Introduction

Hardware

A couple of weeks ago I was asked to write an assembly which should contain the necessary code for communicating with a payment terminal for a Point of Sale (POS) application. As you may have guessed from reading the title, the device in question is the Omni 3750 terminal. This terminal is produced by Verifone, check out the link if you require more information about this piece of hardware.

Although Verifone distributes this terminal in several countries supprisingly little can be found on the internet regarding communicating wih this device from any programming language whatsoever. The only thing I found was a rather short post on a Delphi forum. Together with some documents I received from the distributor of the Omni terminal here in Belgium this was enough to get me started.

This article concentrates on communicating with this device by using the SerialPort type and adhering to the required protocol. Hopefully this addition to the internet will save some people time in figuring this out.

Let's get started...

Table Of Contents

The Terminal

Not much needs to be said about the terminal itself. In order to be able to communicate with it you need to connect it to one of your computer's serial ports. In this day and age this can be a problem since most computers aren't equipped with serial ports anymore, which is also the case with mine.

To circumvent this problem I used a USB to serial port convertor. I used the Aten UC232A convertor, it provides you with a virtual serial (COM) port which you can use. If you're interested in purchasing this cable or any other they cost about 15 € if I recall correctly.

The only other issue that needs to be addressed regarding the terminal are the communication settings you need to set when communicating over the serial port.

Set these as follows:

- Baud Rate: 9600 (= default)
- Data bits: 7
- Stop bits: 1 (= default)
- Parity: Even

Top of page

Two Step Protocol

The main goal is to instruct the terminal to start a new payment transaction for a certain amount. The protocol used to communicate with the terminal is called the two step protocol. You initiate a new transaction by sending a message of the correct type to the terminal. After the customer has swiped his card an entered his pin code the transaction will be validated. As soon as the validation completes you'll receive a message from the terminal. So each payment transaction involves two messages, the transaction request and response, hence the name of the protocol.

The following two listings display the contents of the request and response messages.

Listing 1 - The Transaction Request

Field	Length	Type	Contents
Message type	4	String	00C1
Terminal number	8	Integer	Id of the terminal
Message version number	1	Integer	0: No additional info, 1: additional info
Amount	11	Integer	Decimals: Last two positions
Product information	20	String	Reserved for future use
Receipt number	6	Integer	Reserved for future use
Transaction type	2	Integer	00 = Purchase
Length of additional information	3	Integer	Length of additional information
Indicator additional information	2	Integer	01: Currency, 02: Petrol product, 03: Other product
Additional information	var	String	The additional information

The last three fields are part of a repeating group. Let's clarify this with an example. For illustrative purposes each field is separated by a /.

00C1/00001234/1/00000009999/(20 spaces)/000000/00/005/ 01/EUR/007/03/Candy/010/03/Magazine/014/03/Water bottle

As you might have noticed all integer fields require leading zeroes to fill in the missing gaps if the provided value isn't long enough. This message contains additional information consisting out of four groups, namely:

1. Length: 5, Indicator: 01, Information: EUR
2. Length: 7, Indicator: 03, Information: Candy
3. Length: 10, Indicator: 03, Information: Magazine
4. Length: 14, Indicator: 03, Information: Water bottle

The length always includes the length of the indicator and the information itself. Now let's move on to the transaction response. Listing 2 lists all the fields contained within the response.

Listing 2 - The Transaction Response

Field	Length	Type	Contents
Message type	4	String	00D1
Terminal number	8	Integer	Id of the terminal
Amount	12	Integer	Decimals: Last two positions
Type of card used to pay	2	String	E.g.: VI: Visa, 00: No cardswipe...etc.
Transaction result	1	Integer	0: OK, 1: Not OK...etc.

The transaction response can return more codes than just 0 or 1 to indicate the result of the transaction. Click here for a more exhaustive list. The same goes for the supported cards, you can find a complete list here.

Top of page

The Flow

The illustrate the flow of a transaction ponder the following example. A customer has to pay an amount of 39,95€. The first step is to send the transaction request to the Omni terminal. The following message needs to be sent (for illustrative purposes each field is once again separated with a /):

<STX>00C1/00001234/1/00000003995/(20 spaces)/000000/00/005/ 01/EUR<ETX><LRC>

As you may have noticed this example contains some additional elements beyond the fields discussed in the previous section. The following elements have been added:

- **<STX>**: Start of text (Hexadecimal value: 0x2)
- **<ETX>**: End of text (Hexadecimal value: 0x3)

The LRC or Longitude Redundancy Check is the exclusive OR of the entire message excluding the <STX> symbol but including the <ETX> symbol. In most cases only the amount and the <LRC> will be different.

After the terminal received this message it will reply with an acknowledge or a negative acknowledge. After receiving an acknowlegde you should wait for the transaction response (= the 00D1 message). If you receive a negative acknowlegde you must abort the transaction.

The acknowlegde and negative acknowlegde elements are expressed as follows:

- **<ACK>**: Acknowledge (Hexadecimal value: 0x6)
- **<NACK>**: Negative acknowlegde (Hexadecimal value: 0x15)

After the customer has swiped his card (e.g. Visa), entered his pin code and the transaction has been validated the terminal will respond with the transaction response. Ponder the following example:

<STX>00D1/00001234/1/000000003995/VI/1<ETX><LRC>

In most cases only the amount, card type, transaction result and <LRC> will differ.

The elements <STX>, <ETX>, <ACK> and <NACK> have been put in the following enumeration:

Listing 3 - Two Step Protocol

```
internal enum TwoStepProtocol
{
    StartOfText = 0x2,
    EndOfText = 0x3,
    Acknowledge = 0x6,
    NegativeAcknowledge = 0x15
}
```

The source code accompagnying this article contains even more elements in this enumeration, but this article only focuses on the absolute necessary things. Be sure to check out the source code on the download page.

Top of page

Longitude Redundancy Check

The Longitude Redundancy Check (LRC) is used to check if all characters of a message are received in the same order as they were sent. Listing 4 shows you how to calculate the LRC of a string. This function is contained within a static helper class termed TerminalRequest.

You need this function to calculate the LRC for the transaction request messages sent to the terminal and to verify the contents of the transaction response messages received from the terminal. I chose to not expose this function beyond the boundaries of the assembly it is contained in hence the internal access modifier, but feel free to change that.

Listing 4 - Calculating The Longitude Redundancy Check

```
internal static class TerminalRequest
{
    public static char CalculateLongitudinalRedundancyCheck(string source)
    {
        int result = 0;
        for (int i = 0; i < source.Length; i++)
        {
            result = result ^ (Byte)(Encoding.ASCII.GetBytes(source.Substring(i, 1))[0]);
        }
        return (Char)result;
    }
}
```

Top of page

Transaction Request

For brievity's sake this article only demonstrates how to send the transaction request. Handling the transaction response is left to the interested reader. You've already seen which parts are necessary to construct the message needed to request a transaction. Most of these parts are represented by enumerations. Listing 5 shows a few additional enumerations.

Listing 5 - MessageType, TransactionType and AdditionalInformationType enumerations

```
internal enum MessageType
{
    [StringValue("00C1")]
    TransactionRequest = 0,
    [StringValue("00D1")]
    TransactionResponse = 1
}

internal enum TransactionType
{
    [StringValue("00")]
    Purchase = 0
}

internal enum AdditionalInformationType
{
    [StringValue("01")]
    Currency = 0,
    [StringValue("02")]
    PetrolProductInformation = 1,
    [StringValue("03")]
    SpecialProductInformation = 2
}
```

Consult the first table in the Two Step Protocol section for more information about what these enumerations represent. Together with the TwoStepProtocol enumeration and the TerminalRequest helper class you can compose the following structure to represent all the parts needed to construct a message to request a transaction.

Listing 6 - TransactionRequest Structure

```
internal struct TransactionRequest
{
    public MessageType MessageType { get; set; }
    public int TerminalId { get; set; }
    public bool HasAdditionalInformation
    {
        get { return !String.IsNullOrEmpty(AdditionalInformation); }
    }
    public double Amount { get; set; }
    public int ReceiptNumber { get; set; }
    public TransactionType TransactionType { get; set; }
    public AdditionalInformationType AdditionalInformationType { get; set; }
    public string AdditionalInformation { get; set; }

    public override string ToString()
    {
        StringBuilder result = new StringBuilder();
        // Message type (= Payment request), Length: 4, Fieldtype: AN
        result.Append(StringValueAttribute.GetStringValue(MessageType));
        // Terminal number, Length: 8, Fieldtype: N
        result.Append(TerminalId.ToString(CultureInfo.InvariantCulture).PadLeft(8, '0'));
        // Message version number, Length: 1, Fieldtype: N
        // 0 = No addititional information, 1 = additional information
        result.Append(HasAdditionalInformation ? "1" : "0");
        // Amount, Length: 11, Fieldtype: N
        result.Append(Amount.ToString("#.##", CultureInfo.CurrentCulture).Replace(
            CultureInfo.CurrentCulture.NumberFormat.NumberDecimalSeparator, "").PadLeft(11, '0'));
        // Product info, Length: 20, Fieldtype: AN (Future use)
        result.Append(new String(' ', 20));
        // Receipt number, Length: 6, Fieldtype: N
        result.Append(ReceiptNumber.ToString(CultureInfo.InvariantCulture).PadLeft(6, '0'));
        // Transaction type, Length: 2, Fieldtype: N
        // 00 = Purchase
        result.Append(StringValueAttribute.GetStringValue(TransactionType));
        // Length of additional information, Length: 3, Fieldtype: N
        result.Append(AdditionalInformation.Length.ToString(CultureInfo.InvariantCulture).PadLeft(3, '0'));
        // Indicator additional information, Length: 2, Fieldtype: N
        // 01 = Currency, 02 = Product information (only for petrol)
        // 03 = Product information (Special terminal)
        result.Append(StringValueAttribute.GetStringValue(AdditionalInformationType));
        // Additional information, Length: variable, Fieldtype: AN
        result.Append(AdditionalInformation);
        // End of text
        result.Append((Char)TwoStepProtocol.EndOfText);
        // LRC Checksum
        result.Append(TerminalRequest.CalculateLongitudinalRedundancyCheck(result.ToString()));
        // Start of text
        result.Insert(0, (Char)TwoStepProtocol.StartOfText);
        return result.ToString();
    }
}
```

Each part of a transaction request is present in this structure. The ToString() method has been overridden to return the correct string to send out to the Omni terminal. The structure is self-describing so I will not explain all the details. Just compare the structure to Listing 1 and have a look at the comments in the overridden ToString() method.

The MessageType, TransactionType and AdditionalInformationType enumerations all contain a custom attribute called StringValue for each of their members. In the overridden ToString() method of the TransactionRequest structure the value of this custom attribute is read using the static GetStringValue() method of the StringValueAttribute class. A bit of reflection is used to retrieve these values, but since handling a transaction with a payment terminal is a lengthy request anyway it's worth the cost in my opinion. Discussing how to write custom attributes is beyond the scope of this article so I recommend that you check out the accompagnying source code and this article which explains how to write such an attribute.

Remark: As you may have noticed this structure offers no support for repeating groups of additional information. This is left as an exercise for you. Feel free to add support for this feature.

Top of page

The Terminal Class

Now that everything is in place it's time to design a class which represents an Omni terminal. This the first of two classes which are made public in the containing assembly. Most, if not all, of the interactions by the end-user with this assembly will be with this class.

Listing 7 shows you the code for the VerifoneOmni3750 class which represents the Omni terminal. The code is briefly discussed afterwards.

Listing 7 - VerifoneOmni3750 Class

```
public class VerifoneOmni3750 : IDisposable
{
    #region Instance fields

    private SerialPort serialPort = new SerialPort();

    #endregion

    #region Properties

    public SerialPort SerialPort
    {
        get { return serialPort; }
    }

    public int TerminalId { get; set; }

    #endregion

    #region Methods

    public void SendTransactionRequest(double amount, int receiptNumber, string currency)
    {
        TransactionRequest request = new TransactionRequest
        {
            MessageType = MessageType.TransactionRequest,
            TerminalId = this.TerminalId,
            Amount = amount,
            ReceiptNumber = receiptNumber,
            TransactionType = TransactionType.Purchase,
            AdditionalInformationType = AdditionalInformationType.Currency,
            AdditionalInformation = currency
        };

        SerialPort.WriteBinary(request.ToString());
    }

    #endregion

    #region IDisposable Members

    private bool disposed;

    private void Dispose(bool disposing)
    {
        if (!this.disposed)
        {
            if (disposing)
            {
                // Clean up any managed resources here.
                // ...
            }
            // Clean up any unmanaged resources here.
            serialPort.Dispose();
            disposed = true;
        }
    }

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    ~VerifoneOmni3750()
    {
        Dispose(false);
    }

    #endregion
}
```

This class consists out of four regions, namely:

1. **Instance Fields**: The class has only one instance field, namely an instance of the class SerialPort. The Omni terminal has exactly one serial port. Obviously all communication is done through this port.
2. **Properties**: Actually the class has two instance fields, the second one is not so visible to the naked eye because it is declared using an automatic property. An instance field for this property will be generated automatically in the resulting CIL code. The two properties of this class are SerialPort and TerminalId. The first one exposes the serial port instance field so the end-user of this class can set the baud rate, parity ... and so on. The second property allows you to set the id of the Omni terminal which is sent out with every transaction request.
3. **Methods**: The only method exposed by this class is the SendTransactionRequest(...) method. As you can derive from the name it sends out a transaction request to the Omni terminal. You can specify three parameters which will end up in the request, namely the amount, the receipt number and the used currency.
4. **Disposal Pattern**: Last but not least this class implements the IDisposable interface and adheres to the Disposal pattern to ensure that any unmanaged resources maintained by the serial port instance are properly disposed of.

The second class to be exposed from this assembly is a static class named SerialPortExtensions which adds a few extension methods to the SerialPort class from the System.IO.Ports namespace.

Listing 8 SerialPort Extension Methods

```
public static class SerialPortExtensions
{
    public static void Open(this SerialPort port, int portNumber)
    {
        port.PortName = "COM" + portNumber.ToString(CultureInfo.InvariantCulture);
        port.Open();
    }

    public static void WriteBinary(this SerialPort port, string command)
    {
        byte[] buffer = new byte[command.Length];
        for (int i = 0; i < command.Length; i++)
        {
            buffer[i] = (Byte)(Encoding.ASCII.GetBytes(command.Substring(i, 1))[0]);
        }
        port.Write(buffer, 0, buffer.Length);
    }
}
```

These extension methods are pretty simple in nature. The overloaded Open(...) method makes it easier to open a specific serial port and the WriteBinary(...) method allows you to send a string to said port in binary format.

Top of page

Client Application

Using the VerifoneOmni3750 class is pretty straightforward. Just create a new client application (Windows Forms, Console application...etc.), add a reference to the assembly containing the class and use the code shown in Listing 9.

Remark: The source code accompagnying this article contains a more elaborate version of the VerifoneOmni3750 class with added support for retries, timeouts and the ability to detect if the Omni terminal send back a <ACK> or <NACK>. Although this is not discussed in this article you should now have a strong enough foundation to explore this on your own.

Listing 9 Using The VerifoneOmni3750 Class

```
using System;
using System.IO.Ports;
using CGeers.Cardfon;

namespace ConsoleClientApplication
```

```
{
    class Program
    {
        static void Main()
        {
            using (VerifoneOmni3750 omniTerminal = new VerifoneOmni3750 { TerminalId = 1234 })
            {
                // Set the communication settings of the serial port.
                omniTerminal.SerialPort.BaudRate = 9600;
                omniTerminal.SerialPort.Parity = Parity.Even;
                omniTerminal.SerialPort.DataBits = 7;
                omniTerminal.SerialPort.StopBits = StopBits.One;

                // Establish a connection to the serial port (COM1).
                omniTerminal.SerialPort.Open(1);
                omniTerminal.SendTransactionRequest(39.95, 101, "EUR");
            }

            Console.ReadLine();
        }
    }
}
```

The code is self-explantory so I will not dive into further details, but this is basically everything the end-user of this class needs to do in order to communicate with the Omni terminal. All the complexities are nicely hidden away in the assembly containing the class. The code in Listing 9 shows you the simplest possible scenario. Feel free to adjust the VerifoneOmni3750 class to expose methods for more complex scenarios should the need arise.

Top of page

Summary

This article started off with discussing the connection settings and the two step protocol that is used in order to be able to communicate with the Omni terminal. Next the first step of the two way protocol, namely the transaction request was dissected and poured into C# language constructs. Finally all of these parts came together in the form of the VerifoneOmni3750 class which represents the Omni terminal. This class is used by end-user and thanks to the fact that the complexities of dealing with the terminal have been shielded this creates a slender and straightforward manner of use.

Top of page

Download

You can find the source code for this article on the Download page of this blog.

Top of page