

Forum:[Hack Forums](#)

Thread: [143 \[?\] - C# Programming Tutorials!!! \(Console- & Windows Form Applications\) #2](#)

BaussHacker - October 22, 2012, 2:55 am

Data Packets & Packet Structures

Introduction:

In this chapter we'll be looking at datapackets and how to structure packets. We'll be looking at the way I structure packets as there is a lot of way to do so!

Why do I cover this? Well we'll use this class in the socket tutorial, so it's important you follow here. We won't use structs at any time in this, but the pointer information is still important!

Let's start out by making our datapacket class.

It has to have a variable for a byte array and a property to get it.

It has to be unsafe, because we're going to use pointers.

Code public unsafe class DataPacket

```
{
    private byte[] _buffer;
    public byte[] DataBuffer
    {
        get
        {
            return _buffer;
        }
    }
}
```

Now we'll have to make two constructors. One constructor to pass a byte array to our datapacket and one constructor to give our datapacket a size and an id.

Code public DataPacket(byte[] inBuffer)

```
{
}
public DataPacket(int packetSize, int packetID)
{
}
```

In the constructor for passing a byte array we'll copy the data into a new byte array. We can do that using System.Buffer.BlockCopy, however we can go native as well and use PInvoke from msvcrt.dll with memcpy. Memcpy is far faster, but I felt lazy :p

<http://www.pinvoke.net/default.aspx/msvcrt.memcpy>

Code public DataPacket(byte[] inBuffer)

```
{
    _buffer = new byte[inBuffer.Length];
    System.Buffer.BlockCopy(inBuffer, 0, _buffer, 0, inBuffer.Length);
}
```

We won't look into the other constructor just yet as we need to create our reading/writing.

First of all we'll need to create a pointer to our buffer. We'll make the pointer a property.

Code private byte* dataPointer

```
{
    get
    {
    }
}
```

Now to create the pointer we'll have to use the fixed keyword.

Code ' get

```
{
    fixed (byte* pointer = _buffer)
    return pointer;
}
```

Reading and writing is basically the same with pointers.

I'll make the first write and read method for you then you can make for the rest datatypes. (Don't make for string just yet as I'll get into that after!)

The writing method has to take two parameters. The value and the offset.

Then remember from how we read data from the pointer (in the struct part.)

That's the same way we'll do to write, but instead of returning we'll just give it a value.

Code public void WriteByte(byte value, int offset)

```
{
    (*(byte*)(dataPointer + offset)) = value;
}
```

The same goes for reading, except it should only take one parameter as the offset and then return the value.

Code public byte ReadByte(int offset)

```
{
    return (*(byte*)(dataPointer + offset));
}
```

If you cannot figure out how to make them here is the all, but really... Try yourself.

Code public void WriteSByte(sbyte value, int offset)

```
{
    (*(sbyte*)(dataPointer + offset)) = value;
}
```

public void WriteInt16(short value, int offset)

```
{
    (*(short*)(dataPointer + offset)) = value;
}
```

public void WriteInt32(int value, int offset)

```
{
    (*(int*)(dataPointer + offset)) = value;
}
```

public void WriteInt64(long value, int offset)

```
{
    (*(long*)(dataPointer + offset)) = value;
}
```

public void WriteByte(byte value, int offset)

```
{
    (*(byte*)(dataPointer + offset)) = value;
}
```

public void WriteUInt16(ushort value, int offset)

```

{
    (*(ushort*)(dataPointer + offset)) = value;
}
public void WriteUInt32(uint value, int offset)
{
    (*(uint*)(dataPointer + offset)) = value;
}
public void WriteUInt64(ulong value, int offset)
{
    (*(ulong*)(dataPointer + offset)) = value;
}
public sbyte ReadSByte(int offset)
{
    return (*(sbyte*)(dataPointer + offset));
}
public short ReadInt16(int offset)
{
    return (*(short*)(dataPointer + offset));
}
public int ReadInt32(int offset)
{
    return (*(int*)(dataPointer + offset));
}
public long ReadInt64(int offset)
{
    return (*(long*)(dataPointer + offset));
}
public byte ReadByte(int offset)
{
    return (*(byte*)(dataPointer + offset));
}
public ushort ReadUInt16(int offset)
{
    return (*(ushort*)(dataPointer + offset));
}
public uint ReadUInt32(int offset)
{
    return (*(uint*)(dataPointer + offset));
}
public ulong ReadUInt64(int offset)
{
    return (*(ulong*)(dataPointer + offset));
}

```

Now let's look into writing and reading string data.

Remember back at data types that I explained strings are not a real data type? Well they're a collection of unicode characters (char) and a char equals a byte. However we'll have to convert the chars to bytes.

What we can do is loop through all the characters in the string and then convert them to a byte and write them into our buffer.

```

Code public void WriteString(string value, int offset)
{
    foreach (char _char in value)
        WriteByte((byte)_char, offset++);
}

```

}The reason we do offset++ it's because we don't want all the characters to be stored at the same offset.

Now there is another writing method for strings we want to implement which is writing the length of the string as well. The reason is that we cannot always know what the length of the string is so we want to have it as dynamically as we can!

```
Code public void WriteStringWithLength(string value, int offset)
{
    WriteByte((byte)value.Length, offset);
    offset++;
    WriteString(value, offset);
}
```

Basically what the code above does is writing the length of the string into the current offset and then it will add 1 to the offset and write the string values from there.

Now what about reading a string? That's basically the same. We'll have to convert the bytes to characters. A good idea is to use StringBuilder here for a better performance.

```
Code public string ReadString(int offset, int length)
{
    StringBuilder stringBuilder = new StringBuilder();

    for (int i = 0; i < length; i++)
    {
        byte _byte = ReadByte(offset + i);
        stringBuilder.Append((char)_byte);
    }

    return stringBuilder.ToString();
}
```

The code above will read a string with a specific length. Now what about if we could make it dynamically? Well that would only work if the string was written with the WithLength method, but we'll just have to gather the length from there then.

```
Code public string ReadStringFromLength(int offset)
{
    byte Length = ReadByte(offset);
    offset++;
    return ReadString(offset, Length);
}
```

And that's it. Now we actually have a class that wraps around a datapacket. Now the time to get back to the other constructor comes.

All we have to do is write the size to the offset 0 and the id to the offset 4, because it's an int.

```
Code public DataPacket(int packetSize, int packetID)
{
    _buffer = new byte[packetSize];
    WriteInt32(packetSize, 0);
    WriteInt32(packetID, 4);
}
```

Now what you can use this for is to wrap it around an actual class for a specific packet.

Example:

```
Code public class AuthPacket : DataPacket
{

```

```

public AuthPacket(byte[] inBuffer)
: base(inBuffer)
{
}

public AuthPacket(string account, string password)
: base(8 + account.Length + password.Length + 2, 1000)
{
    WriteStringWithLength(account, 8);
    WriteStringWithLength(password, account.Length + 8 + 1);
}

public string Account
{
    get
    {
        return ReadStringFromLength(8);
    }
}

public string Password
{
    get
    {
        return ReadStringFromLength(Account.Length + 8 + 1);
    }
}

```

Example Usage:

Code // creating

```

AuthPacket packet = new AuthPacket("test", "1234");
byte[] buffer = packet.DataBuffer;

```

// reading

```

Console.WriteLine(packet.Account);
Console.WriteLine(packet.Password);

```

I think that was it about datapackets and packet structures!

Downloads/Links:

[DataPackets Download \(Source\)](#)

BaussHacker - October 22, 2012, 2:56 am

Sockets #1 (Server)

Introduction:

In this chapter we'll be looking into sockets. This chapter will be covering the server part.

If you haven't read any of the tutorials and just skipped here then if anything is referred to old material then it's up to you to figure it out! I'd strongly suggest to read ALL the parts before jumping into sockets.

I will cover both server and client sockets.

First of all it's important that you understand what a socket is.

A socket is basically an endpoint that is bound to a specific ip and port. You can send/receive data through sockets.

The way we're going to design our sockets will be the following:

The reason I wrote linked is that they'll both be using the same wrapper, but we'll set it up to work for both and the reason is that there is no point in coding the same thing two times.

I will start out by explaining the server part.

A server is an application which is located either on a local machine or an extern machine. You connect to the server through a socket, the servers IP and Port would be the end point.

A server does not necessary has to be something that you connect to on an extern machine, an example could be sql server. If you're hosting the sql server and wants to use it ex. through your application then you'll most likely connect to it local. (It's possible to connect remote to a database server as well, but NEVER, I repeat NEVER do it through your client application, because it'll leave it open to security issues and I have seen a lot of people do that (Usually in the VB section, ironic LOL.!!))

For this we'll need to use two namespaces:
System.Net and System.Net.Sockets.

We'll start by making a class for our socket server.

```
Code public class SocketServer
{
}
```

We'll have to add one variable to it as a Socket.

```
Code private Socket _socket;
```

Now we need a constructor in which we'll create our socket.

```
Code _socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
```

Let me explain to you why I've chosen to use the following things.

The address family is InterNetwork because we want to connect to the internet, there is also options for Firefox and such, but I won't go into that stuff.

The socket type is Stream, because we want to stream data over the network. Raw is also used a lot, but we don't want that now.

Now for the protocol type. I've chosen Tcp for a simple reason. Tcp will never have its data lost and if it loses any data it will always report back with an error. If we were to choose Udp then we'd be risking a chance that we might lose data, because Udp is about speed. It's faster to use Udp, but they might lose data and it won't report back if any data was lost or anything. Tcp on the other side is about being effecient and that it actually works like it should. Usually Tcp is used for real servers, games and such where Udp is usually used for streaming (Ex. streaming movies) because they'll deliever the data faster, ut it might get lost on the way and such!)

Now we need to create 3 methods.

2 public methods called Start and Accept. The start method should take two parameters, one as a string (IP) and one as an int (Port). The accept method should take no parameters at all.

```
Code public void Start(string IP, int Port)
```

```

{
}
public void Accept()
{
}

```

Let's look at the Start method first. In the start method we'll bind the server and start to listen for connections.

To bind our socket to an end point we'll have to create an IPEndPoint from the IP and Port passed as parameters. IPAddress.Parse will parse a string to an IP (If the string is a valid IP lol.)

Code `_socket.Bind(new IPEndPoint(IPAddress.Parse(IP), Port));`

When we call the Listen method we'll have to pass an int as the parameter. That is the pending queue for our connections. I usually use 100, but you can use any value you want, I've seen some use 500 and such as well, but I'll just use 100.

Code `_socket.Listen(100);`

Now let's make a new class and call it SocketClient, because we're going to use that now (We'll not do anything in it yet!)

Now go back to the server.

We'll have to make a new method now which should be private and take one parameter as IAsyncResult. The method itself is a callback from an async callback.

Code `private void Accept_Callback(IAsyncResult asyncResult)`

```

{
}

```

Now in the Accept method we'll call the BeginAccept method in our socket class and it takes 2 parameters. the async call back (A delegate) and an object (Our SocketClient). We'll create a new instance of the SocketClient here.

Code `public void Accept()`

```

{
    _socket.BeginAccept(new AsyncCallback(Accept_Callback), new SocketClient());
}

```

Now we need to handle the call back. In the end of the call back call the Accept method again and every thing that we do from now on should be above that!

Code `private void Accept_Callback(IAsyncResult asyncResult)`

```

{
    // do things here

```

```

    Accept();
}

```

Now we'll go into our SocketClient class.

First thing we'll do is making a few variables.

Code `private byte[] dataHolder;`

`private Socket _socket;`

The dataHolder is a byte array which should hold all the data that's received.

The _socket is the connection of the client.

Now we'll make a new method returning a bool called EndAccept which should take 2 parameters as a Socket (as server) and IAsyncResult. Basically this should call the Socket.EndAccept method.

Code `public bool EndAccept(Socket server, IAsyncResult asyncResult)`

```

{
}

```

We should create a try/catch and on catch we'll return false and in try we'll return true.

Code try

```
{
    return true;
}
catch
{
    return false;
}
```

What we'll do now is making the socket in our class equal to the end accept method and pass the async result as parameter.

Code public bool EndAccept(Socket server, IAsyncResult asyncResult)

```
{
    try
    {
        _socket = server.EndAccept(asyncResult);
        return true;
    }
    catch
    {
        return false;
    }
}
```

And that's basically it. What we'll do now is creating 3 delegates (Do it in a new file if you want or wherever you prefer, but not in any class!)

Code public delegate void ConnectionEvent(SocketClient Client);

public delegate void BufferEvent(SocketClient Client, byte[] Buffer);

In the server class we'll have to define the 2 delegates (Make them static!)

2 for connection (OnConnect and OnDisconnect) and 1 for receive (OnReceive)

Code public static ConnectionEvent OnConnection, OnDisconnection;

public static BufferEvent OnReceive;

Now let's return to the accept callback.

First of all we'll check if the connection was accepted and if it's accepted we'll invoke the OnConnection delegate.

However we'll need to invoke the SocketClient and to do that we'll need to get it and we can do that from our async result.

Code private void Accept_Callback(IAsyncResult asyncResult)

```
{
    SocketClient Client = asyncResult.AsyncState as SocketClient;
    if (Client.EndAccept(_socket, asyncResult))
    {
        OnConnection.Invoke(Client);
    }
    Accept();
}
```

Now in the SocketClient class make a method called Receive and it should take no parameters. Call that method after the OnConnection has been invoked.

Code private void Accept_Callback(IAsyncResult asyncResult)

```
{
    SocketClient Client = asyncResult.AsyncState as SocketClient;
    if (Client.EndAccept(_socket, asyncResult))
```

```

{
    OnConnection.Invoke(Client);
    Client.Receive();
}
Accept();
}

```

Now our server socket is done. The only thing we need is to receive/send data which is in the SocketClient we'll handle that.

First of all make a private method called ReceiveData which should have one parameter as an int which is the size of the data we want to receive.

```

Code private void ReceiveData(int size)
{
}

```

Now we need to make another call back, this time it's for receive.

```

Code private void Receive_Callback(IAsyncResult asyncResult)
{
}

```

Okay now add two other variables to the class which should be for the data size received and the required data size.

```

Code private int requiredSize, currentSize;

```

And now another variable as a bool which should be indicating whether or not we're receiving the header and it should be true as a start.

```

Code private bool isHeader = true;

```

Now call ReceiveData from Receive with a size of 4, because we want that to get the size of the packet.

```

Code public void Receive()
{
    ReceiveData(4);
}

```

Now in ReceiveData we have to call the BeginReceive method from our socket.

```

Code private void ReceiveData(int size)
{
    dataHolder = new byte[size];
    _socket.BeginReceive(dataHolder, 0, size, SocketFlags.None, new
AsyncCallback(Receive_Callback), null);
}

```

Now in the receive callback we have to put a try catch, 2 variables for size and socket-error, but we also have to make a check if it's in header or not.

We also have to check if the receive was a success.

```

Code private void Receive_Callback(IAsyncResult asyncResult)
{
    try
    {
        SocketError err;
        int receiveSize = _socket.EndReceive(asyncResult, out err);
        if (err != SocketError.Success)
        {
            return;
        }
        if (isHeader)

```

```

{
}
else
{

}
}
catch
{
}
}

```

Now implemented the data packet class that we used in the previous tutorial.

Before we continue we have to make a disconnect method.

Basically it should just call the `Socket.Disconnect` method with the parameter as false so the user can connect again! It should also check if the user is already disconnected, because we don't want to invoke the event two times.

```

Code public void Disconnect()
{
    if (alreadyDisconnected)
        return;
    alreadyDisconnected = true;

    _socket.Disconnect(false);
    SocketServer.OnDisconnection.Invoke(this);
}

```

Okay back to the callback.

At this block:

```

Code if (err != SocketError.Success)
{
    return;
}It should disconnect.

```

Same goes here:

Code catch

```

{
}

```

Result:

```

Code private void Receive_Callback(IAsyncResult asyncResult)
{
    try
    {
        SocketError err;
        int receiveSize = _socket.EndReceive(asyncResult, out err);
        if (err != SocketError.Success)
        {
            Disconnect();
        }
        return;
    }
    if (isHeader)
    {

```

```

    }
    else
    {

    }
}
catch
{
    Disconnect();
}
}

```

Now we first have to handle the header which is a simple process.

We'll create a new datapacket as the header and it should have the dataholder as parameter. Then the required size should be the headers data (as int) - 4, because we do not want to receive the size again.

Then we'll call ReceiveData with the required size and at last we'll set isHeader to false.

```

Code DataPacket header = new DataPacket(dataHolder);
    requiredSize = (header.ReadInt32(0) - 4);
    ReceiveData(requiredSize);
    isHeader = false;

```

Now to handle the body.

First of all the `currentSize` should have added the `receivedSize` to it.

```

Code currentSize += receivedSize;

```

Then we'll check if the `currentSize` is equal to the `receivedSize`.

```

Code if (currentSize == requiredSize)

```

And else we'll check if the `currentSize` is above the `requiredSize` and if it is then we'll disconnect, because that means we've received an invalid buffer.

```

Code else if (currentSize > requiredSize)
{
    Disconnect();
}

```

This should be the callback as of now:

```

Code private void Receive_Callback(IAsyncResult asyncResult)
{
    try
    {
        SocketError err;
        int receiveSize = _socket.EndReceive(asyncResult, out err);
        if (err != SocketError.Success)
        {
            Disconnect();
            return;
        }
        if (isHeader)
        {
            DataPacket header = new DataPacket(dataHolder);
            requiredSize = (header.ReadInt32(0) - 4);
            ReceiveData(requiredSize);
            isHeader = false;
        }
        else
        {

```

```

currentSize += receiveSize;
if (currentSize == requiredSize)
{

}
else if (currentSize > requiredSize)
{
Disconnect();
}
}
}
catch
{
Disconnect();
}
}

```

Now if the sizes match we'll reset the currentSize and tell it that it's not in the header anymore.

Code `currentSize = 0;`

`isHeader = true;`

Then we'll make a new byte array with the requiredSize and copy the data from dataholder into it.

Code `byte[] rBuffer = new byte[requiredSize];`

`System.Buffer.BlockCopy(dataHolder, 0, rBuffer, 0, requiredSize);`

At last we'll invoke the OnReceive event and then start to receive the next header.

Code `SocketServer.OnReceive.Invoke(this, rBuffer);`

`ReceiveData(4);`

Now the last thing to do for the socket client is to make a send method as we also want to send data. This should just take one parameter as a DataPacket and within the method we'll copy the buffer from the data packet into a new byte array.

Code `public void Send(DataPacket packet)`

```

{
    int size = packet.DataBuffer.Length;
    byte[] sendBuffer = new byte[size];
    System.Buffer.BlockCopy(packet.DataBuffer, 0, sendBuffer, 0, size);
}

```

We'll also call the BeginSend method here, but before that we'll have to check if the client is connected and also lock the client in case of cross-threading problems!

Code `try`

```

{
    System.Threading.Monitor.Enter(this);

    if (_socket.Connected)
    {
        _socket.BeginSend(sendBuffer, 0, size, SocketFlags.None, new AsyncCallback(Send_Callback),
null);
    }
}
finally
{
    System.Threading.Monitor.Exit(this);
}

```

We also need a callback, but I don't think I need to explain this callback...

Code `private void Send_Callback(IAsyncResult asyncResult)`

```

{
    try

```

```

{
int size = _socket.EndSend(asyncResult);
if (size < 8) // if the size/id is not included in the packet
Disconnect();
}
catch
{
Disconnect();
}
}

```

Downloads/Links:

N/A (Please look at the client part!)

BaussHacker - October 22, 2012, 2:58 am

Sockets #2 (Client)

Introduction:

In this chapter I won't explain a lot and just explain and put some code that you should understand as I've already went through it in the server part.

Now to the client. I won't really explain a lot about that, because there isn't much to explain. The only difference from that is that the SocketClient class should not be accepted by any server, but connect to one, so the SocketServer class is not needed here and neither is the EndAccept method. All we actually have to do after is implementing the events for disconnecting and receiving.

You should be able to do that by yourself, but if you struggle here is the complete client:

```

Code public delegate void ConnecionEvent(SocketClient Client);
    public delegate void BufferEvent(SocketClient Client, byte[] Buffer);

```

```

    public class SocketClient
    {
    private byte[] dataHolder;
    private Socket _socket;

```

```

    public ConnecionEvent OnDisconnection;
    public BufferEvent OnReceive;

```

```

    public void Receive()
    {
        ReceiveData(4);
    }

```

```

    private void ReceiveData(int size)
    {
        dataHolder = new byte[size];
        _socket.BeginReceive(dataHolder, 0, size, SocketFlags.None, new
AsyncCallback(Receive_Callback), null);
    }

```

```

    private int requiredSize, currentSize;
    private bool isHeader = true;

```

```

    private void Receive_Callback(IAsyncResult asyncResult)

```

```

{
    try
    {
        SocketError err;
        int receiveSize = _socket.EndReceive(asyncResult, out err);
        if (err != SocketError.Success)
        {
            Disconnect();
            return;
        }
        if (isHeader)
        {
            DataPacket header = new DataPacket(dataHolder);
            requiredSize = (header.ReadInt32(0) - 4);
            ReceiveData(requiredSize);
            isHeader = false;
        }
        else
        {
            currentSize += receiveSize;
            if (currentSize == requiredSize)
            {
                currentSize = 0;
                isHeader = true;

                byte[] rBuffer = new byte[requiredSize];
                System.Buffer.BlockCopy(dataHolder, 0, rBuffer, 0, requiredSize);
                OnReceive.Invoke(this, rBuffer);
                ReceiveData(4);
            }
            else if (currentSize > requiredSize)
            {
                Disconnect();
            }
        }
    }
    catch
    {
        Disconnect();
    }
}

```

```

private bool alreadyDisconnected = false;

```

```

public void Disconnect()
{
    if (alreadyDisconnected)
        return;
    alreadyDisconnected = true;

    _socket.Disconnect(false);
    OnDisconnection.Invoke(this);
}

```

```

public void Send(DataPacket packet)
{
    int size = packet.DataBuffer.Length;
    byte[] sendBuffer = new byte[size];
    System.Buffer.BlockCopy(packet.DataBuffer, 0, sendBuffer, 0, size);
    try
    {
        System.Threading.Monitor.Enter(this);

        if (_socket.Connected)
        {
            _socket.BeginSend(sendBuffer, 0, size, SocketFlags.None, new AsyncCallback(Send_Callback),
null);
        }
    }
    finally
    {
        System.Threading.Monitor.Exit(this);
    }
}
private void Send_Callback(IAsyncResult asyncResult)
{
    try
    {
        int size = _socket.EndSend(asyncResult);
        if (size < 8) // if the size/id is not included in the packet
            Disconnect();
    }
    catch
    {
        Disconnect();
    }
}

public void Connect(string IP, int Port)
{
    _socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
    _socket.Connect(new IPEndPoint(IPAddress.Parse(IP), Port));
}
}

```

Now what about implementing these two?

Here is an example for the server:

Code namespace Server

```

{
    class Program
    {
        static void Main(string[] args)
        {
            SocketServer server = new SocketServer();
            SocketServer.OnConnection = new ConnecionEvent(OnConnection);
            SocketServer.OnDisconnection = new ConnecionEvent(OnDisconnection);
            SocketServer.OnReceive = new BufferEvent(OnReceive);
        }
    }
}

```

```

server.Start("127.0.0.1", 8886);
server.Accept();

while (true)
    Console.ReadLine();
}

static void OnConnection(SocketClient Client)
{
    Console.WriteLine("connected.");
}
static void OnDisconnection(SocketClient Client)
{
    Console.WriteLine("disconnected.");
}
static void OnReceive(SocketClient Client, byte[] Packet)
{
    Console.WriteLine("received data");
    DataPacket packet = new DataPacket(10 + 8, 100);
    packet.WriteString("0123456789", 8);
    Client.Send(packet);
}
}
}

```

Here is an example for the client:

Code namespace Client

```

{
    class Program
    {
static void Main(string[] args)
{
    SocketClient Client = new SocketClient();
    Console.ReadLine();
    Client.OnDisconnection = new ConneccionEvent(OnDisconnection);
    Client.OnReceive = new BufferEvent(OnReceive);
    Client.Connect("127.0.0.1", 8886);
    Client.Receive();
    while (true)
    {
        Console.ReadLine();
        DataPacket packet = new DataPacket(10 + 8, 100);
        packet.WriteString("0123456789", 8);
        Client.Send(packet);
    }
}

static void OnDisconnection(SocketClient Client)
{
    Console.WriteLine("disconnected.");
}

static void OnReceive(SocketClient Client, byte[] Packet)
{

```

```

    Console.WriteLine("received data.");
}
}
}

```

That was my socket tutorial! :) The difference from my old is that it required packets to be split, this does not!

Downloads/Links:

[Sockets Download \(Source\)](#)

PInvoke

Introduction:

In this chapter we'll take a look at the basics of PInvoke.

Okay I won't explain much about PInvoke, but basically PInvoke is the ability of invoking unmanaged code ex. code created in C++. It allows us to execute methods in unmanaged libraries.

You can do a lot things just by PInvoking the Windows API.

To do this we'll have to use the System.Runtime.InteropServices namespace.

To import any unmanaged DLL we'll have to use the DllImport attribute.

Example on PInvoke for memcpy:

Code [DllImport("msvcrt.dll", EntryPoint = "memcpy", CallingConvention = CallingConvention.Cdecl, SetLastError = false)]

public static extern void* memcpy(void* dest, void* src, uint count);

Now what does all that mean?

First of all we specify the DLL file we wants to use. Which is msvcrt.dll (Usually located in c:\windows\system32). The next thing we specify is the entry point (The method) which is memcpy. This is not necessary to have as we're already declaring our method as extern. It's necessary if the metho names does not match! Then we have the calling convention which for memcpy is cdecl, there is also stdcall etc. You'll be more familiar with this if you're from the C++ environment.

<http://cplus.about.com/od/introductiontoprogramming/a/conventions.htm>

The last thing we see is SetLastError which just tells the dll import that if the calls should set last error before returning from the attribute.

There is a lot more named parameters which you can see here:

<http://msdn.microsoft.com/en-us/library/system.runtime.interopservices.dllimportattribute.aspx>

Now how do you use the import?

For memcpy it would be like:

```

Code byte[] ToCopy = new byte[] { 10, 55, 21, 44 };
byte[] CopyTo = new byte[ToCopy.Length];
fixed (byte* toCopy = ToCopy, copyTo = CopyTo)
{
    memcpy(copyTo, toCopy, (uint)ToCopy.Length);
}
foreach (byte b in CopyTo)
    Console.WriteLine(b);

```

I don't think there is much more to explain about PInvoke and I know it was a weak subject, but I

could of course cover how to create a DLL for PInvoke, but this is not a C++ tutorial, so I'd rather not do that and just end it like this.

?

Downloads/Links:

pinvoke.net

[PInvoke Download \(Source\)](#)

BaussHacker - *October 22, 2012, 2:59 am*

Windows Forms

Introduction:

In This category you will be learning about Windows Forms & Basic Application Design. This category will cover Windows Form Applications.

Chapters:

Windows Forms Introductions

User Controls

Messageboxes

Labels

Buttons

Textboxes

Comboboxes

Checkboxes & Radiobuttons

Listboxes

Pictureboxes

Timers

Threads & Invoking

Windows Forms Introductions

Introduction:

In this chapter you'll be introduced to Windows Forms and what they are.

So now it's time to make some "real" applications. Yes that's right, we'll be looking into Windows Forms now. Though here comes the bad thing for me, as I can't get Visual Studio working on my sisters laptop (Which I am stuck with at the moment as mine broke X_X) then I can only use SharpDevelop which is another C# IDE. It's not that different though, but I won't be doing an image

tutorial though as it wouldn't do much difference as we'd probably be using different IDE's. However it shouldn't make any difference coding wise, so there isn't really that much of difference as the SharpDevelop is close to the older versions of Visual Studio! ;)

What about we get started? Well first of all what is the difference on the console applications we've done earlier and then windows form applications? Well a Windows Form application has a form for a GUI which means we can actually design an application rather than relying on text only then we can rely on graphics as well. Windows Forms uses the GDI+ graphics to draw itself, controls etc. with.

A windows form is basically build up by events and drag/drop of controls. There isn't actually any advanced stuff to it, but the general knowledge which you have got in earlier tutorials will be a great help here!

To create a Windows Form basically create a new prjct and choose Windows Forms.

You'll see your form and then a toolbox. From your toolbox you can drag different controls and you can set different properties of the controls in the property window. You can also set properties of the form itself.

Every control has different events for when different stuff is done. You can also see those events from the property window and if you wish to create an event for a certain thing that's done with that control then you just click on the "empty" box beside the event name and it will create the event by itself.

Some controls has default events as well which will be created when double clicking them. An example could be the form itself. If you double click it then it will create an event for when the form is loaded. Another example could be a button. When you double click it then it will create an event for when the button is clicked.

I will not go in depth with the controls for windows forms and only take the very basal controls and explain. I think it's better if you actually explore a bit about properties etc. yourself here, so that's why I'll leave the more "advanced" things out and only take the very basics of Windows Forms.

Downloads/Links:

[GDI+](#)

User Controls

Introduction:

In this chapter we'll be looking at User Controls.

There is something called UserControl in C# which is a bit different from a form. You can add a new user control by clicking at your project and choosing new item and then finding the user control. When you've done that you'll notice that it looks similar to the form except for that it doesn't have the controlbox / border around it. Well a UserControl is just a control and they're useful to making a "panel" of controls that you may have to use on another form or more than once in a single form. Before you can drag the user control as an item from the toolbox you would have to rebuild your project.

So what was a usercontrol actually? It's a control that can contain more controls within itself and then it can be used as a single control to use more than once in either one or mre forms!

I don't think I need to demonstrate anythings for this as it's only basal information.

Downloads/Links:

[MSDN](#)

Message-boxes

Introduction:

In this chapter we'll take a look at messageboxes.

A messagebox is a form of a dialog, though there is different kind of messageboxes as well different ways to use them. A messagebox can be called from the MessageBox class using the MessageBox.Show() method. It has different parameters which defines what kind of messagebox we're dealing with.

I won't explain the different types and only show examples. The Show() method returns an enum as the DialogResult which is equal to the action which is done by the user. An example would be if we had a messagebox with a "Yes" and a "No" button then the button the user was clicking would be the result of the dialog result.

Message only:

```
Code MessageBox.Show("Hello Hack Forums!");
```

Message and title (caption) only:

```
Code MessageBox.Show("Hello Hack Forums!", "Hack Forums");
```

Information:

```
Code MessageBox.Show("Hello Hack Forums!", "Hack Forums", MessageBoxButtons.OK, MessageBoxIcon.Information);
```

Warning:

```
Code MessageBox.Show("Hello Hack Forums!", "Hack Forums", MessageBoxButtons.OK, MessageBoxIcon.Information);
```

Error:

```
Code MessageBox.Show("Hello Hack Forums!", "Hack Forums", MessageBoxButtons.OK, MessageBoxIcon.Error);
```

Yes/No:

```
Code      switch (MessageBox.Show("Say yes/no!", "Yes/No Example",
MessageBoxButtons.YesNo))
    {
        case DialogResult.Yes:
            MessageBox.Show("That's a good choice!");
            break;
```

```
        case DialogResult.No:
```

```
            MessageBox.Show("That's a bad choice!");
            break;
    }
```

Yes/No/Cancel:

```
Code      switch (MessageBox.Show("Say yes/no!", "Yes/No Example",
MessageBoxButtons.YesNoCancel))
```

```
    {
```

```
        case DialogResult.Yes:
```

```
            MessageBox.Show("That's a good choice!");
            break;
```

```
        case DialogResult.No:
```

```
MessageBox.Show("That's a bad choice!");
break;

        case DialogResult.Cancel:
            MessageBox.Show("What do you think you're doing??!!");
            break;
    }
}
```

I think that's it about message boxes as there isn't really too much to say.

Downloads/Links:
N/A

Labels

Introduction:
In this chapter you'll be introduced to labels.

Basically a label is text drawn as a control. You can use labels to have text on your form! A label has a lot of properties such as foregroundcolor (The color of the text), backgroundcolor (The color of the background, which can be transparent!), text (The text obviously.) and such things. A lot of properties are similar for all the controls, because they're all inheriting the control class.

Try to drag a label to your form. Click one time on it and then you'll see its properties. Find "Text" and change it to whatever you want. I will change mine to "Hello Hack Forums!". If you want you can try to mess around with other properties to see what they do as it's a good way to practice as well!

Try to run your program and you'll notice that the text is drawn to the form, so you cannot mark or anything as if it was in a textbox.

Now let's look at the first part of events if you haven't tried to explore yourself! ;)

Double click the label and it should create a click event. If not just open the properties and switch to the event tab (Looks like some lightning) and then find the "Click" event and double click it.

You should get this then: (Or something similar)

```
Code    void Label1Click(object sender, EventArgs e)
        {

        }
```

Within that method you can handle all clicks done on the label. Let's call a messagebox saying something here.

```
Code    void Label1Click(object sender, EventArgs e)
        {
            MessageBox.Show("Hello Hack Forums!");
        }
```

Now every time the label is clicked that messagebox will show.

I don't think I'll explain more about labels.

Downloads/Links:
N/A

Buttons

Introduction:

In this chapter you'll be introduced to buttons and its different properties.

Now let's take a look at buttons. We'll look at the click event as well as a few properties. As I've already demonstrated the click event (For label) then you should be able to understand it here as well!

First of all start by dragging a button to your form.

Now in the properties find Text. You can change that to whatever you want the button's text should be. As you can see now the text is "button1".

The button does already have the click event as default, so just double click on the button to create that event.

It should look something like below.

```
Code    void Button1Click(object sender, EventArgs e)
        {

        }
    }
```

In there you can do whatever you want that should happen, when a button is clicked.

I'll just make mine create a messagebox as of now.

```
Code    void Button1Click(object sender, EventArgs e)
        {
            MessageBox.Show("Hello Hack Forums!", "Button Event Test!");
        }
    }
```

And as you can see this would be the result!

What about changing the design of the button? You can change the different flatstyle of it which will make the button look different.

Flat:

Standard:

Popup:

System:

I think that's about it what I want to explain about buttons.

Downloads/Links:

N/A

Textboxes

Introduction:

In this chapter you'll be introduced textboxes and its properties.

Okay now it's time for textboxes. I won't explain a lot about the rich textbox as the different is just that the rich textbox can have formatted text, images and such. You can research about that one yourself!

Drag a textbox control from the toolbox and into your form.

If you click on it then you can see its properties. If you find the "Text" property then that would be the text displayed in the textbox as default. You don't necessary have to display any text at all.

Now if you want to use the input of the textbox you'll just call the "Text" property from your textbox.

If your Textbox's name is textbox1 then you'd call it like:

Code `textbox1.Text`

Example:

Code `string input = textbox1.Text;`

However you could also set the text of the textbox.

Example:

Code `textbox1.Text = "Here is some text.";`

You can also change the borderstyle of the textbox. I don't think I'll explain much more about textboxes as there isn't really a lot to say about it.

Downloads/Links:

N/A

Comboboxes

Introduction:

In this chapter you'll be introduced to comboboxes.

A combobox is basically a dropdown control.

Start out by dragging a combobox to your form and click on it. Find the "Items" property and click it. Each line you write will represent an item in the combobox.

However you can also add the items to the combobox from your code. Just call the `Items.Add` from your combobox and same goes for removing. It would be `Items.Remove`.

Example:

Code `comboBox1.Items.Add("item1"); // adds item1`

`comboBox1.Items.Remove("item1"); // removes item1`

To get the selected value of the combobox there is a few things you can choose. You can use the `selectedIndex` or get the actualy select vaue using `selecteditem`. However the selected item will return an object, so unless it's nothing but a string then convert it to a string.

Code `int selectedIndex = comboBox1.SelectedIndex;`

```
string selectedvalue = comboBox1.Items[selectedindex];...
```

```
Code string selectedvalue = comboBox1.SelectedItem.ToString();
```

You can also use the `selectedIndex` to select a specific index or `selecteditem` to select a specific item.

```
Code comboBox1.SelectedIndex = 0; // first item in the combobox...
```

```
Code comboBox1.SelectedItem = comboBox1.Items[3]; // #4 items as a combobox's items are 0 indexed.
```

If you do not wish to let people write values in your combobox you can set it the `"DropDownStyle"` property to `"DropDownList"`.

Result:

I won't be explaining more about comboboxes for now.

Downloads/Links:

N/A

Checkboxes & Radiobuttons

Introduction:

In this chapter you'll be introduced to checkboxes and radiobuttons.

Okay now it's time to look at checkboxes and radiobuttons. The difference on checkboxes and radiobuttons is that a checkbox will only handle itself while radiobuttons will only have the ability to have one checked at a time. While it's not exactly true you can force your program to check more than one radiobutton, but that's not regular to do so, that's what checkboxes are for.

Let's look at checkboxes first. Try to drag a few checkboxes down and find the `"Text"` property of them all. It'll be the text displayed beside the checkbox.

Here is how mine looks:

Now what if we want a checkbox to be checked as default?

Find the `"Checked"` property and change it to true.

You can also handle things when a checkbox is checked. It's also the default event of the checkbox, so just go ahead and double click it.

You should get something similar to this:

```
Code      void CheckBox2CheckedChanged(object sender, EventArgs e)
          {

          }
```

Whenever that method is called then the checkbox has either been checked or unchecked.

You can use the `"Checked"` property to check if it's checked or unchecked.

```
Code      if (checkBox2.Checked)
          {
            // the checkbox is checked
          }
```

```

else
{
    // the checkbox is not checked
}

```

You can also set the value whether it's checked or not by casting a value to the "Checked" property.

Code `checkBox2.Checked = false;` // unchecking the checkbox

Now let's look at radio buttons.

Try to drag a few radio buttons into your form.

Here is mine:

Just like the checkboxes they have a "Checked" property which indicates whether they're checked or not, but usually radio buttons are used to gain only one checked radio button. That's why if you go ahead now and try to select all then you'll just keep switching between them. In my case I could have placed a question above in ex. a label saying "What is your favorite color?"

There isn't really much to tell about radio buttons except what's explained at checkboxes as well. The only difference really is the look of them and that you can select multiple checkboxes, while you can only select a single radio button (Unless you force it to do something else.)

That was a bit about checkboxes and radiobuttons.

Downloads/Links:

N/A

Listboxes

Introduction:

In this chapter you'll be introduced to listboxes.

Now it's time to look at listboxes. A listbox is just a list of items whom you can select and depending on what's selected you could do different things.

Drag a listbox to your form first of all. Now find the "Items" property. Remember the way it works from comboboxes? Each line represents an item. That's the same here.

Now if you double click the listbox you should get the default event of it which is called when an index of the listbox is changed.

Should look something like this:

```

Code      void ListBox1_SelectedIndexChanged(object sender, EventArgs e)
{
}

```

That means it won't get you the selected value, but the index. You can use that index to get the item from the listbox though.

```

Code      int selectedIndex = listBox1.SelectedIndex;
          string selectedvalue = listBox1.Items[selectedIndex];

```

You can also set the selectedIndex and such in the listbox, just like from comboboxes lol. Notice I'm not comparing those two as they're very different, just the similarities regarding "Item-indexing" as it's the same for most controls containing items.

You can also add/remove items from the listbox, which again is basically the same way.

```
Code      listBox1.Items.Add("item1"); // adds item1
          listBox1.Items.Remove("item1"); // removes item1
```

That was a bit about listboxes.

Downloads/Links:

N/A

Pictureboxes

Introduction:

In this chapter we'll take a look at pictureboxes.

Basically a picturebox is just a box displaying a picture. It can also display a gif image, but a gif image is just a frames of images. Same goes for videos, so technically you could make a wrapper to make it display ex. an avi video. Would require you to study the file structure of an AVI file though, but enough of that.

Drag a picturebox into your form.

You can set an image to it by using either the "BackgroundImage" or "Image" property. You can also use both if you wish. The properties will both import your image into the Resources of your application.

I've never really been a fan of that as the graphics will just fill in the application as well, so I've always initialized the images from the actual image file.

You can also do that, if you wish to and basically you'd just call the property of the image and then `System.Drawing.Image.FromFile("filename")`, but you can also cast a bitmap etc. to it, if that's your wish.

Example:

```
Code pictureBox1.Image = Image.FromFile("mypicture.png");
```

Technically you can use a picturebox as a button as well. The default event for a picturebox is the click event, so just double click your picturebox and you should get something like this:

```
Code      void PictureBox1Click(object sender, EventArgs e)
          {

          }
}
```

Everytime the picturebox is clicked that event is called then.

I think that's what I'll explain about pictureboxes.

Downloads/Links:

N/A

Timers

Introduction:

In this chapter you'll be introduced to timers.

You have already heard about threading. Timers are a bit different, though they can allow you to do something "similar" to multithreading. I would suggest using threads whenever it's possible, but if

you need to invoke your form a lot because of that then it's actually better to use timers as they're already working on the main thread and does not require invoke and invoking to the main thread will do no difference for threading then.

Just drag a timer to your form and you'll notice a small icon in the bottom of your design view. In the properties you can see "Interval" which is how many milliseconds there should be between each execution of the timer event. Then there is the "Enabled" property and if it's set to true then it will start as soon as the form has been initialized.

You can also start the timer by yourself and stop it as well.

```
Code      timer1.Start(); // starts the timer
          timer1.Stop(); // stops the timer
```

To handle the timer events just double click your timer and it will create its default event which is the "Tick" event. That event is executed every time the last interval of the events execution has passed.

It should look something like this:

```
Code      void Timer1Tick(object sender, EventArgs e)
          {

          }
```

In there you can handle whatever you want to happen on repeat.

Example:

```
Code      void Timer1Tick(object sender, EventArgs e)
          {
            label1.Text = DateTime.Now.ToString(); // sets label1 to the current datetime.
          }
```

I think that's it about timers.

Downloads/Links:

N/A

Threads & Invoking

Introduction:

In this chapter you'll be introduced to using threads with Windows Form as well invoking.

If you haven't tried it before then try to make a new thread and call any control in your application. You'll notice that an exception will be thrown, because of illegal cross threading. Well that's a good thing there is this and the reason why it has been made it's because what if you tried to access a control the same time from 2 threads and then tried to change it as well, that would cause some strange errors and for that reason you're not allowed to do that. You can always set "CheckForIllegalCrossThreading" to false, but it's only there for debugging reasons, but to get around this problem the real way you would have to invoke either the form or the control you want to invoke.

Every control and the form itself has a method called "Invoke" which can be used to do this.

However you should only invoke if it's required and for that you can call a boolean within them called "InvokeRequired" which checks if it's required.

If it's not required to invoke then you can do whatever you want with the control without invoking it.

Here is an example of invoking a label from another thread.

```
Code    new System.Threading.Thread(() =>
    {
        if (this.InvokeRequired)
        {
            this.Invoke(new MethodInvoker(() =>
                {
                    label1.Text = "Hello Hack Forums!";
                }));
        }
        else
        {
            label1.Text = "Hello Hack Forums!";
        }
    }).Start();
```

A good idea when using invoke is to invoke it in another thread than the thread you're invoking from, but don't create a new thread for every invoke, but instead create a wrapper with a single thread handling all invokes.

Here is an example of such thing: (Can probably be optimized as well.)

```
Code    /// <summary>
        /// Delegate used to handle safe invoke events.
        /// </summary>
        public delegate void SafeEvent();

        /// <summary>
        /// A class to handle safe invokes on Windows Form Applications. It can be used to handle
        invokes on cross-threading problems as well it won't slow any threads down. It runs over its own
        thread.
        /// </summary>
        public class SafeInvoker
        {
            private static ConcurrentDictionary<int, SafeInvokeEvent> safeEvents;
            private static SafeRandom safeRandom;

            /// <summary>
            /// Event invoked by the safe invoker.
            /// </summary>
            private class SafeInvokeEvent
            {
                public bool Invoked = false;
                public SafeEvent Event;
                public Control Ctrl;
            }

            static SafeInvoker()
            {
                safeEvents = new System.Collections.Concurrent.ConcurrentDictionary<int,
                SafeInvokeEvent>();
                safeRandom = new SafeRandom();
                new Thread(new ThreadStart(Safe_Thread)).Start();
            }
        }
```

```

/// <summary>
/// Invoking a control/form with an event.
/// </summary>
/// <param name="ctrl">The control/form to invoke.</param>
/// <param name="safeEvent">The event to invoke.</param>
public static void Invoke(Control ctrl, SafeEvent safeEvent)
{
    SafeInvokeEvent siEvent = new SafeInvokeEvent();
    siEvent.Event = safeEvent;
    if (ctrl != null)
        siEvent.Ctrl = ctrl;
    int id = safeRandom.Next();
    while (!safeEvents.TryAdd(id, siEvent))
        id = safeRandom.Next();
}

private static void Safe_Thread()
{
    while (true)
    {
        foreach (KeyValuePair<int, SafeInvokeEvent> sEvent in safeEvents)
        {
            if (!sEvent.Value.Invoked)
            {
                if (sEvent.Value.Ctrl != null)
                {
                    if (sEvent.Value.Ctrl.InvokeRequired)
                        sEvent.Value.Ctrl.Invoke(new MethodInvoker(() =>
                        {
                            sEvent.Value.Event.Invoke();
                        }));
                }
                else
                {
                    sEvent.Value.Event.Invoke();
                }
            }
            else
            {
                sEvent.Value.Event.Invoke();
            }
            sEvent.Value.Invoked = true;
        }
        SafeInvokeEvent rEvent;
        safeEvents.TryRemove(sEvent.Key, out rEvent);
        Thread.Sleep(50);
    }
}
}

```

Example usage:

```

Code          SafeInvoker.Invoke(this,
              () =>
              {
                  label1.Text = rnd.Next().ToString();
                  label2.Text = rnd.Next().ToString();
              }

```

```
label3.Text = rnd.Next().ToString();
```

```
richTextBox1.Text += rnd.Next().ToString();
```

```
richTextBox1.Text += "\n";
```

```
});
```

SafeRandom would be a thread-safe random which you can find in the Chat Application source! ;)

I don't think I'll explain much more about invoking for now!

Downloads/Links:

N/A

BaussHacker - *October 22, 2012, 3:01 am*

Applications

Introduction:

In this category there will be downloads for different applications in C#. This category will cover Windows Form Applications.

Chapters:

Basic Calculator

Webbrowser Implementation

Process & Website Opener

Screenshots

Simple Chat

Basic Calculator

Features: Multiple Calculations

Add, Sub, Div, Mul

Decimal Support

Downloads/Links:

[Basic Calculator Download \(Source\)](#)

Webbrowser Implementation

Features: Browse

Go Forward

Go Backward

Downloads/Links:

[Webbrowser Implementation Download \(Source\)](#)

Process & Website Opener

Features:Open a process or a website.

Downloads/Links:

[ProcessAndWebsite Download \(Source\)](#)

Screenshots

Features:Takescreenshot of screen

Threaded invoking

Hide/Showing the form when taking screenshot

Downloads/Links:

[Screenshots Download \(Source\)](#)

Simple Chat

Features:Multiple Chat Clients

Name Change

Both Server/Client

Async sockets

Downloads/Links:

[Simple Chat Download \(Source\)](#)

That was it and I hope you enjoyed all the tutorials and that you have learned something.
Now it's just for you to explore all the things that I have left out in this ;)
Thank you for reading this!

?

Mr Kewl - *October 22, 2012, 3:02 am*

Great thread and good job on this Bauss!