

Forum:[Hack Forums](#)

Thread: [Generic Type Implicit Conversion?](#)

SIM-1984933 - September 30, 2016, 6:36 pm

lets say i have the follow code....

```
public void SetValue<t>(t Val)
{
    BitConverter.GetBytes((t)Val);
}
```

since BC won't accept Generic vars as valid args, how would i efficiently be able to do a implicit conversion with t?

without having to do a if --> typeof().

thanks;

BahNahNah - September 30, 2016, 8:11 pm

you could use reflection

```
Code public byte[] SetValue<t>(t Val)
{
    return (byte[])typeof(BitConverter).GetMethod("GetBytes", new Type[] { typeof(t) }).Invoke(null,
new object[] { Val });
}
or
```

```
Code public static byte[] SetValue<t>(t Val)
{
    byte[] returnBytes = new byte[Marshal.SizeOf(Val)];
    var handle = GCHandle.Alloc(Val, GCHandleType.Pinned);
    Marshal.Copy(handle.AddrOfPinnedObject(), returnBytes, 0, returnBytes.Length);
    handle.Free();
    return returnBytes;
}
```

Ninja - October 1, 2016, 6:29 am

Correct me if I am wrong but can't you do this:

```
Code public void SetValue<t>(t Val) where Val : TypeYouWant
{
    BitConverter.GetBytes((t)Val);
}Or am I understanding the problem wrong.
```

Chickendipper - October 1, 2016, 6:41 am

The only way you would be able to do this is via reflection (as BahNahNah gives the example of). The marshaling example is interesting, however.

SIM-1984933 - October 1, 2016, 8:28 am

(Sep 30, 2016 08:11 PM)BahNahNah Wrote: [\[CITE\]](#) you could use reflection

```
Code public byte[] SetValue<t>(t Val)
```

```
{
    return (byte[])typeof(BitConverter).GetMethod("GetBytes", new Type[] { typeof(t) }).Invoke(null,
new object[] { Val });
}
```

or

```
Code public static byte[] SetValue<t>(t Val)
```

```
{
    byte[] returnBytes = new byte[Marshal.SizeOf(Val)];
    var handle = GCHandle.Alloc(Val, GCHandleType.Pinned);
    Marshal.Copy(handle.AddrOfPinnedObject(), returnBytes, 0, returnBytes.Length);
    handle.Free();
    return returnBytes;
}
```

}thanks.

0xDEAD10CC - October 1, 2016, 5:46 pm

(Sep 30, 2016 08:11 PM)BahNahNah Wrote: [\[CITE\]](#) you could use reflection

```
Code public byte[] SetValue<t>(t Val)
```

```
{
    return (byte[])typeof(BitConverter).GetMethod("GetBytes", new Type[] { typeof(t) }).Invoke(null,
new object[] { Val });
}
```

or

```
Code public static byte[] SetValue<t>(t Val)
```

```
{
    byte[] returnBytes = new byte[Marshal.SizeOf(Val)];
    var handle = GCHandle.Alloc(Val, GCHandleType.Pinned);
    Marshal.Copy(handle.AddrOfPinnedObject(), returnBytes, 0, returnBytes.Length);
    handle.Free();
    return returnBytes;
}
```

I would recommend adding to that:

```
Code class GCHandleObject : IDisposable
```

```
{
    private bool _handleDisposed;
    private GCHandle _handle;

    public object Target => _handle.Target;
    public bool IsAllocated => _handle.IsAllocated;

    public GCHandleObject(object value, GCHandleType type)
    {
        _handleDisposed = false;
        _handle = GCHandle.Alloc(value, type);
    }

    public IntPtr AddrOfPinnedObject()
    {
        return _handle.AddrOfPinnedObject();
    }
}
```

```

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

protected virtual void Dispose(bool disposing)
{
    if (disposing)
    {
        if (!_handleDisposed)
        {
            _handleDisposed = true;
            _handle.Free();
        }
    }
}
}

```

```

public static byte[] GetBytes<T>(T value) where T : struct
{
    byte[] returnBytes = new byte[Marshal.SizeOf(value)];
    using(var obj = new GCHandleObject(value, GCHandleType.Pinned))
    {
        Marshal.Copy(obj.AddrOfPinnedObject(), returnBytes, 0, returnBytes.Length);
    }
    return returnBytes;
}

```

If not taking on the RAII-like pattern, at least constrain your type parameter to what `Marshal.SizeOf()` expects it's arguments to be.