5.17.11 : Equivalent String Methods in NET

Len has its equivalent in the length method. ***Ucase*** and ***Lcase*** have their equivalent in ***ToUpper*** and ***ToLower*** respectively.

Old style	New Style
T$ = Ucase$(t$)	***T= T.ToUpper()***
T$ = Lcase$(t$)	***T = T.ToLower()***
X = Len(t$)	***X = T.length***

These methods can be invoked in both the old style and new style but the upgrade wizard will change them to the new style as later versions of Visual Basic might do away with the old style operators.

5.17.12 : Padleft and PadRight methods (NET only).

Padleft and ***PadRight*** allow you to add white space to either the left side or right side of the string. You specify how much white space is needed. In the event that the string is already equal to or longer than the desired size no operation is performed.

```
F = "Hello"
F = F.Padleft(10)  ' F now contains "     Hello"
F = F.Padright(15) ' F now contains "     Hello     "
```

5.17.13 : Insert and Remove (NET only).

These commands allow you to insert and remove sections of a string. The old style constructs ***Mid$***, ***Left$*** and ***Right$*** can perform a similar operation if you write some surrounding code.

```
A = "This is a string"
B = "manipulated "
C = a.insert (8, B)
' C now contains the string "This is a manipulated string"
```

In a similar fashion, the remove operation can remove a chain of characters from a string.

```
A = "This is a string"
B = "manipulated "
C = a.insert (8, B)
' C now contains the string "This is a manipulated string"
C = c.remove (8, 12)
```

The remove method takes two arguments: the starting position in the string and the number of characters that need to be removed. In the above example, 12 characters are removed starting at the eighth position in the string.

5.17.14 : Join and split (NET only).

These methods allow you to store elements of an array in a string and specify a character to be use as separator.

```
Dim Muffins() as string = {“Blueberry”, “Chocolate Chip”, “Cinnamon”}
Muffinlist = string.join(“,”, muffins)

‘ muffinlist contains a string the looks like this
‘ Blueberry,Chocolate Chip,Cinnamon
```

This string can now be passed as an argument to a subroutine, or even used as return value from a function. Passing arrays as a return value is hard and these methods give you an escape path.

To break the string into separate units you can use the split method.

```
Dim desserts() as string
desserts = string.split (muffinlist, “,”)
dim dessert as string
for each dessert in desserts
    debug.print dessert
next
```

5.18 : File Manipulation under Visual Basic Classic

During your programming work, you will often find yourself in a situation where you need to store something to disk or retrieve it; the information can be all sorts of data, whether it is numerical, text, binary or even an entire database with linked lists, records, custom styles etcetera. Well, you could have not picked a simpler language. Basic in general is probably the only language in which file manipulation is so simple, yet at the same time so extended. This chapter might look overwhelming at first but that is just because there are so many things you can do with files. Since file input and output is one of the most used operations when writing software, I tried to be as complete as possible. This is material that you can read as-you-go. You can easily skip this chapter if you are not going to deal with files yet and come back later.

Files are referenced to using ‘handles’. A handle is a storage space that the computer uses to remember where the file resides physically on disk, at what position you are reading, and what the current file status is. This handle points to the information that defines the file to the operating system.

Chapter 20 : Building better programs

What is a better program? That strongly depends on the definition of a good program. How can you classify a program as good? If it is better then a bad program. So it all comes down to defining bad programs. Then what are bad programs?

Well, I can give you many examples:

- ⇒ A program that crashes very often;
- ⇒ eats memory and does practically nothing;
- ⇒ is terribly slow;
- ⇒ looks awfully ugly (depends strongly on personal taste, though);
- ⇒ behaves strangely some times;
- ⇒ corrupts and wastes your data.

I think we can all largely agree on the above. Well except maybe the 'Looks awfully ugly'. This depends on personal taste. And I am not going into that one. After all this is a book on programming, not on style.

So what can we do to make faster, smaller, more stable programs?

The very first step is taken during coding process. Write clean source code! Do not make constructions that you yourself hardly understand. Insert comments. It does not hurt and will not waste memory or speed once compiled. And most of all: adhere to the KISS principle.

20.1 : The KISS Way

No, I am not asking you to kiss your computer. KISS is the abbreviation for 'Keep It Simple Stupid'. It means you need to write code that is as simple to understand as possible. Do not write 'complex' things like:

```
X=0
Doagain:
IF x < 4 then x=x+1 else x=x-1 ; If X=4 then goto stopit else goto _
Doagain; Stopit: End
```

Any clue what this is doing? Let's have a look at this. First, let us write it out so that it becomes clearer.

```
X=0
Doagain:
  IF X<4 then
     X=x+1
```

```
    Else
       X=x-1
    End if
    If x = 4 then
       goto Stopit
    Else
       Goto Doagain
    End if
  Stopit:
    End
```

Well, at least it has turned out a bit better. But still, what does it do? Let's analyze.

If ***X*** is smaller than four, it gets incremented with one. If it is equal or larger, it is decremented with one. Next, ***x*** is compared against four. If it is four, we jump to the label ***Stopit***. If it is different from four we do it all over again. If you think a bit more about this code then you will see that the decrementing part never is executed. When you start at zero the first ***if-then-else*** will be executed while ***x*** is smaller than 4. When X is 3 the decision of the ***If-Then-else*** will increment ***X*** to 4 and the next ***If-then–else*** will jump to ***Stopit***. Therefore, ***X*** will never be decremented. We just found our first piece of DEAD code. Dead code is code that consumes memory but does absolutely nothing. It does not even get executed! Now the optimizer in the compiler can catch and eliminate dead code as long as we are talking about entire procedures or functions that are never called. However, it cannot eliminate the above case of dead code.

Therefore, we could rewrite the code as follows:

```
X=0
Doagain:
  IF X<4 then
    X=x+1
  End if
  If x = 4 then
    goto stopit.
  Else
    Goto doagain
  End if
Stopit:
End
```

If we now look again, we will see that all we are doing here is incrementing X until it reaches 4. So why not use a ***While-Wend*** construction?

```
X=0
While x <4
  X = x+1
Wend
End
```

Now isn't that a quite a bit more readable?

This demonstrates a number of basic KISS principles:

GPIBcore: This library acts as the traffic controller for all GPIB operations. It manages all operations on the GPIB bus and offers error and status reporting via the standard debugging console of Visual Basic.

GPIBspy: Monitors all GPIB operations during runtime, and offers a debugging window to take control over GPIB operations when necessary

GPIBweb: Allows you to redirect GPIB operations via TCP/IP protocol to anywhere in the world. This can be used, not only to run, but as a monitor to control and debug test-setups remotely.

Instrument Library: A collection of instrument specific routines that ease the control of GPIB machines.

IO library: A collection of routines to control plug-in data-acquisition boards.

ClassWork Module: A Class holding all information to control an instrument or IO channel (Printer port, DAQ board etc). This exposes the hardware as an object to the programmer. Instruments and IO channels can be treated just like any other Visual Basic object.

TestBench Control: An ActiveX control that can interface to a ClassWork module. TestBench offers a rapid way of adding a GUI for your instruments or IO channels.

34.1 : GPIBcore

GPIBcore is a GPIB handler for Visual Basic written in Visual Basic. It handles basic GPIB operations in cooperation with the GPIB card driver provided with the board manufacturer. However, why not use the vendor-supplied driver directly? Well, a number of problems arise in doing that.

- The card drivers are board vendor specific.
- Initialization code is vendor dependent.
- Command set is different.
- Capabilities are different.

The command set might be extensive and sometimes hard to understand. Many card drivers contain so many routines that it becomes hard trying to understand what function you need and when you need it. Furthermore, you will need to understand the GPIB bus before you can talk to instruments. Not all machines respond in a uniform manner to GPIB operations.

This is where GPIBcore kicks in. It physically isolates the card-level operations from the programmer. At the same time, it takes care of all the low-level work related with device initialization and management. When different cards are to be used, or when the card driver changes completely, all you have to do is adapt the GPIBcore.

Note	GPIBcore relies on the GPIB32.DLL to access the GPIB bus. Any interface board that has followed this driver can thus be used without adaptation. Almost all GPIB boards in existence use this common GPIB32.DLL.

34.1.1 : GPIBcore features

GPIBcore sends a lot of debugging information to the standard debug console of Visual Basic. This should assist you in debugging your own instrument drivers. It is possible from the interactive window in visual basic to take control over the entire GPIB bus and interactively test operations. Besides GPIB functionality the GPIBcore also includes hardware IO functionality, binary data handling routines as well as some other missing instructions such as LOG10 calculations.

34.1.2 : Installing GPIBcore

All you need to do in order to use the GPIBcore, is simply load GPIBcore.BAS into your project, and call GPIBinit prior to executing any GPIB related command. It is strongly suggested NOT to modify GPIBcore in any way. Keep your own routines in your libraries. If an update should be made to GPIBcore, this will avoid backwards compatibility problems. Although it is possible that GPIBcore could contain programming errors, the core has been tested extensively and no known bugs are present.

34.2 : GPIBcore programming guide

The core can be divided in roughly 3 parts. The real GPIB related operations, The Hardware IO operations (via Win95io) and the supporting functions.

34.2.1 : GPIB functions

All GPIB management is handled by dedicated commands. All commands begin with GPIB and have meaningful variable declarations to facilitate programming.

34.2.2 : GPIBinit

Syntax:

```
GPIBinit
```

Description:

This function initializes the GPIB stream. It sets up communication with the card vendor specific driver. It resets the GPIB subsystem of the computer, and unlocks all attached devices