6.2.0 Stream-Input-Output-Funktionen

In den folgenden Abschnitten werden Ihnen Stream-Input-Output-Funktionen vorgestellt. Beispiele ergänzen die Beschreibungen.

6.2.0.1 OPEN

```
Stream = OPEN FileName [ FOR [ READ | INPUT ] [ WRITE | OUTPUT ] [ CREATE | APPEND ] [ WATCH ] ]
```

Öffnet einen Stream zum Lesen, Schreiben, Erzeugen oder zum Hinzufügen von Daten – optional auch zum Beobachten. Der Stream muss existieren oder es muss zusätzlich das Schlüsselwort CREATE verwendet werden.

- Wenn das Schlüsselwort CREATE angegeben ist, wird der Stream erzeugt. Falls er bereits existiert wird er vorher gelöscht.
- Wenn das Schlüsselwort APPEND angegeben ist, wird der Stream-Zeiger unmittelbar nach dem Öffnen an das Ende des Streams verschoben.
- Wenn die Schlüsselwörter READ oder WRITE angegeben sind, werden die Ein- und Ausgänge nicht gepuffert.
- Wenn die Schlüsselwörter INPUT oder OUTPUT angegeben sind, werden die Ein- und Ausgänge gepuffert.
- Wenn das Schlüsselwort WATCH angegeben ist, wird der Stream vom Interpreter über den Systemaufruf select(2) überwacht:
  - Wenn mindestens ein Byte aus der Datei gelesen werden kann, wird das Ereignis File_Read() ausgelöst.
  - Wenn mindestens ein Byte in die Datei geschrieben wird, wird das Ereignis File_Write() ausgelöst.

6.2.0.2 CLOSE

```
CLOSE [ # ] hStream
hStream.Close()
```

Beide Anweisungen schließen einen geöffneten Stream. Diese Anweisungen scheitern nie. Wenn Sie einen Prozess-Stream geöffnet haben, dann schließt er seine Standard-Eingabe – so, als ob Sie die Tasten-Kombination CTRL+D in einem Terminal eingeben würden. Die Standard-Eingabe und die Standard-Ausgabe beziehen sich auf die Daten-Kanäle in einer Konsole.

6.2.0.3 OPEN STRING

```
Stream = OPEN STRING [ aString ] [ FOR [ READ ] [ WRITE ] ]
```

Mit dem Befehl OPEN STRING können Sie auf einen String mit der Stream-Schnittstelle zugreifen.

- Wenn der String aString zum Lesen geöffnet wird, muss das String-Argument aString angegeben werden! Dann wird sequentiell auf den Inhalt der Zeichenkette zugegriffen.
- Das Lesen eines String-Streams ist immer möglich.
- Wird der String aString zum Schreiben geöffnet, so sammelt ein interner String-Stream-Puffer alle geschriebenen Daten.

Wenn Sie nach dem Öffnen die Tag-Eigenschaft mit einem Wert – wie hStringStream.Tag = "open" – belegen, dann haben Sie die Möglichkeit zu prüfen, ob der StringStream bereits geschlossen wurde.

6.2.0.4 CLOSE StringStream

```
String = CLOSE [ # ] StringStream
```

Das Schließen eines String-Streams gibt den kompletten Inhalt des internen String-Stream-Puffers zurück.

Beispiel

```
' Gambas class file

Public sLog As String
Public hLogStream As Stream
Public fTemperature As Float

Public Sub Form_Open()
  hLogStream = Open String sLog For Write
  SetLogHeader()
  LogTimer.Delay = 1 * 1000
  LogTimer.Start()
End

Public Sub btnReadFromStreamString_Click()
  TextArea1.Text = Close #hLogStream
  hLogStream = Open String sLog For Write
  SetLogHeader()
End

Public Sub SetLogHeader()
  Write #hLogStream, "PROTOKOLL" & gb.NewLine
  Write #hLogStream, "Datum: " & Format(Now, "dd. mmmm yyyy") & gb.NewLine
  Write #hLogStream, String$(38, "-") & gb.NewLine
End

Public Sub LogTimer_Timer()

' fTemperature = Round(RS232_Value,-2)
  fTemperature = Round(Rnd(19, 20), -1)

  Write #hLogStream, Format(Now, "hh:nn:ss") & "  |  " & "T = " & Str(fTemperature) & " °C"
  Write #hLogStream, gb.NewLine

End
```

Für den praktischen Einsatz können Sie die im o.a. Beispiel zufällig erzeugten Temperaturwerte durch die Temperaturwerte auch einer RS232-Schnittstelle mit Temperatursensor auslesen.

So sieht ein Protokoll-Auszug aus:

```
PROTOKOLL
Datum: 08. November 2018
-------------------------
09:36:31  |  T = 19,7 °C
09:36:32  |  T = 20,0 °C
09:36:33  |  T = 19,1 °C
09:36:34  |  T = 19,2 °C
09:36:35  |  T = 20,0 °C
09:36:36  |  T = 19,9 °C
```

6.2.0.5 WRITE

```
(a)    WRITE [ # wStream , ] Expression AS Datatype
(b)    WRITE [ # wStream , ] WString [ , Length ]
(c)    WRITE [ # wStream , ] Pointer , Length
```

(a) Schreiben von Daten eines bestimmten Datentyps

Die erste Syntax schreibt einen Ausdruck in den Stream wStream, indem dessen binäre Darstellung verwendet wird.

- Ist wStream nicht angegeben, wird die Standard-Ausgabe verwendet.
- Beim Schreiben eines Strings wird die Länge des Strings vor dem Inhalt des Strings gesendet.
- Der Datentyp des Ausdrucks kann einer der folgenden sein: NULL, Boolean, Byte, Short, Integer, Long, Pointer, Single, Float, Date, String, Variant, jedes Array oder Collection, oder eine beliebige Struktur.
- Wenn Expression eine Collection, ein Array oder eine Struktur ist, dann wird ihr Inhalt rekursiv geschrieben.
- Beim Schreiben einer Struktur muss der Strukturtyp als Datentyp angegeben werden.
- Wird ein nicht unterstützter Datentyp geschrieben oder eine zirkuläre Referenz erkannt, dann wird ein Fehler ausgelöst.
- Dieser Befehl verwendet die Byte-Reihenfolge von wStreams, um die Daten zu schreiben.

Beispiel

Sie können ein String-Array in einer Datei abspeichern und es so exportieren. Ein Array wird über Write serialisiert in eine Datei geschrieben. Der Inhalt der Datei besitzt ein gambas-spezifisches Datei-Format:

```
Public hFile As File
Public aNames As String[]
...
' Data export
If Dialog.SaveFile() Then Return
hFile = Open Dialog.Path For Write Create
Write #hFile, aNames As ARRAY
Close #hFile
```

(b) Schreiben des Inhalts einer Zeichenkette

Die zweite Syntax schreibt eine bestimmte Anzahl von Bytes – angegeben durch den Wert Length – aus dem String wString in den angegebenen Stream.

- Ist der Stream nicht angegeben, wird die Standard-Ausgabe verwendet.
- Wenn Length nicht angegeben ist, wird die Länge von wString verwendet.

(c) Schreiben des Speicherinhalts

Die dritte Syntax schreibt eine bestimmte Anzahl von Bytes – angegeben durch den Wert Length – von der Speicheradresse Pointer in den angegebenen Stream.

- Ist der Stream nicht angegeben, wird die Standard-Ausgabe verwendet.
- Wenn Length nicht angegeben ist, wird die Länge von wStream verwendet.

6.2.0.6 PRINT

```
PRINT [ # hStream , ] Expression [ { ; | ;; | , } Expression ... ] [ { ; | ;; | , }  ]
```

Die Anweisung schreibt den Inhalt von Expression in den Stream hStream.

- Wenn hStream nicht angegeben ist, dann wird die Standard-Ausgabe verwendet.
- Die Standard-Ausgabe kann durch die Anweisung OUTPUT TO umgelenkt werden.
- Die Ausdrücke können Sie durch die Funktion Str$ in Strings umwandeln.
- Wenn nach dem letzten Ausdruck kein Semikolon oder Komma steht, dann wird nach dem letz-

ten Ausdruck ein Zeilenendezeichen eingefügt. Das Zeilenendezeichen kann mit der Eigenschaft Stream.EndOfLine definiert werden.
- Wird das Semikolon verdoppelt, so wird zwischen den Ausdrücken ein Leerzeichen eingefügt.
- Wird anstelle eines Semikolons ein Komma verwendet, so wird ein Tabulatorzeichen (ASCII-Code 9) zur Trennung der Ausdrücke eingefügt.

6.2.0.7 READ

```
(a)    Variable = READ [ # rStream ] AS Datatype
(b)    Variable = READ [ # rStream , ] iLength
```

(a) Lesen von Daten eines bestimmten Datentyps

- Die erste Syntax liest den Stream als Datenstrom, dessen Typ durch *Datatype* angegeben wird.
- Ist der Stream rStream nicht angegeben, wird die Standard-Eingabe verwendet.
- Der zurückgegebene Datentyp kann einer der folgenden Typen sein: NULL, Boolean, Byte, Short, Integer, Long, Pointer, Single, Float, Date, String, Variant, beliebige Arrays, Collection oder Strukturen.
- Beim Lesen eines Strings muss die Länge des Strings dem Stringinhalt in den Stream-Daten vorangestellt werden.
- Wenn der Stream-Inhalt nicht gelesen werden kann, so wird ein Fehler ausgelöst.

Beispiel

Wurde ein Array über Write serialisiert in einen (Datei-)Stream geschrieben wurde, so kann das Array mit Read aus diesem ausgelesen werden:

```
Public hFile As File
Public aNames As String[]
...
' Data import
If Dialog.OpenFile() Then Return
hFile = Open Dialog.Path For Read
aNames = Read #hFile As ARRAY
Close #hFile
```

(b) Lesen des Inhalts eines Streams

- Die zweite Syntax liest aus dem Stream rStream eine Anzahl von Bytes, die durch das Length-Argument angegeben wird und gibt sie als String zurück.
- Ist iLength negativ, dann werden maximal (-Length) Bytes bis zum Ende des Streams gelesen.
- Ist der Stream nicht angegeben, wird der Standard-Eingabe verwendet.

Beispiel

```
Public sTemperatureDigit As String

hRS232 = New SerialPort As "hRS232"

Public Sub hRS232_Read()
  sTemperatureDigit  = Read #hRS232, Lof(hRS232)
End
```

6.2.0.8 LINE INPUT

```
LINE INPUT [ # hStream , ] Variable
```

Liest eine Text-Zeile aus dem Text-Stream in eine String-Variable.

Ist der Stream hStream nicht angegeben, dann wird die Standard-Eingabe verwendet. Es wird stets die komplette Zeile gelesen – mit Ausnahme des Zeilenendezeichens. Standardmäßig ist es die Konstante gb.Unix, die ein einzelnes Chr$(10)-Zeichen repräsentiert. Das Zeilenendezeichen kann mit der Eigenschaft Stream.EndOfLine definiert werden.

- Verwenden Sie LINE INPUT … nicht, um aus Binärdateien zu lesen, da Sie sie keine Zeilenstruktur verwenden. Nutzen Sie stattdessen READ.

- Verwenden Sie diesen Befehl nicht innerhalb des Read-Ereignisses eines Prozesses, der nicht unbedingt eine neue Zeile sendet, weil er zum Beispiel eine Eingabeaufforderung ausgibt. Es wird für immer blockiert und auf den Zeilenumbruch gewartet. So sendet zum Beispiel die Shell einen Befehlsprompt um zu signalisieren, dass sie bereit ist, ein neues Kommando zu empfangen. Dieser Prompt wird in der Regel nicht durch ein NewLine beendet, weshalb ein Programm – wenn es LINE INPUT verwendet – diesen Prompt nicht lesen kann. Die implizierte Bereitschaft der Shell weiterzuarbeiten geht verloren.

Beispiel 1

```
Public Sub AddTextToFile(FilePath As String, Text As String)

  Dim hFile As File

  Try hFile = Open FilePath For Append
  Print #hFile, Text

  Finally
    If Exist(FilePath) Then Close #hFile
  Catch
    Message.Error("Error:\n" & Error.Text & " in " & Error.Where)

End

Public Function GetTextFromFile(FilePath As String) As String

  Dim hFile As File
  Dim sLine, Text As String

  Try hFile = Open FilePath For Read

  While Not Eof(hFile)
    Line Input #hFile, sLine
    Text &= sLine & "\n"
  Wend

  Return Text

  Finally
    If Exist(FilePath) Then Close #hFile
  Catch
    Message.Error("Error:\n" & Error.Text & " in " & Error.Where)

End

Public Sub btnAddTextToFile_Click()

  Dim sLogDir As String

  sLogDir = Desktop.DataDir &/ "gambasbook" &/ Application.Name
  If Not Exist(sLogDir) Then Shell.MkDir(sLogDir)

  AddTextToFile(sLogDir &/ "rs232.log", "Time = " & Format(Now, "hh:nn:ss"))
  Catch
    Message.Error("Error:\n" & Error.Text & " in " & Error.Where)

End

Public Sub btnGetTextFromFile_Click()

  Dim sLogPath As String

  sLogPath = Desktop.DataDir &/ "gambasbook" &/ Application.Name &/ "rs232.log"
  txaLog.Text = GetTextFromFile(sLogPath)

End
```

Beispiel 2

Über die Standard-Eingabe (Terminal) werden Zeichen eingelesen, um damit den MediaPlayer zu steuern. Hinweise finden Sie auf: http://www.mplayerhq.hu/DOCS/man/de/mplayer.1.html.

```
' Gambas module file

Public mPlayer As New MediaPlayer

Public Sub Main()
 If mPlayer Then mPlayer = Null
```

```
 Start()
End

Public Sub Start()

  mPlayer = New MediaPlayer
  mPlayer.URL = "http://mp3channels.webradio.rockantenne.de/classic-perlen"

  mPlayer.Play()
  mPlayer.Audio.Volume = 1.0

  Print #File.Out, ""
  Print #File.Out, "-------------------------------------"
  Print #File.Out, "Instructions for use"
  Print #File.Out, "-------------------------------------"
  Print #File.Out, "+  ▶ Audio.Volume ▲"
  Print #File.Out, "-  ▶ Audio.Volume ▼"
  Print #File.Out, "p  ▶ Player.Pause"
  Print #File.Out, "r  ▶ Player.Run (After a pause)"
  Print #File.Out, "m  ▶ Audio.Mute (off/on)"
  Print #File.Out, "q  ▶ Player.Stop"
  Print #File.Out, "-------------------------------------"
  Print #File.Out, "Each command is followed by <ENTER>."
  Print #File.Out, "-------------------------------------"
  Print #File.Out, ""

End

Public Sub Application_Read()

  Dim sInput As String
  Dim fDeltaVolume As Float

  If mPlayer.Audio.Volume > 1.1 Then
    fDeltaVolume = 1.0
  Else
    fDeltaVolume = 0.1
  Endif

  Line Input #File.In, sInput

  Select Case sInput
    Case "q"
      mPlayer.Stop()
      Quit
    Case "p"
      mPlayer.Pause()
    Case "r" ' run
      mPlayer.Play()
    Case "m" ' toggle switch: mute on/mute off
      mPlayer.Audio.Mute = Not mPlayer.Audio.Mute
    Case "+"
      If mPlayer.Audio.Volume > 0.09 And mPlayer.Audio.Volume < 9.0 Then
         mPlayer.Audio.Volume += fDeltaVolume
      Endif
    Case "-"
      If mPlayer.Audio.Volume > 0.2 And mPlayer.Audio.Volume < 10.0 Then
         mPlayer.Audio.Volume -= fDeltaVolume
      Endif
  End Select

End
```

In diesem Beispiel werden Zeichen auf die Standard-Ausgabe ausgegeben (File.Out) und von der Standard-Eingabe (File.In) gelesen. Die Standard-Fehler-Ausgabe wird nicht genutzt.

6.2.0.9 INPUT FROM STREAM

```
INPUT FROM sStream
```

Leitet die Standard-Eingabe auf den Stream sStream um. Die Standard-Eingabe wird von INPUT, READ, LINE INPUT, Eof und Lof verwendet, wenn Sie kein Stream-Argument angeben. Aufrufe dieser Anweisung können verschachtelt werden.

6.2.0.10 INPUT FROM DEFAULT

```
INPUT FROM DEFAULT
```

Leitet die Standard-Eingabe auf den Stand vor der letzten Umleitung zurück.

6.2.0.11 OUTPUT TO STREAM

```
OUTPUT TO sStream
```

Leitet die Standard-Ausgabe auf den Stream sStream um. Die Standard-Ausgabe wird von PRINT und WRITE verwendet, wenn Sie kein Stream-Argument angeben. Aufrufe dieser Anweisung können verschachtelt werden.

6.2.0.12 OUTPUT TO DEFAULT

```
OUTPUT TO DEFAULT
```

Leitet die Standard-Ausgabe auf den Stand vor der letzten Umleitung zurück.

6.2.0.13 ERROR TO STREAM

```
ERROR TO eStream
```

Leitet die Standard-Fehlerausgabe auf den Stream eStream um. Die Standard-Fehlerausgabe wird von den Anweisungen ERROR und DEBUG verwendet. Aufrufe der Anweisung können verschachtelt werden.

6.2.0.14 ERROR TO DEFAULT

```
ERROR TO DEFAULT
```

Leitet die Standard-Ausgabe auf den Stand vor der letzten Umleitung zurück.

6.2.0.15 SEEK

```
SEEK [ # ] hStream , iPosition
```

Positioniert den Stream-Zeiger für den nächsten Lese-/Schreibvorgang. Wenn iPosition negativ ist, dann wird der Streamzeiger an eine Stelle relativ zum Ende der Datei verschoben. Um den Stream-Zeiger an das Ende einer Datei zu verschieben, müssen Sie die Lof(hStream)-Funktion verwenden. Beispiel:

Beispiel

Eine *Text-Datei* wird auf unterschiedliche Art ausgelesen. Nach der Variante 1 wird der Stream-Zeiger wieder auf den Anfang (Position 0) gesetzt und die Datei nach einer zweiten Variante noch einmal vollständig ausgelesen:

```
  hFile = Open $sCurrentFilePath For Input

' Variant 1
  hFile = Open $sCurrentFilePath For Input
  While Not Eof(hFile)
    Line Input #hFile, sLine
    sContent = sContent & sLine & gb.NewLine
  Wend

  sContent = sContent & gb.NewLine

' Variant 2
  Seek #hFile, 0
  For Each sLine In hFile.Lines
    sContent = sContent & sLine & gb.Lf
  Next

  sContent = sContent & gb.NewLine
```

6.2.0.16 Seek

```
iPosition = Seek ( Stream )
```

Gibt den aktuellen Wert des Stream-Zeigers des angegebenen Streams zurück. Der Rückgabewert iPosition ist eine Integer-Zahl vom Typ Long. Beachten Sie: Viele Stream-Typen wie zum Beispiel Prozess oder Socket besitzen keinen Stream-Pointer.

6.2.0.17 LOCK

```
hStream = LOCK sPath
```

Verwenden Sie LOCK und den angegebenen Pfad sPath, um eine systemweite Stream-Sperre zu erreichen. Wenn der angegebene Stream bereits durch einen anderen Prozess gesperrt ist (Advisory look), schlägt der Befehl fehl. Einen gesperrten Stream können Sie mit dem Befehl UNLOCK wieder entsperren.

Beispiel

Um einen zweiten Programmstart sicher zu verhindern, können Sie eine systemweite Sperre auf eine (Pseudo-)Datei legen. Bei einem weiteren Programm-Start wird in *** ein Fehler ausgelöst, weil bereits eine Sperre existiert. Vergessen Sie nicht, die Sperre beim Beenden des Programms wieder aufzuheben.

```
' Gambas class file

Public hLockFile As File
Public sFilePath As String

Public Sub _new()

  sFilePath = Desktop.DataDir &/ "lock.lock"
  Try hLockFile = Lock sFilePath ' <--- Step 1 ***
  If Error Then
     Message.Warning(Subst("&1 '&2' &3", ("There is already an instance of"), Application.Name, "!"))
     FMain.Close()
  Endif

End

' Main program ...

Public Sub Form_Close()
    Try Unlock hLockFile ' <--- Step 2
    FMain.Close()
End
```

6.2.0.18 LOCK WAIT

```
Stream = LOCK Path fWait Delay
```

Sperrt einen Stream für eine bestimmte Zeit fWait. Die Verzögerung wird in Sekunden angegeben. Der Typ von fWait ist Float. Wenn die Verzögerung vergangen ist, bevor die Sperre erfasst wurde, schlägt der Befehl fehl – es wird ein Fehler ausgelöst.

6.2.0.19 UNLOCK

```
UNLOCK [ # ] Stream
```

Entsperrt einen zuvor durch einen LOCK-Befehl gesperrten Stream. Auch das Schließen des Streams bewirkt automatisch das Entsperren des Streams.

6.2.0.20 EOF (end of file)

```
Result = Eof ( [ hStream AS Stream ] ) AS Boolean
```

Die Funktion gibt TRUE zurück, wenn das Ende des Streams erreicht wurde.

- Wenn hStream nicht angegeben ist, wird die Standard-Eingabe verwendet.
- Das Verhalten von Eof() hängt vom Stream-Blocking-Modus ab: (1) Wenn sich der Stream im Non-Blocking-Modus befindet, dann gibt Eof() den Funktionswert True zurück, wenn mindestens ein Byte aus dem Stream gelesen werden kann. (2) Wenn sich der Stream dagegen im Blo-

ckiermodus befindet, wartet Eof() zuerst auf Daten, bevor geprüft wird, ob etwas gelesen werden kann.

```
  hFile = Open $sCurrentFilePath For Input
  While Not Eof(hFile)
    Line Input #hFile, sLine
    sContent = sContent & sLine & gb.NewLine
  Wend
```

6.2.0.21 LOF (length of file)

```
Length = Lof ( hStream AS Stream ) AS Long
```

Gibt die Länge des geöffneten Streams hStream zurück, wenn hStream eine reguläre Datei ist. Ist hStream dagegen keine reguläre Datei, wie beispielsweise ein Socket, dann wird die Anzahl der Bytes zurück gegeben, die gleichzeitig gelesen werden können.

6.2.0.22 FLUSH

```
FLUSH [ [ # ] Stream ]
```

Die Daten eines gepufferten Streams werden unverzüglich ausgegeben – der Zwischenspeicher wird geleert. Wenn kein Stream angegeben ist, werden die Daten jedes geöffneten Streams ausgegeben. Sie können mit der Flush-Instruktion aber keine Daten aus einem anderen Prozess anfordern, wenn Sie dessen Daten aus einem Stream lesen wollen.

Beispiel

Es soll die Aufforderung "Geben Sie Ihren Namen ein: " im Terminal angezeigt werden und der Cursor soll – wie üblich – in der gleichen Zeile für die Eingabe der Antwort verbleiben. Das wird durch das Doppel-Semikolon nach der Print-Instruktion geleistet. Es gibt ein Leerzeichen aus und unterdrückt den Zeilenumbruch, den Print normalerweise an die Ausgabe anhängt. Nun sind Terminals aber zeilengepuffert. Das bedeutet, dass das Gambas-Programm den ge-Print-eten String entgegennimmt und ihn puffert, bis es das nächste Zeilenende-Zeichen liest. Erst dann werden die Daten an das Terminal geschickt. Da jetzt nach der Print-Instruktion in *** aber durch das Doppel-Semikolon der Zeilenumbruch fehlt, würden Sie (ohne Flush) im Terminal keine Aufforderung zu Gesicht bekommen, obwohl das Gambas-Programm schon bei der Line-Input-Instruktion ist und auf Eingaben wartet. Probieren Sie es doch einmal ohne Flush aus und tippen einfach blind Ihren Namen ein. Es wird funktionieren, aber Sie sehen die Aufforderung erst nach der zweiten Print-Instruktion, die ein Zeilenende liefert! Mit Flush teilen Sie Gambas mit, dass es die Daten unverzüglich ausgeben soll. Damit funktioniert das Programm wie erwartet.

```
' Gambas module file

Public Sub Main()

  Dim sName As String

  Print "Enter your name:";; ' ***

  Flush

  Line Input sName
  Print "Good to know that you are " & sName;
  Print

End
```

Programm-Start im Projektverzeichnis:

```
hans@mint-183 ~/GB3BUCH/6K_Stream/6.2.0_Stream-Input-Output-Funktionen/Projekte/Flush $ gbr3 flush.gambas
Enter your name: Mister Red
Good to know that you are Mister Red
```

6.2.0.23 NAMED PIPE

```
hStream = PIPE sPipeName FOR [ READ ] [ WRITE ] [ WATCH ]
hStream = OPEN PIPE sPipeName FOR [ READ ] [ WRITE ] [ WATCH ]
```

Öffnet eine benannte Pipe zum Lesen, zum Schreiben oder für beides. Wenn die Named Pipe nicht existiert, wird sie automatisch angelegt. Wenn die Named Pipe erfolgreich geöffnet wurde, wird ein Stream-Objekt an die Variable hStream zurückgegeben.

Eine ausführliche Beschreibung zu *Named Pipes* finden Sie im Kapitel 6.2.2.