,
        CServiceHelper::HandlerEx, this);

    if (m_hss != NULL)
    {
    dwServiceType = fOwnProcess ? SERVICE_WIN32_OWN_PROCESS
                            : SERVICE_WIN32_SHARE_PROCESS;

    if (fInteractWithDesktop)
        dwServiceType |= SERVICE_INTERACTIVE_PROCESS;

        // Изначально находимся в состоянии "сейчас заведемся"
    dwCurrentState = SERVICE_START_PENDING;

        // По умолчанию принимаем от SCM только одно сообщение
    dwControlsAccepted = SERVICE_ACCEPT_STOP;

    dwWin32ExitCode = NO_ERROR;
    dwServiceSpecificExitCode = 0;
    dwCheckPoint = 0;

        // Предполагаемое время работы до следующего опроса
    dwWaitHint = 2000;

        //Предотвращаем возникновение "гонок"
    m_cs.Lock();

        // Создаем порт завершения
    return Create();
    }
    else
        return FALSE;
}
```

Первым делом мы регистрируем обработчик сообщений от SCM. Далее, если все нормально, мы настраиваем члены структуры SERVICE_STATUS, которая используется для передачи информации о состоянии функцией SetServiceStatus().

ПРИМЕЧАНИЕ

CServiceContext наследован от SERVICE_STATUS, как нетрудно догадаться. Также он является наследником от CIOCP, т.е. использует множественное наследование.

Примечательным в этой функции является вызов m_cs.Lock().m_cs - это закрытый член класса, являющийся критической секцией.

ПРЕДУПРЕЖДЕНИЕ

Критическая секция определена как CComAutoCriticalSection m_cs; Таким образом, в основной программе необходимо включить файл atlbase.h

Применение критической секции решает проблему "гонок" (race conditions). Она может возникнуть, когда приостанавливающейся службе посылается запрос на завершение. При этом вполне возможна такая последовательность состояний службы:

1. SERVICE_PAUSE_PENDING
2. SERVICE_STOP_PENDING
3. SERVICE_PAUSED
4. SERVICE_STOPPED

Решается проблема просто: при сообщении о начале действия (SetState) вызывается функция блокировки критической секции, т.е. EnterCriticalSection(). При сообщении о конечном состоянии (ReportState) вызывается функция разблокировки, т.е. LeaveCriticalSection(). Так как после вызова RegisterService() вызывается функция ReportState(), которая разблокирует критическую секцию, мы должны сначала ее заблокировать. Каждому вызову LeaveCriticalSection() должен предшествовать вызов EnterCriticalSection().

Теперь рассмотрим AcceptControls(). Она сообщает SCM о том, какие команды данная служба может обрабатывать.

```
void AcceptControls(DWORD dwFlags, bool fAccept = TRUE)
{
    if (fAccept)
        dwControlsAccepted |= dwFlags;
    else
    dwControlsAccepted &= ~dwFlags;

    if (dwControlsAccepted & SERVICE_ACCEPT_PAUSE_CONTINUE)
    fAllowPauseCont = fAccept;

    if (dwControlsAccepted & SERVICE_ACCEPT_PARAMCHANGE)
    fAllowParamChange = fAccept;
};
```

Функция изменяет поле dwControlsAccepted структуры SERVICE_STATUS и два закрытых флага класса, которые используются в ProcessControlCode().

Вот собственно и все, что касается запуска и функционирования служб. Следующий простой класс реализует функции создания и удаления службы из внутренней базы SCM.

Class CSCMaganer

Так как это очень простой класс, мы рассмотрим только его описание и функцию DeleteService().

```
class CSCManager
{
    //Public functions
public:
    BOOL Close();
    BOOL DeleteService(LPCTSTR szServiceName, bool fStop = true);
    BOOL CreateService(LPCTSTR szServiceName, LPCTSTR szDisplayName,
        LPTSTR szDescription, bool fOwnProcess = true);
    bool Open(DWORD dwAccess);

    CSCManager();
    ~CSCManager();

    TCHAR*    m_modulename;

    //private attributes
private:
    SC_HANDLE m_hScm;
};
```

Функция Open() открывает SCM и сохраняет полученный хэндл в закрытую переменную m_hScm. В конструкторе эта переменная обнуляется, а в деструкторе хэндл закрывается, если он не равен нулю.

CreateService() создает службу. Параметры szServiceName и szDisplayName обязательны, а для szDescription можно передать NULL. Функция Close() закрывает SCM и предназначена на тот случай, если вам будет необходимо заново открыть SCM с другими правами доступа.

Функция DeleteService() удаляет службу. Перед удалением служба останавливается, если флаг fStop установлен.

```
BOOL CSCManager::DeleteService(LPCTSTR szServiceName, bool fStop)
{
    _ASSERTE(szServiceName);

    if (!szServiceName)
    {
        _TRACE("Имя службы не задано");
        SetLastError(ERROR_INVALID_PARAMETER);
        return false;
    }

    // Открываем службу с необходимыми правами доступа
    SC_HANDLE hServ = OpenService(m_hScm, szServiceName,
                                  DELETE |( fStop ? SERVICE_STOP : 0));

    if (!hServ) return FALSE;

    if (fStop)
    {
    SERVICE_STATUS status;

        // Посылаем службе запрос на остановку
    if (!ControlService(hServ, SERVICE_CONTROL_STOP, &status))
        {
        int err = GetLastError();

            // Если служба уже остановлена или в процессе
            // остановки, то нужно ее просто удалить
        if (err != ERROR_SERVICE_NOT_ACTIVE &&
        err != ERROR_SHUTDOWN_IN_PROGRESS)
                // В противном случае - выходим из процедуры
        return FALSE;
    }
    else
            // Ждем указанное службой время
        Sleep(status.dwWaitHint);
    }

    //Удаление службы из БД SCM
    BOOL fOk = ::DeleteService(hServ);

    int err;
    if (!fOk) err = GetLastError();

    //Закрываем хэндл службы
    CloseServiceHandle(hServ);

    if (!fOk) SetLastError(err);
    return fOk;
}
```

Следует заметить, что код остановки службы не совсем корректный. Дело в том, что после вызова Sleep() с заданным интервалом нет никакой гарантии, что служба остановлена. Ее состояние можно проверить вызовом QueryServiceStatus(). Эта функция возвращает состояние службы в структуру

SERVICE_STATUS. Если текущее состояние службы не SERVICE_STOPPED, необходимо еще раз вызвать функцию Sleep().

ПРИМЕЧАНИЕ

Поле dwCheckPoint должно содержать значение большее предыдущего. Об этом должна заботиться служба. Если это значение меньше или равно прерыдущему - вероятнее всего служба не отвечает.

Так должно повторяться до тех пор, пока функция QueryServiceStatus() не вернет состояние SERVICE_STOPPED. Только после этого можно быть уверенным, что служба остановлена.

ПРИМЕЧАНИЕ

Остановка службы не означает выгрузку процесса, ее содержащего. Процесс завершается, когда остановлены все его службы. Но иногда даже в этом случае он продолжает "висеть" некоторое время, и лишь через несколько секунд исчезает.

На этом мы закончим обзор классов и их функций и займемся созданием рабочей службы. Под рабочей службой подразумевается такая служба, которая реагирует на сообщения SCM, корректно запускается и останавливается. Все работа службы будет заключаться в выводе двух сообщений о запуске и остановке службы. Для вывода сообщений используется функция MessageBox() с флагом MB_SERVICE_NOTIFICATION. Этот флаг позволяет выводить диалоговое окно на активный рабочий стол, даже если службе не разрешено взаимодействие с рабочим столом.

ПРЕДУПРЕЖДЕНИЕ

Никогда не используйте такой прием в рабочих службах и не устанавливайте флаг SERVICE_INTERACTIVE_PROCESS. Более подробную информацию по этому поводу можно получить из источников, указанных в списке литературы.

Создание рабочей службы

Для того чтобы реагировать на сообщения SCM, нужно создать дочерний класс от CServiceContext и переопределить следующие виртуальные функции:

- OnServiceStart() - вызывается при старте службы;
- OnServiceStop() - вызывается при остановке службы;
- OnServicePause() - вызывается при приостановке службы;
- OnParamChange() - вызывается при изменении параметров системы;
- OnUserEvent(DWORD dwEventCode, DWORD dwSubCode) - вызывается, когда принято пользовательское сообщение.

В классе CServiceContext они объявлены как виртуальные. Мы переопределим только две из них.

```
void CMWContext::OnServiceStart()
{
    MessageBox(NULL, L"Start", L"Simple service", MB_SERVICE_NOTIFICATION);
}

void CMWContext::OnServiceStop()
{
    MessageBox(NULL, L"Stop", L"Simple service", MB_SERVICE_NOTIFICATION);
}
```

Теперь рассмотрим код wWinMain.

```
CMWContext mwctx;
mwctx.fInteractWithDesktop = false;
mwctx.m_szServiceName = L"SServ";

// Служба в процессе одна
mwctx.fOwnProcess = true;

CServiceHelper serv;

// Можно не определять глобальную ServiceMain!
serv.SetServiceCtx(&mwctx);

BOOL fOk = serv.Start();
```

Для нас здесь представляет интерес только функция SetServiceCtx(). Дело в том, что кроме всего прочего CServiceHelper предоставляет статическую функцию ServiceMain. Она, используя закрытую переменную типа CServiceContext, вызывает функцию ServiceMainHelper, с которой мы уже знакомы. Давайте, чтобы не было вопросов, рассмотрим код функции SetServiceCtx().

```
bool SetServiceCtx(CServiceContext* pCtx)
{
    bool fOk = true;

    // Назначаем контекст
    m_servctx = pCtx;

    //Добавляем сервис
    if (AddService(pCtx->m_szServiceName,ServiceMain))
        // Устанавливаем флаг о количестве служб
    m_ServiceCnt = ONESERVICE;
    else
    fOk = false;
    return fOk;
};
```

Функция AddService() добавляет службу, если значение m_ServiceCnt не равно ONESERVICE. Это делается для того, чтобы нельзя было установить две службы имеющие одинаковые ServiceMain. В качестве второго параметра AddService() принимает адрес закрытой статической функции класса CServiceHelper, которая реализована очень просто.

```
void WINAPI CServiceHelper::ServiceMain(DWORD dwArgc, LPTSTR* pszArgv)
{
    _ASSERTE(m_servctx != NULL);

    if (m_servctx == NULL)
    {
    _TRACE("Service context is not specified");
    return;
    }
    ServiceMainHelper(m_servctx);
}
```

Вот собственно и все. Следует также добавить, что в производном от CServiceContext классе можно переопределить функцию ProcessControlCode(), которая тоже является виртуальной. Однако, трудно представить себе ситуацию когда это может понадобиться.

Заключение

В заключение необходимо отметить, что приведенный код не является избыточным. Он практически не содержит ненужных (не вызываемых) функций. С помощью представленных классов можно в течение 5-10 минут создать службу, выполняющую вполне определенную задачу, умеющую себя регистрировать и удалять, что является неплохим показателем для WinAPI-программирования.

Литература

1. Сергей Холодилов, *Программирование служб: подробности*.
2. Александр Федотов, *Управление системными службами Windows NT. Часть 1*.
3. Александр Федотов, *Управление системными службами Windows NT. Часть 2*.
4. Джеффри Рихтер, Джейсон Кларк, *Программирование серверных приложений для Windows 2000*. Издательства "Русская Редакция", "Питер", 2001.