



DATENTRANSFER ZWISCHEN WINDOWS GERÄTEN MITTELS IRDA

Was wäre wohl ein gemütlicher Fernsehabend ohne Infrarot-Fernbedienung? Ein ständiges Aufstehen, um den Ton lauter oder leiser zu stellen oder zwischen den unzähligen Programmen zu wechseln. Doch nicht mehr alles ganz so gemütlich, oder?

Aber nicht nur die Infrarot-Fernbedienung hält Einzug in unser tägliches Leben: Immer mehr Geräte mit Infrarotunterstützung kommen auf den Markt. So ist eine Infrarotschnittstelle bei Notebooks oder Handys heute schon fast Standard. Auch unter den Pocket PCs ist der Datentransfer zwischen zwei Geräten mittels IrDA inzwischen zum Standard geworden. Doch was nützt die Infrarotschnittstelle, wenn sie von der Software nicht unterstützt wird?

Wie einfach Sie eine Infrarotunterstützung in Ihre Windows-Anwendungen implementieren, soll der folgende Artikel zeigen. Zur Veranschaulichung wurde eine kleine Server/Client-Beispielanwendung erstellt, mit deren Hilfe Sie Nachrichten von Ihrer Pocket-PC-Anwendung an eine normale Windows-Anwendung versenden können. Dieses Beispiel können Sie als Basis für ihre eigenen Anwendungen verwenden, sodass sich recht einfach ganze Dateien oder Messdaten zwischen verschiedenen Geräten transferieren lassen.

Die Server/Client-Anwendung besteht aus zwei Teilen, der Server-Applikation, die auf Nachrichten wartet, und der Client-Applikation, die Nachrichten an den Server sendet. Beide Applikationen sind recht einfach und schlicht gehalten, um sich auf das eigentliche Thema des Artikels, dem **„Datentransfer zwischen Windows Geräten mittels IrDA“** zu konzentrieren.

Um die beiden Applikationen erstellen zu können, benötigen Sie zum einen die eMbedded Visual Tools 3.0 und zum anderen das Visual Studio 6.0 oder höher. Beide Produkte sind aus dem Hause Microsoft, wobei Sie die eMbedded Visual Tools 3.0 kostenlos von der Microsoft Homepage laden können. Die eMbedded Visual Tools werden für die Client-Anwendung benötigt, die auf einem Pocket PC läuft. Für die Server-Anwendung, die auf einem herkömmlichen Windows-System läuft, wird das Microsoft Visual Studio 6.0 oder höher benötigt.

Nun aber zuerst ein wenig allgemeines zum Thema **„Datentransfer mittels IrDA“**.

Was ist eigentlich IrDA?

IrDA steht für Infrared Data Association – eine Organisation von über 150 Firmen, die einen gemeinsamen Standard für die infrarote Kommunikation zwischen verschiedenen Geräten erstellt hat. Der IrDA-Standard beinhaltet alle wesentlichen Spezifikationen für die kabellose Datenübertragung per Infrarot. Dadurch wird eine größtmögliche Kompatibilität zwischen Geräten verschiedener Hersteller erreicht. Die IrDA hat bisher eine Sammlung von Standards herausgegeben, die die physikalischen und softwarespezifischen Schichten definieren, die für die reibungslose Kommunikation zwischen zwei Infrarotgeräten notwendig sind.

Inzwischen wird die Infrarot-Schnittstelle vieler Geräte auch IrDA-Schnittstelle genannt. Der Aufbau dieses IrDA-Stacks ist in der folgenden Abbildung (siehe Abb. 1) zu sehen.

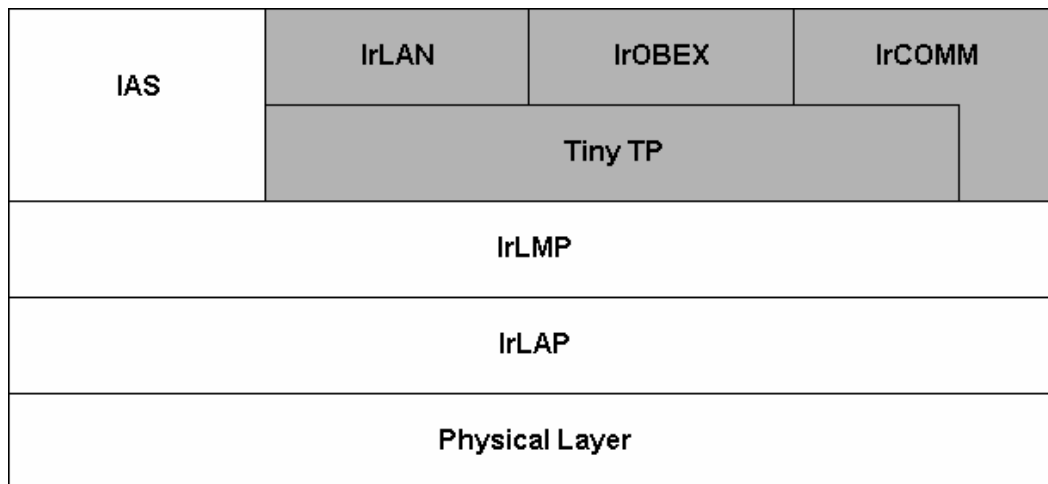


Abb. 1: IrDA-Stack

Die einzelnen Schichten dieses IrDA-Stacks werden nur kurz und sehr einfach beschreiben, da eine detaillierte Beschreibung all dieser Schichten bzw. Protokolle den Rahmen dieses Artikels sprengen würde.

Physical Layer: Spezifiziert optische Eigenschaften, Codierung der Daten und Rahmengröße für verschiedene Geschwindigkeiten.

IrLAP (Link Access Protocol): Stellt eine zuverlässige Verbindung zwischen zwei Infrarotgeräten her.

IrLMP (Link Management Protocol): Stellt Multiplexed-Dienste über das IrLAP zur Verfügung, d.h. es bietet zum Beispiel verschiedenen parallel laufenden Anwendungen die Möglichkeit unabhängige Verbindungen aufzubauen.

IAS (Information Access Service): Stellt die Informationen über die vorhandenen Dienste eines Gerätes zur Verfügung.

Tiny TP: Stellt eine Art Flusskontrolle für einen Kanal zur Verfügung.

IrOBEX: Definiert den einfachen Transfer von Dateien und anderen Datenobjekten zwischen zwei Geräten.

IrCOMM: Seriell- und Parallelportemulation, die es bestehenden Anwendungen, die serielle oder parallele Kommunikation nutzen, ermöglicht Infrarot ohne Softwareänderungen verwenden zu können.

IrLAN (Local Area Network): Beschreibt ein Protokoll zur drahtlosen Anbindung an lokale Netze mittels Infrarot-Technik.

Ein detailliertes Wissen über den Aufbau dieser Schichten ist erforderlich, falls Sie sich genauer mit der IrDA Programmierung auseinandersetzen wollen. Informationen

hierzu finden Sie beispielsweise auf der offiziellen Homepage der IrDA (<http://www.irda.org>).

IrDA Programmierung unter Windows

Unter Windows lässt sich die Infrarotschnittstelle ganz einfach mit Hilfe der IrSock-Erweiterung des vorhandenen WinSock2-Standards ansprechen. Bei der IrSock-Erweiterung handelt es sich genaugenommen um keine Erweiterung des bestehenden WinSock2-Standards, sondern nur um die Unterstützung für TinyTP/IrLMP als weiteres Transportprotokoll, das wie TCP/IP über Sockets angesteuert werden kann. Sie als Programmierer arbeiten also nur mit den herkömmlichen Socket-Funktionen, wie es zum Beispiel von der TCP/IP-Programmierung her bekannt sein dürfte. Wie Ihre Anwendung über das WinSock-Interface auf die IrDA-Schnittstelle zugreift, ist der nachfolgenden Abbildung (siehe Abb. 2) zu entnehmen.

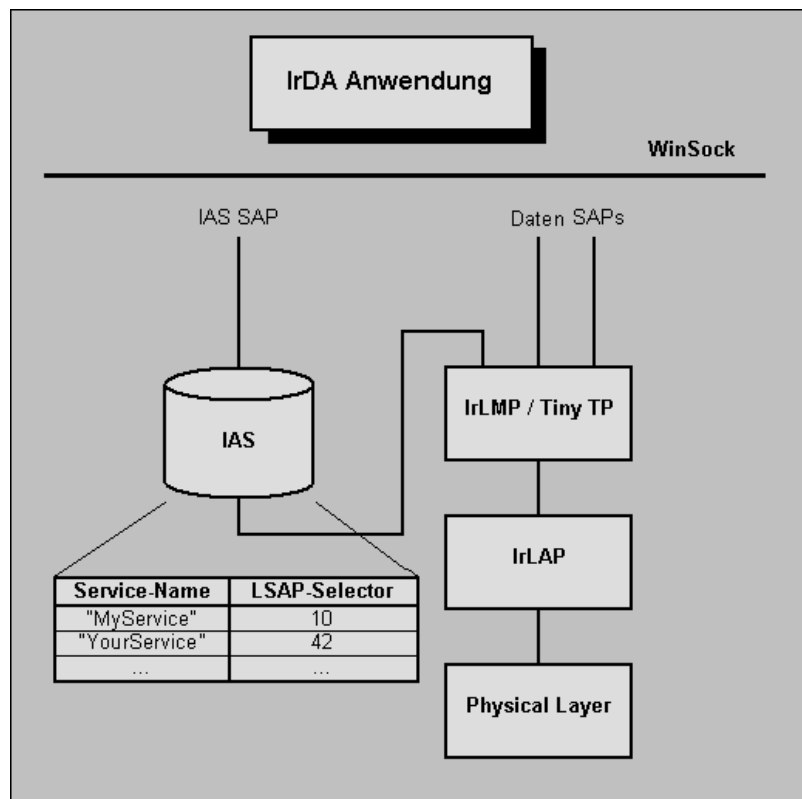


Abb. 2: Zugriff mittels WinSock auf die IrDA-Schnittstelle

Allerdings bestehen einige kleine Unterschiede bei der Socket-Programmierung der Infrarotschnittstelle. So mussten zum Beispiel einige Funktionen wegen der speziellen Eigenschaften der Infrarotkommunikation (wie fehlende feste Adressen, kurzfristig bestehende Verbindungen) angepasst werden. Einige Socket-Funktionen werden deshalb bei der Infrarotkommunikation nicht unterstützt, wie zum Beispiel die „gethostbyname“-Funktion. Außerdem gibt es bei der Infrarotkommunikation noch einige Einschränkungen bzw. Änderungen verglichen mit der herkömmlichen Socket-Programmierung, wie zum Beispiel bei TCP/IP. IrSockets können nur als Stream-Sockets arbeiten, d.h. sie funktionieren nicht als Datagramm-Sockets.

Auch die Adressierung der IrDA-Geräte wurde speziell auf die Infrarotverbindung angepasst. Statt der bei der TCP/IP-Programmierung verwendeten TCP-Port- und

IP-Nummer kommen bei der Infrarotverbindung nun neben der Device-ID noch ein sogenannter LSAP-Selector und ein Service-Name zum Einsatz. Der LSAP-Selector kann mit der Portnummer unter TCP/IP verglichen werden und entspricht momentan einer Zahl zwischen 0 und 127. Will man ein IrDA-Gerät ansprechen, so kann man entweder einen gewünschten LSAP-Selector selbst auswählen oder einen Service-Namen angeben, wobei dann automatisch ein noch freier LSAP-Selector ausgewählt wird.

Bei der einfachen Implementierung der Infrarotverbindung reicht aber zu wissen, dass jede IrDA-Server-Applikation, die eine Infrarotverbindung bereitstellt, einen solchen LSAP-Selector oder wie in unserem Beispiel einen Service-Namen bereitstellt und jede Client-Anwendung über diese Art der Adressierung versucht, eine Verbindung zur Server-Applikation aufzubauen. Bis auf diese Einschränkungen bzw. Änderungen können alle bekannten Socket-Funktionen, wie zum Beispiel „send“ oder „recv“, in gewöhnlicher Art und Weise verwendet werden.

Durch diese Art der Adressierung bei der Infrarotverbindung lassen sich mehrere Anwendungen über ein und denselben Infrarotport betreiben, was auch in der folgenden Abbildung (siehe Abb. 3) ersichtlich ist.

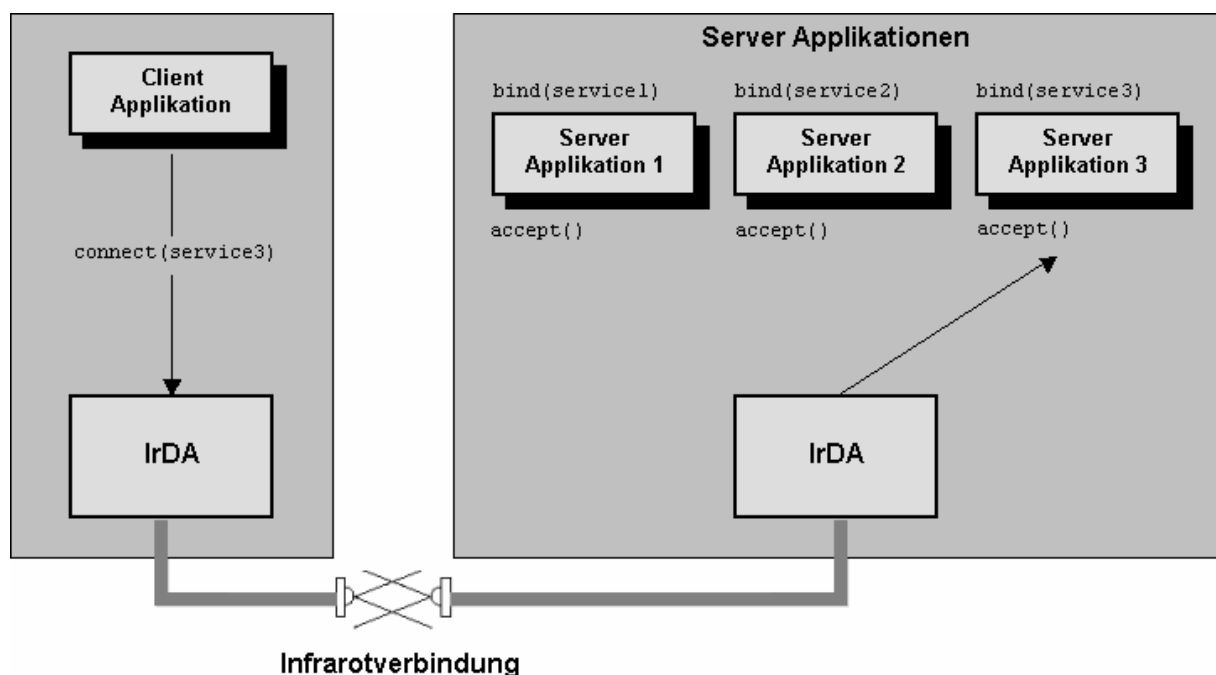


Abb. 3: Adressierung Infrarotverbindung

Die Server-Applikation

Wie schon vorhin erwähnt, benötigen Sie für die Server-Applikation das Visual Studio 6.0 oder höher. Bei dieser Applikation handelt es sich um eine dialogfeldbasierte Anwendung, die mit Hilfe des MFC-Frameworks (Microsoft Foundation Classes) erstellt wurde.

Das User-Interface des Server-Dialogs (siehe Abb. 4) wurde sehr einfach gehalten. Als wichtigstes Element neben dem „Exit“-Button ist hier das „Eingabefeld“ zu erwähnen, das die empfangenen Nachrichten der Client-Anwendung anzeigt.

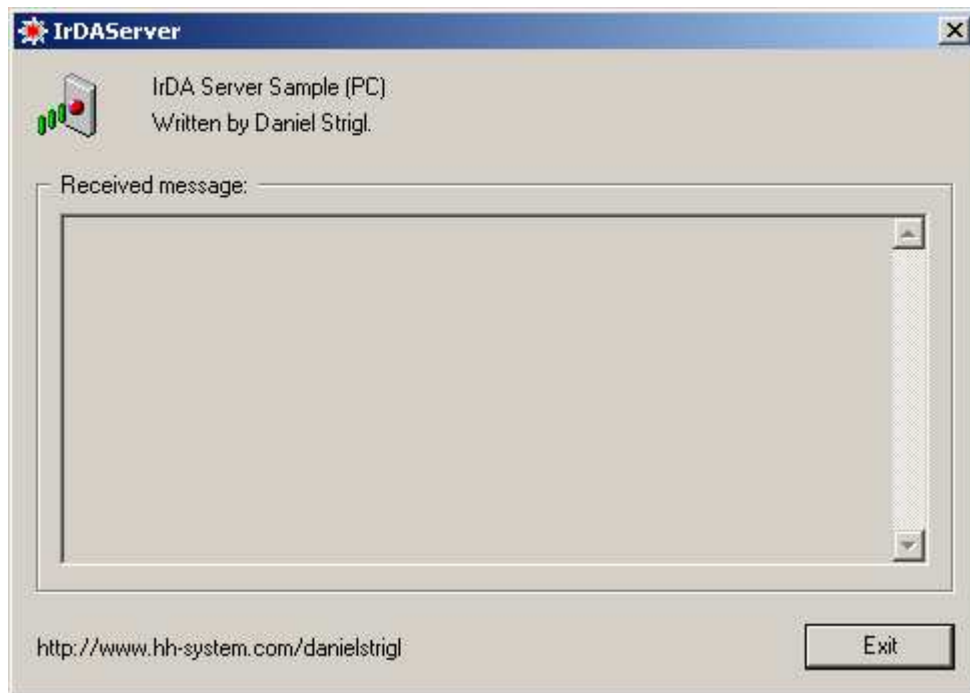


Abb. 4: Screenshot der Server-Applikation

Um die Socket-Funktionalität in Ihrem Projekt nutzen zu können, linken Sie die Library „Ws2_32.lib“ zu ihrem Projekt und binden die dazugehörige Include-Datei „af_irda.h“ ein. Um dies zu bewerkstelligen öffnen Sie zuerst den Dialog „Projekteinstellungen“ über den Menüpunkt „Projekt“ und „Einstellungen“. Anschließend wählen Sie den Reiter „Linker“ aus und selektieren im Auswahlfenster „Kategorie“ den Eintrag „Allgemein“. Sie fügen den Text „Ws2_32.lib“ im „Objekt-/Bibliothek-Module“-Eingabefeld des Dialogs ein und schließen den Dialog mittels des „OK“-Buttons. Die folgende Abbildung (siehe Abb. 5) zeigt Ihnen, wie es funktioniert.

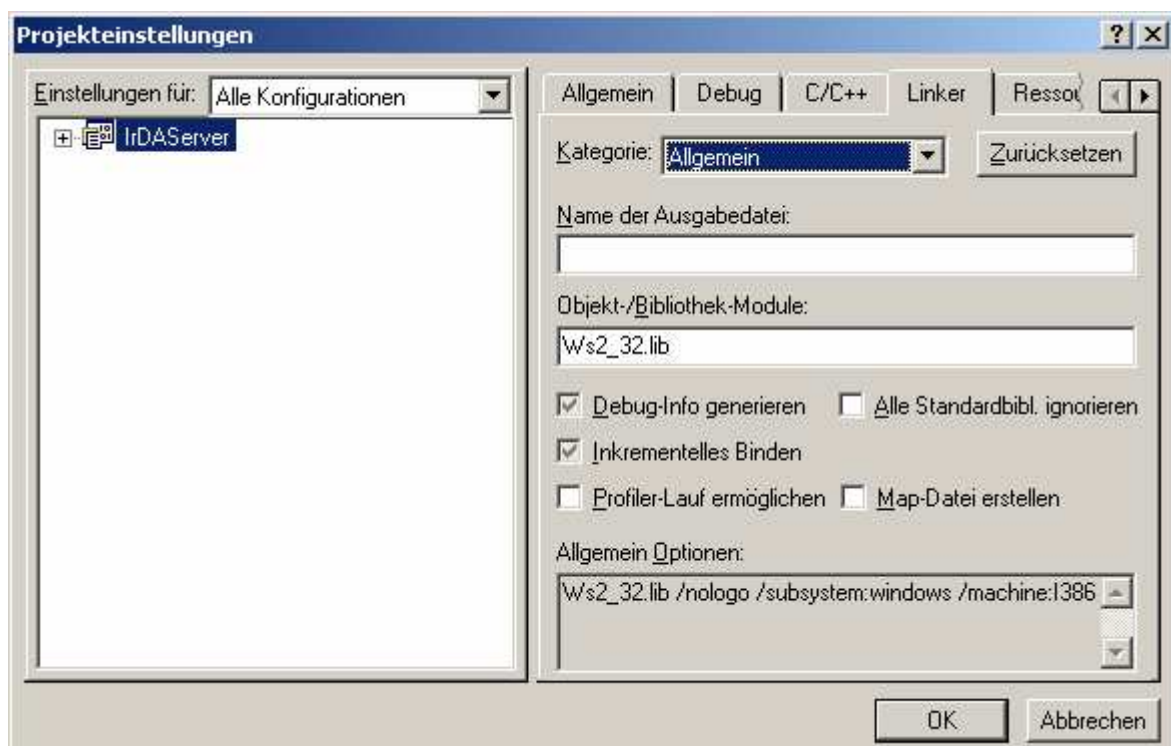


Abb. 5: Screenshot der Visual C++ Projekteinstellungen

Bevor Sie jedoch nun eine Socket-Funktion nutzen können, müssen Sie zuerst die WinSock-DLL initialisieren, was mittels der Funktion „WSAStartup“ funktioniert. Dieser Funktion übergeben Sie nun die Version der Windows-Sockets, die Sie benötigen. Wir übergeben hier der Funktion die Versionsnummer 1.1. Die Funktion liefert Ihnen bei Erfolg eine detaillierte Beschreibung der Windows-Sockets-Implementierung in der Struktur „WSADATA“ zurück. Bei Programmende rufen Sie noch die zu „WSAStartup“ komplementäre Funktion „WSACleanup“ auf, welche alle noch offenen Ressourcen freigibt und sich anschließend ordnungsgemäß abmeldet.

Da die Socket-Funktionen standardmäßig im blockierenden Modus arbeiten, wird für das Warten bzw. Empfangen von Client-Nachrichten ein eigener Hintergrundtask erzeugt. Blockierend heißt hier zum Beispiel, dass die „recv“-Funktion erst zurückkehrt, wenn Daten vom Socket eingelesen wurden. Selbstverständlich lassen sich die Socket-Funktionen auch mittels der Funktionen „WSAAsyncSelect“ (funktioniert nicht unter Windows CE) und „ioctlsocket“ in den nicht-blockierenden Modus umschalten, jedoch soll hier der Weg über einen eigenen Hintergrundtask für den Datenempfang gezeigt werden. Wie das Versenden von Nachrichten im nicht-blockierenden Modus funktioniert, sehen Sie später anhand der Client-Applikation.

Wie schon vorhin erwähnt, wird nun beim Programmstart innerhalb der „OnInitDialog“ Funktion zuerst die WinSock-DLL initialisiert und anschließend der Hintergrundtask „WaitForMessage“ erstellt und auch gleich gestartet.

```
// ZU ERLEDIGEN: Hier zusätzliche Initialisierung einfügen

WSADATA wsaData;
VERIFY(WSAStartup(MAKEWORD(1,1), &wsaData) == 0);

// Start the thread for the IrDA messages
m_pWaitForMessageThread = AfxBeginThread(WaitForMessage, this);
ASSERT_VALID(m_pWaitForMessageThread);
```

Als nächstes wollen wir uns nun den Hintergrundtask, der für die Entgegennahme der Client-Nachrichten zuständig ist, etwas genauer ansehen.

Innerhalb der „WaitForMessage“-Funktion (dem Hintergrundtask) wird nun der Socket erzeugt, der auf eingehende Client-Verbindungen wartet. Dieser Socket wird mittels der Funktion „socket“ erzeugt. Der erste Parameter der „socket“-Funktion gibt hierbei die Adressfamilie an, die bei der IrDA-Verbindung der Konstanten „AF_IRDA“ entspricht. Der zweite Parameter gibt den Typ der Verbindung an, wobei hier nur die Konstante „SOCK_STREAM“ in Frage kommt, da wie schon vorhin erwähnt bei der IrDA-Verbindung nur Stream-Sockets unterstützt werden.

Anschließend wird dieser Socket mittels dem „bind“-Kommando mit unserem Service-Namen „MySampleIrDAService“ verbunden und durch den „listen“-Befehl in den Zustand versetzt, auf eingehende Verbindungen zu horchen. Der „listen“-Befehl erwartet neben dem Socket als zweiten Parameter die Länge der Liste mit den anstehenden Verbindungen, d.h. es kann maximal diese Anzahl an Verbindungen mittels dem „accept“-Kommando angenommen werden. Alle anderen Verbindungsversuche werden dabei vom Server abgewiesen. Da wir nur einen Client zulassen bzw. nur auf eine Client-Verbindung vorbereitet sind, übergeben wir hier den Wert „1“.

```

// Create the server socket
m_serverSocket = socket(AF_IRDA, SOCK_STREAM, 0);
if (m_serverSocket == INVALID_SOCKET)
{
    return;
}

// Associate the server service with the socket
SOCKADDR_IRDA serverSocketAddr = { AF_IRDA, 0, 0, 0, 0,
                                   "MySampleIrDAService" };
if (bind(m_serverSocket,
         reinterpret_cast<struct sockaddr*>(&serverSocketAddr),
         sizeof(serverSocketAddr)) == SOCKET_ERROR)
{
    closesocket(m_serverSocket);
    return;
}

// Listen for an incoming connection
if (listen(m_serverSocket, 1) == SOCKET_ERROR)
{
    closesocket(m_serverSocket);
    return;
}

```

Als nächstes wird darauf gewartet, dass ein Client versucht, sich beim Server anzumelden, was mittels der „accept“-Funktion bewerkstelligt wird. Kommt eine Verbindung zu Stande, liefert die „accept“-Funktion einen Socket zurück, der diese Server-Client-Verbindung beschreibt.

Bevor nun die eigentliche Nachricht vom Client eingelesen wird, ermitteln wir die Länge der Nachricht, die uns der Client zukommen lässt. Dies ist sehr nützlich, um Speicher für die Nachricht zu reservieren und anschließend beim Einlesen der eigentlichen Nachricht zu wissen, wie viel Bytes einzulesen sind. Der Client sendet uns dazu zuerst die Länge der Nachricht in Form einer 32-Bit-Zahl und anschließend die eigentliche Nachricht. Beim Ermitteln der Nachrichtenlänge ist darauf zu achten, dass die eingelesene 32-Bit-Zahl auch die richtige „Byte Order“ (Byte-Reihenfolge) hat. Um sicherzustellen, dass die 32-Bit-Zahl auf beiden Systemen gleich interpretiert wird, werden die Socket-Funktionen „htonl“ und „ntohl“ verwendet. Diese beiden Funktionen konvertieren eine 32-Bit-Zahl in eine sogenannte „TCP/IP Network Byte Order“ und wieder zurück. Der Client konvertiert hierbei die Länge der Nachricht in die „TCP/IP Network Byte Order“ und sendet diese konvertierte Zahl anschließend zum Server. Der Server konvertiert die empfangene Zahl anschließend seinerseits von der „TCP/IP Network Byte Order“ in die „Host Byte Order“ (Byte-Reihenfolge des Target-Systems), wodurch sichergestellt ist, dass beide Systeme diese Zahl gleich interpretieren.

```

m_clientSocket = accept(m_serverSocket, 0, 0);
if (m_bWaitForMessage == FALSE && m_clientSocket == INVALID_SOCKET)
{
    break;
}
else if (m_clientSocket == INVALID_SOCKET)
{
    continue;
}

// Receive the size of the message

```



```

const UINT CIrDAServerDlg::MY_WM_MSG_RECEIVED =
    RegisterWindowMessage(_T("MY_WM_MSG_RECEIVED-{A0422CA6-D03D-466c-A1DC-
        0DF8D4E89386}"));

BEGIN_MESSAGE_MAP(CIrDAServerDlg, CDialog)
   //{{AFX_MSG_MAP(CIrDAServerDlg)
    ON_WM_SYSCOMMAND()
    ON_WM_PAINT()
    ON_WM_QUERYDRAGICON()
    ON_WM_DESTROY()
    //}}AFX_MSG_MAP
    ON_REGISTERED_MESSAGE(MY_WM_MSG_RECEIVED, OnMyMessage)
END_MESSAGE_MAP()

////////////////////////////////////
//
// Benutzerdefinierter Nachrichtenhandler

LRESULT CIrDAServerDlg::OnMyMessage(WPARAM wParam, LPARAM lParam)
{
    LPSTR pszMessage = reinterpret_cast<LPSTR>(wParam);
    ASSERT(AfxIsValidString(pszMessage));

    // Display the received message
    m_strMessage = pszMessage;
    UpdateData(FALSE);

    MessageBeep(MB_ICONASTERISK);

    return 0L;
}

```

Hiermit wäre die eigentliche Arbeit der Server-Applikation beschrieben, nur noch ein paar Zeilen Code „Aufräumarbeit“ nach Beendigung der Applikation sind zu implementieren.

Wird die Server-Applikation über den „Exit“- oder „Schließen“-Button des Fensters beendet, wird das Fenster zerstört und die „OnDestroy“-Routine des Fensters aufgerufen. Innerhalb dieser Funktion können wir nun unsere „Aufräumarbeit“ erledigen, d.h. es werden noch alle geöffneten Socket-Verbindungen geschlossen und der Hintergrundtask beendet bzw. darauf gewartet, bis dieser beendet wird.

Zuerst wird mittels der Variable „m_bWaitForMessage“ geprüft, ob wir überhaupt etwas aufzuräumen haben. Besitzt diese Variable den Wert „FALSE“, existieren keine offenen Socket-Verbindungen, und auch der Hintergrundtask wurde schon beendet.

Besitzt die „m_bWaitForMessage“-Variable allerdings den Wert „TRUE“, so läuft der Hintergrundtask noch und es bestehen noch geöffnete Socket-Verbindungen, die geschlossen werden müssen. Es wird zuerst das „m_bAutoDelete“-Flag des Hintergrundtasks auf „FALSE“ gesetzt, um auf dessen Beendigung warten zu können. Da sich der Hintergrundtask dadurch aber nicht mehr selbst löscht, muss er mittels „delete“ von Hand gelöscht werden. Anschließend wird die „m_bWaitForMessage“-Variable, die innerhalb des Hintergrundtasks abgefragt wird, auf „FALSE“ gesetzt, damit sich der Hintergrundtask auch beendet. Die geöffneten Sockets werden mittels „closesocket“ geschlossen, wobei vorher noch alle Schreib-

und Lesezugriffe mittels „shutdown“ abgebrochen werden. Am Schluss wird noch auf die Beendigung des Hintergrundtask mittels „WaitForSingleObject“ gewartet und wie schon vorhin erwähnt der Hintergrundtask gelöscht. Unabhängig vom Status der „m_bWaitForMessage-Variable wird die zu „WSAStartup“ komplementäre Funktion „WSACleanup“ aufgerufen, welche alle noch offenen Ressourcen freigibt und sich anschließend ordnungsgemäß von der WinSock-DLL abmeldet.

```
// TODO: Code für die Behandlungsroutine für Nachrichten hier einfügen

if (m_bWaitForMessage == TRUE)
{
    m_pWaitForMessageThread->m_bAutoDelete = FALSE;
    m_bWaitForMessage = FALSE;

    // Shutdown and close the client socket
    if (m_clientSocket != INVALID_SOCKET)
    {
        shutdown(m_clientSocket, SD_BOTH);
        closesocket(m_clientSocket);
    }

    // Shutdown and close the server socket
    shutdown(m_serverSocket, SD_BOTH);
    closesocket(m_serverSocket);

    // Wait until the thread is terminated
    WaitForSingleObject(m_pWaitForMessageThread->m_hThread, INFINITE);
    delete m_pWaitForMessageThread;
}

WSACleanup();
```

Die Server-Applikation wäre hiermit fertig dokumentiert und wir können uns der Client-Applikation zuwenden, die am Pocket PC ihren Platz findet.

Die Client-Applikation

Wie die Server-Applikation handelt es sich auch bei der Client-Applikation um eine dialogfeldbasierte Anwendung, die mit Hilfe des MFC-Frameworks erstellt wurde. Jedoch benötigen Sie zum Erstellen dieser Pocket-PC-Anwendung das eMbedded Visual C++ 3.0, welches Sie bei den eMbedded Visual Tools 3.0 finden. Sie können dieses Paket kostenlos von der Microsoft-Homepage herunterladen.

Das User-Interface der Client-Applikation (siehe Abb. 6) wurde ebenfalls recht schlicht gehalten. Neben dem Eingabefeld für die zu sendenden Nachrichten befindet sich nur noch ein Button zum Absenden der Nachricht auf dem Dialog. Beim Drücken des „Senden“-Buttons wird der Text aus dem Eingabefeld ausgelesen und an den Server gesendet.

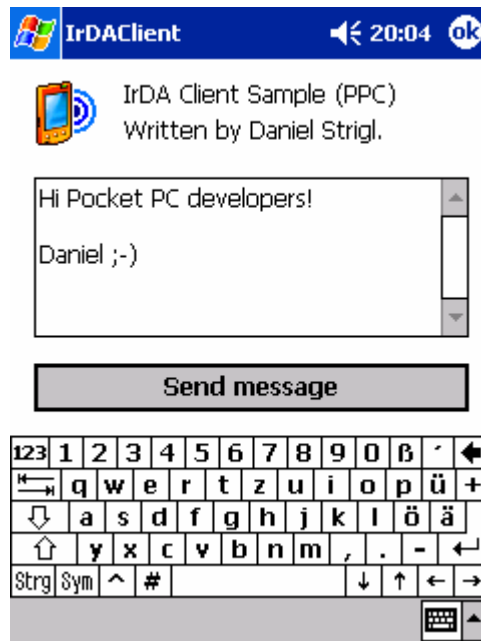


Abb. 6: Screenshot der Client-Applikation

Um die Socket-Funktionalität innerhalb der Client-Applikation nutzen zu können, linken Sie wie bei der Server-Applikation die WinSock Library zu Ihrem Projekt. Jedoch nennt sich diese unter Windows CE nicht „Ws2_32.lib“ sondern „Winsock.lib“. Wie dies erledigt wird haben sie ja bereits bei der Server-Applikation gesehen.

Die gesamte Socket-Funktionalität für die Infrarotverbindung wurde bei der Client-Applikation in zwei Klassen gepackt, die sich „CWinSock“ und „ClrDASocket“ nennen und in der Datei „IrDASocket.h“ und „IrDASocket.cpp“ wiederzufinden sind. Diese beiden Klassen kapseln die gesamte Socket-Funktionalität für den Zugriff auf die Infrarotschnittstelle und verringern somit den Programmieraufwand für die Client-Applikation. Diese beiden Klassen sind so konstruiert, um recht einfach auch in anderen Projekten Wiederverwendung zu finden.

Wird nun der „Sende“-Button des Dialogs betätigt, so wird zuerst das Eingabefeld, welches die zu sendende Nachricht enthält, ausgelesen und der Senderoutine übergeben, welche versucht, diese Nachricht an die Server-Applikation zu senden.

```
// TODO: Add your control notification handler code here

UpdateData();

// Send message to server
UINT uiResult = Send(m_strMessage);
if (uiResult)
{
    // Display an error message
    CString strCaption, strText;
    strCaption.LoadString(IDS_ERROR_MESSAGE_SEND);
    strText.LoadString(uiResult);
    MessageBox(strText, strCaption, MB_ICONEXCLAMATION | MB_OK);
}
}
```

Innerhalb der Senderoutine wird nun zuerst die zu übertragende Nachricht auf ihre Länge hin überprüft, d.h. es wird nachgesehen, ob überhaupt etwas zum Übertragen

ist. Wurde kein Text in das Eingabefeld eingegeben, so hat die Nachricht eine Länge von 0 Zeichen, und die Senderoutine bricht mit einem Fehler ab.

Als nächstes wird eine Instanz der Klasse „CWinSock“ erzeugt um sicherzustellen, dass der WinSock Layer ordnungsgemäß initialisiert wird. Die Klasse „CWinSock“ ruft innerhalb ihres Konstruktors die Funktion „WSAStartup“ und innerhalb des Destruktors die dazu komplementäre Funktion „WSACleanup“ auf. Damit wird sichergestellt dass vor Verwendung eines eigentlichen WinSock-Kommandos der WinSock-Layer ordnungsgemäß initialisiert und dieser bei Beendigung der Senderoutine auch wieder ordnungsgemäß aufgeräumt wurde.

In den nächsten beiden Zeilen Code wird eine Instanz der Klasse „CIrDASocket“ erzeugt und versucht, ein Socket mittels der Funktion „Open“ zu öffnen.

```
ASSERT(AfxIsValidString(lpszMessage));

// Check the message length
if (_tcslen(lpszMessage) <= 0)
    return IDS_ERROR_NO_MESSAGE;

CWaitCursor wc;

USES_CONVERSION;

// Make sure the socket layer is active
CWinSock winSock;

// Open an IrDA socket
CIrDASocket sd;
if (!sd.Open())
    return IDS_ERROR_IRDA_NO_SOCKET;
```

Wenn Sie sich die „Open“-Funktion der „CIrDASocket“-Klasse etwas genauer ansehen, können Sie erkennen, wie die geöffnete Socket-Verbindung, welche zuvor mittels dem Kommando „socket“ erzeugt wurde, durch das „ioctlsocket“-Kommando in den nicht-blockierenden Modus umgeschaltet wird. Dies bedeutet für den Entwickler, dass die WinSock-Funktionen nicht mehr blockierend arbeiten und er selber über die „WaitForOperation“-Funktion der „CIrDASocket“-Klasse auf die Beendigung eines Befehls warten oder nach einer bestimmten Zeit das Ganze abbrechen kann. Die meisten Funktionen der „CIrDASocket“-Klasse erwarten als Parameter eine sogenannte TIMEOUT-Zeit, welche angibt, wie lange auf die Beendigung eines Befehls gewartet werden soll. Wird diese Zeit überschritten, so wird der Befehl mit einem Fehler abgebrochen.

So können Sie z.B. der „Receive“-Funktion eine TIMEOUT-Zeit übergeben, die sie maximal warten soll, bis alle Daten empfangen wurden. Läuft die Zeit ab bevor alle Daten eingetroffen sind, bricht die Funktion mit einem Fehler ab, andernfalls kehrt sie mit Erfolg zurück.

Anschließend wird einige Zeit, hier handelt es sich um ca. 10 Sekunden, nach kompatiblen Infrarotgeräten in der Umgebung gesucht. Konnte kein Gerät mit aktiviertem Infrarot-Port in der näheren Umgebung des Infrarot-Ports gefunden werden, so wird die Senderoutine mit einem Fehler abgebrochen. Die eigentliche Funktionalität, das Suchen nach einem kompatiblen Infrarotgerät in der Nähe des

Pocket PC, steckt in der „EnumDevices“-Methode. Diese Routine listet, wie der Name bereits verrät, alle verfügbaren Infrarotgeräte in unmittelbarer Nähe des Pocket PC auf. Die „EnumDevices“-Funktion ruft wiederum nur die WinSock-Funktion „getsockopt“ mit dem Parameter „IRLMP_ENUMDEVICES“ auf, welcher angibt, im übergebenen Puffer eine Liste mit den gefundenen Infrarotgeräten zurückzugeben.

Wurde ein Gerät gefunden, so wird dessen Name ermittelt und beim User noch einmal nachgefragt, ob die Nachricht an das gefundene Gerät mit dem ermittelten Namen gesendet werden soll oder nicht (siehe Abb. 7).

```
// Try several times to find a device
DEVICELIST deviceList = { 0 };
for (int nTry = 0; nTry < (LONG_TIMEOUT / 1000); nTry++)
{
    // Try to find a device
    if (!sd.EnumDevices(&deviceList, sizeof(DEVICELIST)))
        return IDS_ERROR_IRDA_CANNOT_ENUMDEVS;

    // Abort if we have found a device
    if (deviceList.numDevice > 0)
        break;

    // Wait a second
    ::Sleep(1000);
}

// Check if all retries have been finished
if (deviceList.numDevice == 0)
    return IDS_ERROR_NO_DEVICES_FOUND;

// Obtain the devicename (ANSI)
LPCTSTR lpszDeviceName = A2CT(deviceList.Device[0].irdaDeviceName);

// Ask the user if recipient is okay
CString strCaption, strText;
strCaption.LoadString(IDS_MESSAGE_SEND);
strText.Format(IDS_MESSAGE_SENDTOHOST, lpszDeviceName);
if (MessageBox(strText, strCaption, MB_ICONQUESTION | MB_YESNO) != IDYES)
    return 0;
```

Akzeptiert der User dies, so wird als nächstes versucht, eine Verbindung mit dem Server auf dem zweiten Infrarotgerät herzustellen. Hierbei kommt erneut der sogenannte Service-Name zum Einsatz, bei uns „MySampleIrDAService“. Läuft auf dem zweiten Infrarotgerät keine Server-Applikation, die mit dem Service-Namen „MySampleIrDAService“ verbunden ist, so kommt auch keine Verbindung zustande und die „Connect“-Funktion der „CIRDASocket“-Klasse kehrt mit einem Fehler zurück, wodurch auch die Senderoutine mit einem Fehler beendet wird.

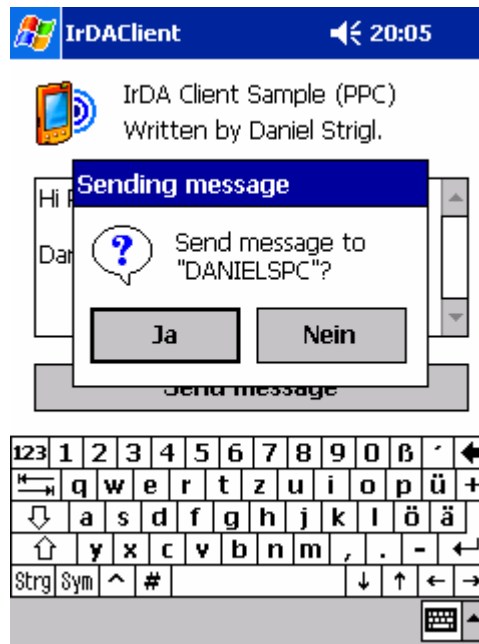


Abb. 7: Beim User noch mal nachfragen

Auch hier bei der „Connect“-Funktion können Sie eine TIMEOUT-Zeit angeben, die maximal gewartet wird bis eine Verbindung zustande kommt. Läuft diese Zeit jedoch ab, bevor eine Verbindung zustande kommt, wird die Senderoutine abgebrochen.

Steht die Verbindung, wird als nächstes die Länge der Nachricht ermittelt und dieser Wert mittels der Socket-Funktion „htonl“ in die sogenannte „TCP/IP Network Byte Order“ konvertiert. Warum dies notwendig ist, wurde bereits bei der Server-Applikation erläutert.

Zum Schluss wird dieser Wert, gefolgt von der eigentlichen Nachricht, an die Server-Applikation gesendet. Da unter Windows CE mit UNICODE-Zeichenketten gearbeitet wird, wird die Nachricht vor dem Absenden noch mit Hilfe des ATL Makros „T2CA“ in einen ANSI-String konvertiert.

```
// Restore the wait cursor
wc.Restore();

// A device has been found, so try to connect
if (!sd.Connect(IRDA_SERVICENAME, deviceList.Device[0].irdaDeviceID,
LONG_TIMEOUT))
return IDS_ERROR_IRDA_CANNOT_CONNECT;

// Send the length of the message to the server
u_long ulLength = htonl(_tcslen(lpszMessage));

if (!sd.Send(ulLength, SHORT_TIMEOUT))
return IDS_ERROR_IRDA_CANNOT_SEND;

// Send the message to the server
if (!sd.Send((LPCVOID) T2CA(lpszMessage), _tcslen(lpszMessage),
SHORT_TIMEOUT))
return IDS_ERROR_IRDA_CANNOT_SEND;

// Return successful
return 0;
```

Geschlossen wird die Socket-Verbindung automatisch beim Verlassen der Senderoutine im Destruktor der „ClrDASocket“-Klasse.

Am Ende des Artikels noch ein kleiner Tipp am Rande: Um das Eingabefeld (z.B. die Software-Tastatur) des Pocket PC, das sogenannte „Soft Input Panel“, automatisch beim Programmstart hochzuklappen bzw. beim Programmende wieder zu minimieren, rufen Sie die API-Funktion „SHSipPreference“ mit unterschiedlichen Parametern auf.

Um das „Soft Input Panel“ beim Programmstart automatisch hochzuklappen, fügen Sie folgende Zeile innerhalb der „OnInitDialog“-Funktion ihres Dialogs ein:

```
SHSipPreference(m_hWnd, SIP_UP);
```

Um das „Soft Input Panel“ beim Programmende wieder zu minimieren, also unsichtbar zu machen, fügen Sie folgende Zeile Code in die „OnDestroy“-Methode Ihres Dialoges ein:

```
SHSipPreference(m_hWnd, SIP_DOWN);
```

Über den Autor

Daniel Strigl arbeitet als Softwareentwickler bei einer österreichischen Firma. In seiner Freizeit entwickelt er Windows-Software und beschäftigt sich seit einiger Zeit mit der Entwicklung von Pocket-PC-Applikationen. Sie erreichen ihn auf seiner Homepage unter <http://www.hh-system.com/danielstrigl/>.