

Visual Basic

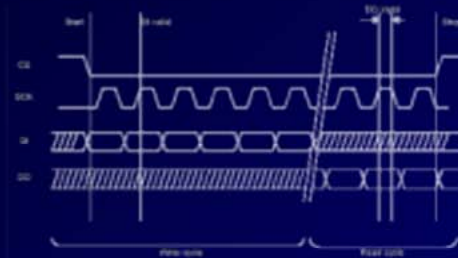
```
Dim MyPSU as new HP6624 ' Create an object of class HP6624
Dim MyDVM as new HP34401 ' Create object from HP34401 class
```

```
Sub Form_load()
    GPIBinit
    MyPSU.address = 5 ' address of the supply
    MyPSU.Assignto = 1 ' use first channel of this supply
    MyDVM.address = 22 ' address of the DVM
    MyDVM.CurrentDC ' select Current DC range
```

```
Set DVM.Target = MyDVM ' TestBench link to ClassWork
Set PSU.Target = MyPSU ' TestBench link to ClassWork
```

```
End Sub
```

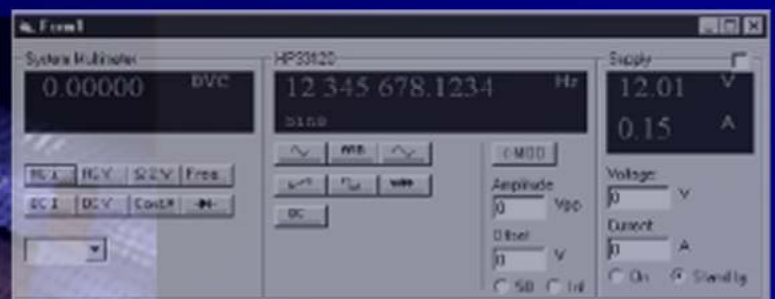
```
Sub Quitprogram
    bye
end
End sub
```



for Electronics Engineering Applications



Vincent Himpe



Second Edition

Visual Basic for Electronics Engineering Applications

*A Crash Course To The World's Premier
RAD Tool.*

Developing R&D test systems in no time.

Copyright notice.

Visual Basic for Electronics Engineering Applications

The book is released as public domain. Reprint, sale or re-sale is prohibited.

Copyright 1999 – 2005 Vincent Himpe.

The author recognizes the copyrights, trademarks and registered trademarks of all products mentioned in this book.

Initial edition 5-9-1999

Second revised edition 09-03-2002

E-mail: vincent_himpe@hotmail.com

Index table

Copyright notice.....	2
Index table.....	3
<i>Visual Basic For the Research & Development LAB.....</i>	<i>19</i>
Part I : The Basics of Visual Basic	19
Introduction.....	19
Conventions used in this manual	22
Monotype Bold.....	22
<i>Chapter 1 : The Visual Basic Background</i>	<i>23</i>
1.1 Windows.....	24
1.2 Object Oriented Programming.....	25
1.3 What OOP does for you	26
1.4 Overview of the definitions:.....	26
<i>Chapter 2 : Exploring the Visual Basic environment</i>	<i>29</i>
2.1 Starting a Visual basic project.....	29
2.2 The programming environment.....	31
2.2.1 Using The Menu-bar	31
2.2.2 Accessing functions with the Toolbar	31
2.2.3 The Object Browser (The Toolbox)	32
2.2.4 The project navigator.....	32
2.2.5 The properties navigator.....	32
2.2.6 Form Layout Window	32
2.2.7 Form Viewer.....	33
2.2.8 Code Viewer.....	34
2.2.9 The Help system.....	35
<i>Chapter 3 : The Basic Objects and Controls.....</i>	<i>37</i>
3.1 The Form	37
3.2 The Controls	38
3.3 The Standard controls inside Visual basic.....	40
3.4 Common Controls	41
3.5 Common Dialog Control	42
3.6 Comm Control.....	42
3.7 Menu's.....	43
3.8 Properties.....	44
3.8.1 Name	45

3.8.2 Top, Left, Height, Width	45
3.8.3 Backcolor, ForeColor, Textcolor.....	45
3.8.4 Caption and Text	46
3.8.5 Enabled and Visible.....	46
3.8.6 Index.....	46
3.8.7 Tabindex.....	47
3.8.9 TooltipText.....	47
Chapter 4: Events and Methods	49
4.1 Tapping into Events.....	49
4.1.1 Click (Most controls).....	50
4.1.2 DbClick (Most controls).....	50
4.1.3 KeyPress (Most controls)	51
4.1.4 MouseMove (Most controls)	51
4.1.5 Activate (Form)	52
4.1.6 Deactivate (Form).....	52
4.1.7 Load (Form)	52
4.1.8 Unload (Form).....	52
4.1.9 Change (Textbox).....	53
4.2 Methods	54
Chapter 5 : The Basic language itself.....	57
5.1 Variables.....	59
5.1.1 Available Types in Visual Basic and how to declare them	60
5.2 Arrays	61
5.2.1 DIM	62
5.2.2 ReDim	63
5.2.3 Ubound	64
5.2.4 Lbound	65
5.2.5 Array.....	66
5.3 Types	69
5.4 Scope of Variables.....	71
5.4.1 Public / Global.....	72
5.4.2 Private.....	72
5.4.3 Static	73
5.5 Module level scope.....	73
5.6 Subroutines and Functions	74
5.6.1 Subroutines or Procedures.....	75
5.6.2 Functions	75
5.7 Scope of procedures	76
5.8 Constants	76
5.9 Numerical Operators	79
5.10 Base conversion.....	80

5.11 Logical Operators	81
5.12 Flow Control.....	81
5.12.1 If then else	82
5.12.2 If-then-else / elseif.....	83
5.12.3 Select case	84
5.12.4 Loop Constructions	85
5.12.5 For Next.....	85
5.12.6 While wend.....	86
5.12.7 Do Until.....	87
5.13 String manipulation Left\$ - Right\$ - Ltrim\$ - Rtrim\$.....	88
5.13.1 Left\$	88
5.13.2 Right\$	88
5.13.3 Mid\$	88
5.13.4 Ltrim\$ / Rtrim\$ / Trim\$.....	89
5.13.5 Ucase\$	89
5.13.6 VAL and STR\$.....	90
5.13.7 LEN	90
5.13.8 INSTR	91
5.14 File Manipulation (Open – Close – Print – Input).....	91
5.14.1 Basic structure to open a file.	92
5.14.2 Output mode	93
5.14.3 Append mode	93
5.14.4 Input mode.....	94
5.14.5 Storing something in a file	94
5.14.6 PRINT constructions (file I/O).....	95
5.14.6.1 Data list Style	95
5.14.6.2 String style.....	96
5.14.7 Reading from a file.....	96
5.14.8 Determining file end.....	97
5.14.9 File names.....	97
Chapter 6 : Creating a user interface.....	99
6.1 Creating The Form	99
6.2 Arrays of Objects and Controls	101
Chapter 7 : Attaching code to your form	103
7.1 Attaching code to objects	103
7.2 Let's Attach some code	104
Chapter 8 : Running and debugging a program	107
8.1 Running a program.....	107
8.1.1 Start , Break , Stop.....	107
8.2 Debugging a program.....	108

8.3 Examining Variables	109
8.4 Advanced Debugging : The Watch Window.....	110
8.4.1 Window Elements	110
8.4.2 Add Watch command	111
8.4.3 Add watch dialog box.....	112
8.4.4 Quick Watch command (Shift F9).....	113
8.4.5 Quick watch dialog box.....	114
8.4.6 Edit Watch command	115
8.4.7 Edit Watch Window	115
8.5 Using Breakpoints	116
8.6 the Debug Object.....	117
Chapter 9 : Distributing a program.....	119
9.1 The First steps	119
9.2 Specifying the Media.....	122
Chapter 10 : Multi-module projects.	125
10.1 Multiple Forms	125
10.2 Modules	127
10.3 Accessing items from other parts of the program.....	127
10.4 Root structure analogy of a project	129
Chapter 11 : A couple of case studies	131
11.1 Case Study 1 : A small Text Editor	133
11.1.1 Attaching Code.....	136
Case Study 2 : A Calculator	143
11.2.1 Designing the user interface	143
11.2.2 Writing Code	147
11.2.3 Attaching code to the user interface	149
Part II :	157
The Advanced World of Visual Basic	157
Introduction to Part II	157
Chapter 12 : One step beyond.....	159
12.1 Forms.....	159
12.2.1 Load.....	159
12.2.2 Unload	160
12.2.3 Show	162
12.2.4 Hide.	164
12.2.5 Modal / Modeless forms	164
12.2.6 MDI forms	166
12.2 Menu's.....	167

12.2.1 Popup menu's	167
12.2.2 Adding images to menu's	169
12.3 Modifying menus from code	170
12.3.1 Enabling and Disabling Menu Commands	170
12.3.2 Displaying a Check Mark on a Menu Control	171
12.3.3 Making Menu Controls Invisible	172
12.3.4 Adding Menu Controls at Run Time	173
12.4 Special Menu features	173
12.4.1 WindowList	174
12.4.2 Negotiating menu's	175
12.5 Option Selectors	175
12.5.1 CheckBoxes	176
12.5.2 OptionButtons or Radio Buttons	176
12.5.3 Grouping Radio Buttons	177
12.5.4 Listboxes	177
12.6 Timer objects	180
12.7 User entry objects	182
12.7.1 Textboxes	182
12.7.1.1 Locked and Enabled	182
12.7.1.2 Keypress Event	182
12.7.1 Combobox	183
12.8 Printing	185
12.9 Taking Advantage of the Windows95 Look	186
Chapter 13 : Graphics.....	189
13.1 Basic coordinate operations	189
13.1.1 CurrentX, CurrentY	190
13.2 Drawing setup	190
13.2.1 Drawwidth	190
13.2.2 Drawmode	191
13.2.3 DrawStyle	193
13.2.4 Fillcolour	194
13.2.5 FillStyle	195
13.3 Drawing primitives	196
13.3.1 PSet	196
13.3.2 Line	197
13.3.3 Circle	198
13.4 Saving and loading graphics	200
13.4.1 Saving Graphics	200
13.4.2 Loading Graphics	202
13.5 Coordinate systems	203
13.5.1 Scale	203
13.5.2 Scalemode	204

13.5.3 ScaleHeight , Scalewidth	206
13.5.4 ScaleLeft and ScaleTop	207
Chapter 14: Communicating to the world around us	209
14.1 SendKeys ...: a simple way of communicating.....	209
14.1.1 AppActivate.....	213
14.1.2 Shell.....	214
14.2 DDE : another means of inter-program communication	215
14.2.1 LinkMode:	215
14.2.2 Linktopic	218
14.2.2.1 Destination Control	219
14.2.2.2 Source Form	220
14.2.3 LinkItem	221
14.3 Serial IO : Talking to world beyond the port.....	224
14.3.1 Inserting the object	224
14.3.2 Portopen.....	226
14.3.3 Handshaking	227
14.3.4 Settings	228
14.3.5 OutbufferSize , InbufferSize	229
14.3.6 OutbufferCount, Inbuffercount	230
14.3.7 Parityreplace	230
14.3.8 DTRenable.....	231
14.3.9 Rthreshold	231
14.3.10 OnComm Event	232
14.3.11 CommEvent.....	232
14.4 Winsock : The world is not enough	234
14.4.1 TCP Basics	235
14.4.2 UDP Basics.....	235
14.4.3 RemoteHost	235
14.4.4 Protocol	236
14.4.5 State	236
14.4.6 Accept.....	237
14.4.7 GetData.....	238
14.4.8 Connectionrequest	239
14.4.9 DataArrival	240
Appendix II : Some more case studies	241
Doodle : A Graphics program	241
Miniterm :A simple terminal	241
AlphaServer : A Telnet Server application.....	241
LoanCalc : Using Excel from your program	241
Case Study 3 : Doodle A graphics program	243
Case Study 4 : The dataterminal.....	251

Case Study 5 : AlphaServe : A Telnet server	259
Case Study 6 : LoanCalc : Using Excel in your applications	261
Conclusion.....	263
Visual Basic For the Research & Development LAB.....	267
Part III :.....	267
Master Programming with Visual Basic.....	267
Introduction	267
Chapter 15: Digging into Windows.....	269
15.1 DLL's	269
15.2 Accessing DLL routines	271
15.3 On Passing parameters to procedures and functions	273
15.4 API programming.....	275
15.4.1 A simple API example.....	276
Chapter 16 : ActiveX Control Creation.....	281
16.1 Creating an ActiveX Object.	282
16.2 Adding property's and events.....	288
16.3 What the wizard came up with	298
15.6 A closer look at the final code.....	300
Chapter 17 : Building better programs.	305
17.1 The KISS Way.....	306
17.2 Atomic Programming	310
17.3 Naming objects.....	312
17.3 Error handling.....	313
17.3.1 The On Error Goto clause.....	314
17.3.2 The Err object.....	318
17.3.3 Resuming execution after handling the error	320
17.3.4 Trappable errors	322
17.3.5 Syntax Errors (errors against the Basic syntax).....	323
17.3.7 Runtime errors	323
17.3.8 Flawed Programming logic errors.	324
17.3.9 File handling errors	325
Chapter 18 : The Windows registry.....	327
18.1 Digging into the registry.....	327
18.2 Data Mining in the registry.....	329
18.2.1 GetSetting	329
18.2.2 SaveSetting	330
18.2.3 DeleteSetting	331
18.3 Make use of the registry	331

Chapter 19 : Scripting interpreters.....	335
19.1 Building A simple script interpreter	335
19.1.1 Running the script	337
19.1.2 The script Parser.....	338
19.1.3 Parameter extraction.....	341
19.2 MSScript : A real script interpreter.	343
19.2.1 Scripting language	344
19.2.2 The MSScript properties	345
19.2.3 Script Control Methods	345
19.2.4 Adding code to the script engine	346
19.2.5 Exposing Objects.....	347
Chapter 20 : Classes.....	349
20.1 The Class concept.....	349
20.2 Creating a Class	350
20.3 Instantiating objects from a class.....	351
20.4 A practical example	352
Appendix III A couple of Case studies.....	355
Killing windows via an API call	355
The LED activeX control	355
The PassBox activeX control	355
MiniBasic : A program editor for MSscript	355
Additional Notes on the use of classes	355
Case Study 7 : Killing Windows via an API call	357
Case Study 8 : The LED ActiveX control	359
Case Study 9 : MiniBasic : A program environment for MSscript	367
Case Study 10 : Additional notes on the use of Classes.....	373
Visual Basic For the Research & Development LAB.....	379
Part IV : Visual Basic for the Engineering Lab	379
Introduction	379
Chapter 20 : The Computer.....	381
20.1 The PC : A Historical Overview	381
20.2 The PC : A Hardware Description.....	382
20.3 The PC's Input and Output Components.....	385
20.3.1 The Parallel port	385
20.3.2 The Serial port	385
20.3.3 The USB port.....	387
20.3.4 FireWire Channel	387
20.3.5 Local Area Network (LAN) and Wide Area Network (Internet)	387
20.3.6 Field buses (CAN VAN etc).....	388

20.3.7 The GPIB Bus	388
20.3.7 VXI / PXI / SCXI / Compact PCI etc	388
20.3.8 SCSI.....	389
20.4 The internal buses.....	391
20.4.1 ISA Bus	391
20.4.2 EISA Bus	392
20.4.3 MICROCHANNEL Bus.....	392
20.4.4 VESA Bus	393
20.4.5 PCI.....	393
20.4.6 AGP port.....	393
20.4.7 PCMCIA (PC Card)	394
20.4.8 I2C Bus.....	394
Chapter 21: Controlling Standard PC ports	395
21.1 Finding the IO ports	395
21.1.1 The BIOS system area	396
21.1.2 Using DEBUG to snoop around	398
21.1.3 The Dump command	399
21.2 Hardware Access	400
Chapter 22 The Printerport In Detail	403
22.1 Functional diagram.....	403
22.2 Register level description	404
22.3 Basic operations	407
22.4 Bit-Banging interfaces.....	408
22.4.1 Simple line control	408
22.4.2 Serial protocol emulation.....	409
22.5 Printerport Control Using ClassWork	412
22.6 Special printerport modes.....	413
22.6.1 Bi-directional Parallel Ports.....	413
22.6.2 The IEEE 1284 Standard	414
22.6.3 Extended Capabilities Port	418
22.6.4 Enhanced Parallel Port	420
Chapter 23 The Serial Port In Detail.....	421
23.1 System description.....	421
23.2 Port interface	422
23.3 Flow Control.....	422
23.3.1 Hardware Flow Control.....	423
23.3.2 Software Flow Control	424
23.3.3 Which Flow Control Method Should I Use?	424
23.4 The UART	425
23.4.1 Basics of Asynchronous Serial Communications	426

23.4.2 UARTs and the PC Serial Port	429
23.5 RS-232 and Other Serial Conventions	431
23.5.1 RS232	431
23.5.2 Current Loop and Other Serial Standards.....	435
23.5.3 RS422 / RS423	436
23.6 Cabling	437
23.5.1 Null modem cable.....	437
23.5.2 Full connection Null Modem Cable	438
23.6 Basic Serial Operations using MSCOMM	438
Chapter 24 : Plug-In boards.....	441
24.1 Description of the ISA bus	441
24.2 common interface chips.....	443
24.2.1 8255.....	443
24.2.2 8253/8254.....	445
24.3 Interfacing to ISA.....	446
24.3.1 Address & Data Lines	448
24.3.2 Utility Lines.....	448
24.3.3 Bus Cycle Definition Lines	449
24.3.4 Bus Control Lines.....	450
24.3.5 Interrupt Request and DMA Lines	451
24.3.6 Basic interface schematic using 8255 I/O controller	451
24.3.7 Basic interface schematic using classic logic	453
24.3.8 Selecting an address for our card.....	454
24.3.9 Accessing our board	456
Chapter 25: The GPIB bus.	459
25.1 The GPIB bus structure	459
25.2 GPIB signals.....	461
25.3 Controlling a device on GPIB	467
25.3.1 Initializing a GPIB system.....	468
25.3.2 Exchanging data	468
25.3.2 EOI assertion	469
25.4 IEEE488.2	469
25.4.1 Common Command Set.....	470
25.5 SCPI	471
Chapter 26: Vision	475
26.1 GPIBcore	477
26.1.1 GPIBcore features	478
26.1.2 Installing GPIBcore	479
26.2 GPIBcore programming guide	479
26.2.1 GPIB functions	479

26.2.2 GPIBinit.....	480
26.2.3 GPIBbye.....	480
26.2.4 GPIBopen.....	481
26.2.5 GPIBclose.....	481
26.2.6 GPIBtimeout.....	482
26.2.7 GPIBreset.....	482
26.2.8 GPIBdefer.....	483
26.2.9 GPIBsinglestep.....	483
26.2.10 GPIBtroff.....	484
26.2.11 GPIBtron.....	484
26.2.12 GPIBwrite.....	485
26.2.13 GPIBread.....	485
26.2.14 GPIBfind.....	486
26.2.15 Other GPIB functions.....	486
26.3 GPIBcore I/O functions.....	487
26.3.1 OUT.....	487
26.3.2 OUTW.....	488
26.3.3 INP.....	488
26.3.4 INPW.....	489
26.4 GPIBcore Miscellaneous support functions.....	489
26.4.1 setBIT.....	489
26.4.2 clearBIT.....	490
26.4.3 flipBIT.....	490
26.4.4 swapBIT.....	491
26.4.5 BITset.....	491
26.4.6 BITclear.....	492
26.4.7 swapNIBBLE.....	492
26.4.8 loNIBBLE.....	492
26.4.9 hiNIBBLE.....	493
26.4.10 SwapBYTE.....	494
26.4.11 loBYTE.....	494
26.4.12 hiBYTE.....	495
26.4.13 Delay.....	496
26.4.14 Microdelay.....	496
26.4.15 SStr\$.....	496
26.4.16 Bin\$.....	497
26.4.17 vVal.....	497
26.4.18 Logentry.....	498
26.5 Instrument and IO libraries.....	498
26.6 ClassWork.....	500
26.6.1 The ClassWork concept.....	500
26.6.2 The ClassWork solution.....	502
26.6.3 Programming using ClassWork.....	503

26.6.4 A Sample ClassWork program	504
26.6.5 Developing ClassWork Modules.....	505
26.6.6 Module Header	505
26.6.7 Internal ClassWork variables.	506
26.6.8 Initialize and Terminate events.....	507
26.6.9 Address assignment.....	508
26.6.10 AssignTo assignment.....	508
26.6.11 Global Lead-in code overview	508
26.7 General Rules for ClassWork module development	509
26.7.1 Properties.....	510
26.7.2 Methods (Sub)	510
26.7.3 Methods (Function).....	511
26.7.4 Special Cases	511
26.7.5 ClassWork implementation of the HP34401 driver.....	512
26.8 TestBench.....	514
Chapter 27 : Designing Test Programs.....	516
27.1 Clean code	516
27.1.1 Modular programming.....	516
27.1.2 Documenting code.....	517
27.1.3 Use indentation and camelwriting	517
27.2 Accessing instruments and hardware	518
27.2.1 Accessing instruments.....	518
27.2.1 Accessing hardware in the computer.....	518
27.3 Collecting data versus Analyzing	518
27.4 Creating log files	519
27.5 Anatomy of a well structured test-program	520
Chapter 28: Special Programming techniques.....	522
28.1 Stream Interpreting.....	522
28.1.1 Monolithic Program.....	522
28.1.2 Modular program.....	524
28.1.3 Creating the stream.....	526
28.2 Report generating on a printer.....	527
28.2.1 The Printer Object	527
28.2.2 The Printers Collection.....	528
28.2.3 NewPage.....	528
28.2.4 EndDoc.....	529
28.2.5 Example.....	529
Chapter 29: Building user interfaces.....	532
29.1 Build a splash screen and design a logo and icon.....	532
29.2 Constructing the Main form.	534

29.2.1 The Workplace of your program	534
29.2.2 Construct a Decent Menu	535
29.2.3 Tooltips.....	536
29.2.4 Toolbars.....	537
29.3 Organizing Objects and controls.	537
29.4 Configuration and tool forms	538
29.6 Help files	538
Chapter 30: Some more case studies.....	540
SPI stack on LPT.....	540
Data export to file.....	540
Building a U/I plotter using standard GPIB	540
Building a U/I Plotter using ClassWork.....	540
Building a U/I Plotter using TestBench.....	541
Case Study 11 : SPI stack on LPT.....	542
Case Study 12 : Data export to file.....	548
Case 13 : A U/I plotter using GPIBcore operations	552
Case 13 : A U/I plotter using ClassWork operations.....	556
Case 13 : A U/I plotter using TestBench operations	558
Appendixes	560
Appendix 1: Suggested Reading List	562
Appendix 2 : Datasheet for 8255 controller	565
Appendix 3 : Win95io users guide	566

Visual Basic

For Electronics Engineering Applications

Part I

The Basics of Visual Basic

Visual Basic

For Electronics Engineering Applications

Part I: The Basics of Visual Basic

Introduction.

Congratulations, you have decided to embark on an exciting journey: learning Visual Basic. So: welcome to the world of Visual Basic programming.

The goal of this book is to show you how you can write your own programs using Visual Basic. However, this is not ‘just another’ book on VB. The target group of this book is people building test setups in R&D environments.

The first part talks briefly over the windows environment and how it works. A basic understanding of this is somehow required to understand the programming techniques applied in Visual Basic. A basic explanation of how to write programs in VB is given.

The second part explains more advanced things such as graphics manipulation, file handling and more.

The Third part details on more advanced things like printing, multiform projects, ActiveX and beyond. Here you will learn to extend the already vast set of functions with those embedded inside Windows. A big deal is explained how to make your programs communicative. Inter-program communication like DDE is explained. Also off-system communication like serial communication and TCP/IP is explained. These are feature often used in creating test systems that can be managed via remote control.

The fourth part will show you how to unleash the power of Visual basic for application in a Lab environment. You will learn how to control instruments over GPIB / RS232, control circuitry over printer ports and even manage your own built plug-in boards.

The fourth part will also show you a how to apply the acquired knowledge to real-world systems. Things like controlling circuitry, emulating protocols and more will be covered. A number of examples combining all of the explained material will be given. A number of test setups will be given that show the capabilities of a programming system like Visual Basic.

Now, I know that most programmers get the chills when they hear about programming in BASIC. Some of you might remember the time of the first home computers such as the Commodore and Apple II, and visions of line numbers, goto's and spaghetti code start to doom. Well this is no longer the case. All languages evolve and so does basic. It has matured from a 'Beginners All Purposes Symbolic Instruction Code' to a full-blown programming language. The people who brought basic up (being Microsoft) have now come up with the latest incarnation of the language. It has been chosen as one of the standard programming language for the Windows environment.

More and more applications include a subset of the Visual Basic programming language. Applications such as Excel, Word, and Access all include Visual Basic for Application or VBA for short. More and more external vendors also include this engine in their products. Software houses like Oracle, AutoDesk (AutoCAD), Protel (PCB / Schematic / Simulation) Mathsoft (MathCAD) start

to embed the power of Visual Basic in their products. New operating systems like Windows 98 and Windows NT 4.0 have the VbScript language inside.

Visual Basic is in the first place a Visual programming language. In today's world of graphical user interfaces and windowing environments this is simply a must. More and more users demand a simple and easy to use interface to the software. Visual Basic enables the programmer to write just this kind of application. The programmer himself however needs not to be deprived of these things. Visual basic is really 'visual' both during development and runtime stage.

In the back of this manual you will find a CD-ROM. It holds an electronic copy of this manual. Besides that all source code of this manual can be found. The schematics and board layouts for various projects can be found in a separate directory.

Enjoy! And happy programming.

Vincent Himpe

Conventions used in this manual

Text	This is the typeset used for plain text
Bold	
<i>Bold Italic</i>	Used to introduce a new concept
<i>Italic</i>	Denotes keystrokes. When it is between arrows it means you have to hold it down while pressing the next letter. Example: <ALT> F means you have to hold the ALT down while pressing F.
Monotype Bold	Denotes named variables or objects. Used for source code
Monotype	Shows source code that can be run in Visual Basic

Chapter 1:

The Visual Basic Background

The use of today's graphical interface based operating systems puts tremendous strain on application programmers. They have to cope with all the stuff that is going on inside the operating system. This calls for a big amount of knowledge that is often hard to understand.

Visual Basic takes a different approach to programming for this kind of GUI (Graphical User Interface) systems. It gives you an interface builder where you no longer 'program' but rather draw the interface. Later on you simply attach your code to the interface and you are ready to run. When **Visual Basic** or **VB** for short, was released in 1987 it quickly gained universal acceptance as a windows programming language. It became the first in a series of new programming tools classified as **Rapid Application Development** systems, or **RAD** for short. It also includes a programming philosophy called **Object Oriented Programming** or **OOP** for short.

Since VB works tightly with the operating system there are some things about the operating system that need explanation first.

1.1 Windows

The windows environment is a graphical oriented environment. This GUI handles any action that occurs here, like a mouse click or a keystroke. Furthermore the operating system itself can invoke certain actions too (interrupts, timers, serial communication etc). Whenever anything happens an **event** is generated. These events can trigger other parts of the operating system or an application running on the system. The target of the event will then take appropriate action can reflect on what the user sees on the screen.

To inform the underlying code what exactly happened and ask what should be done an event is generated. You could compare this to an interrupt under a normal operating system.

The object of the action is called a **control**. It literally allows the user to control the environment he is working in. There are a number of standard controls available inside Windows. Any of these can be use inside a Visual Basic program. Furthermore you can create your own controls, or use third party controls.

The way a control looks and behaves is stored in its **properties**. These can be considered 'variables' that determine the look and feel of the control on the screen. Furthermore these properties can be changed from within the code.

Every control also has a number of **methods**. These are nothing else then stored subroutines or functions. If you want to resize or move a control on the screen you can use the MOVE method. The actual code behind this method is buried inside the control and is control specific. For example: moving a square object is not the same as moving a round object, but you can still do it using the move method. The embedded code in the object (the method) will take care that it gets performed, as it should be.

1.2 Object Oriented Programming

Windows is also an Object Oriented environment. This means that you will have to apply Object Oriented Programming methodology. Now what exactly is this weird stuff?

Well we have already explained most of it. The first criterion is *encapsulation*. This means that all the information about an object (properties) and the processes performed by the object (methods) are combined inside the definition of the object.

A real world object could be your car. You describe your car by its properties: color, number of doors etc. Every of these is a property of the *object* 'my car'. In Windows an object could be a button, a textbox, or a menu item. Any primitive part of windows is considered an object. These objects live a life of their own. They each have their properties, methods and associated events and they know how to do their stuff. Most of all they are accessible to the VB programmer.

As for the *methods*, these are the things that your car does in response of certain actions. For instance you generate the event 'Start Engine' when you turn the ignition key. If you execute the method 'Start Engine' the cars method handler takes over at that time and provides instruction to the underlying system to engage the starter motor, switch on the fuel flow, start ignition and then disengage the starter. You don't have to tell your engine how to start. The method 'Start' of the object 'Engine' knows how to deal with all the low-level stuff.

Now what is happening in this method is object related. A diesel engine has a different way of starting then petrol driven or electric car. But since this is encapsulated in the object, we as a user or programmer don't need to worry about that.

The second criterion for OOPS (Object Oriented Programming Style) is *Inheritance*.

It means that one object can be based on other objects. You could define your car as something that has four wheels, an engine, a chassis etc. Now I could define a car that has a retractable rooftop. This new object would inherit all of

the properties of the object 'car' and have one extra property. I could also include a new method to be able to control the retractable rooftop. I don't have to redefine everything for my new car model.

Actually when I have given the explanation about starting the car, I already gave a hint about this. The object 'car' holds another object called 'engine' inside. So actually inheritance goes a bit further than simply creating derivatives. You can create new objects containing other existing objects.

The last prerequisite is **Polymorphism**. This means that many objects can have the same method, and the appropriate action is taken for the object. For example in your programs you can print either on the screen or on the printer. Both of these objects have a Print method. The difference is that the underlying code will either access the video board or send data to your printer port. Compare this to your engine. No matter whether it is a petrol, diesel or electric engine, you simply need to call the method 'start engine'. The encapsulated code inside the object takes care of the rest.

1.3 What OOP does for you

The key elements of OOP with which you will be working with are re-usable components known as Controls. You will build programs with these controls (both standard, custom or even your own). These controls will have properties that will determine how they look and feel and they will have methods that will allow you to perform actions with them. The controls you will use shield you from many of the tedious tasks of programming.

1.4 Overview of the definitions:

Controls	Re-usable object that provides a visual interface between the program and the user. Examples: button, textbox, label etc.
Event	An action initiated either by the user or the operating system. Examples of events are keystrokes, mouse clicks, a timer overflowing, or incoming data from a Communication (RS232) port.

Methods	Program code that is embedded in the definition of an object. It defines how the object behaves in response to certain events.
Object	A basic element of a program, which contains properties and methods. And responds to events. Examples of objects are controls and forms.
Procedures	Pieces of code you write to accomplish a task. The procedures are usually written to respond to a certain event. Most of the time they handle the occurring event.
Properties	The characteristics of an object that determine the look and feel of the object. Examples: color, font, position, size etc.

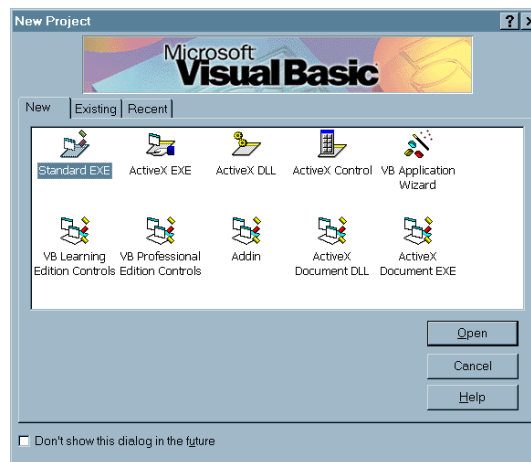
Chapter 2:

Exploring the Visual Basic environment

You might already be eager to start writing code, but let's first have a look at the environment.

2.1 Starting a Visual basic project

When you start up the programming language you are presented the *project dialog box*.

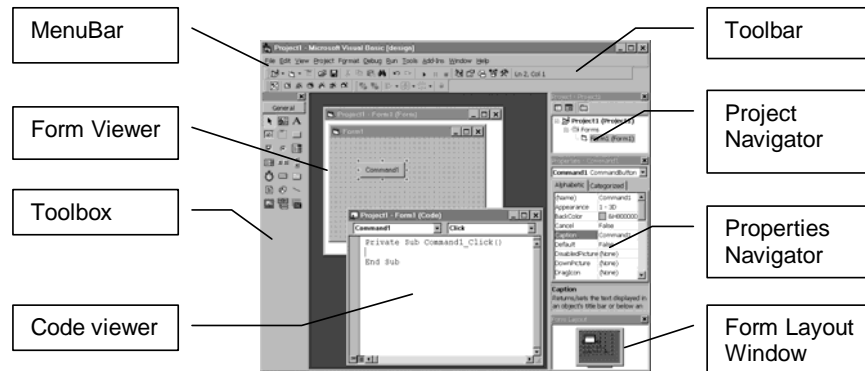


It lets you choose between different project styles. In the first parts I will cover only the standard EXE style. Some other styles will be discussed later on.

Standard Exe	This is the type of project you would use to create a standard Windows based program
ActiveX exe	This is a remote automation program that performs tasks as part of a program.
ActiveX DLL	This is a remote automation library. It cannot be run standalone, but can be called from other applications. An example of such thing could be a database search and retrieval tool.
ActiveX Control	This is a control you create. You can define its properties and create your own methods.
VB application wizard	This choice runs the VB wizard that builds a skeleton for your program based on the answers you give to certain questions.
ADD-In	This is a VB program running inside the VB environment that interacts with your work. It allows you to manage your work in an easier way.
ActiveX Document DLL	This creates an application that can run over the internet (you need Microsoft Internet explorer)
ActiveX Document EXE	This creates an application that can run over the internet (you need Microsoft Internet explorer)

After you have selected the program style you enter the VB desktop. This will be your workspace while designing, writing, debugging and compiling your program.

2.2 The programming environment



The Visual basic programming environment is the workplace where you will perform all of your program development. It neatly organizes a number of toolbars and information panels that assist you in creating user interfaces and writing code. It manages and allows you easy access to the entire project.

2.2.1 Using The Menu-bar

The menu-bar provides you with access to all functions that are available in the VB environment.

As with the menus in any Windows program you can access the functions using hotkeys. An underlined character in the item's name indicates a hotkey. Hold down the <ALT> key and press the underlined character.

2.2.2 Accessing functions with the Toolbar

The toolbar offers an alternative to the Menu-bar. It depicts some functions graphically. When you navigate the mouse over the toolbar you will see information appear in a small box just below the mouse cursor. This is called ToolTip. This functionality can be imbedded in your programs too!

2.2.3 The Object Browser (The Toolbox)

This part of the screen shows you which objects are available to use in your program. There are controls that allow you to edit text, show pictures, connect to a database or make selections. Upon creating a new project only the standard windows objects are available. By right clicking on the panel and selecting Customize you can insert more objects.

2.2.4 the project navigator

This part of the screen shows you the components that build your project. A program can have more than one window (or *form*), additional modules (a collection of your functions) or related files. This navigator shows you what your program is made of. You can use it to quickly jump from one part to another.

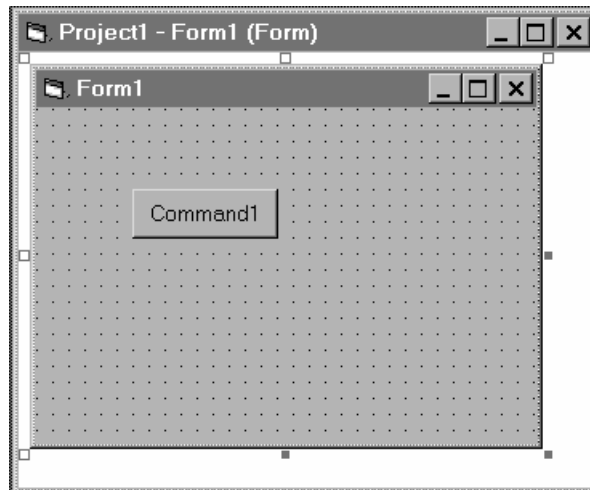
2.2.5 the properties navigator

This allows you to set the properties of the controls on your form. You simply select the control using the mouse (click it) and then edit the properties to the settings of your choice. You can call this window by pressing <F4>.

2.2.6 Form Layout Window

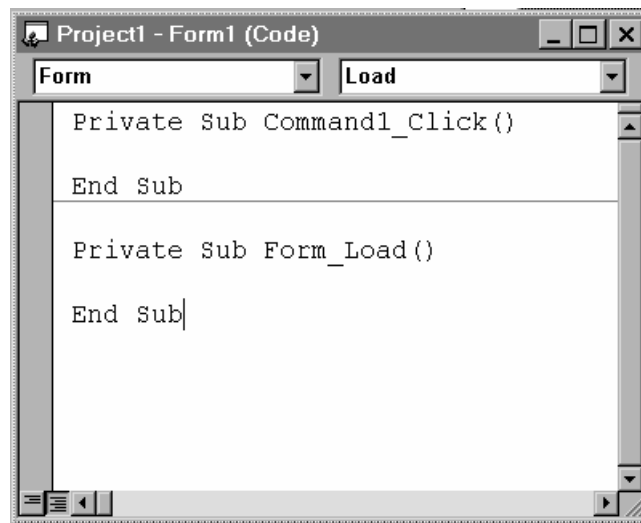
This window gives you an idea how your program will look at different screen resolutions.

2.2.7 Form Viewer



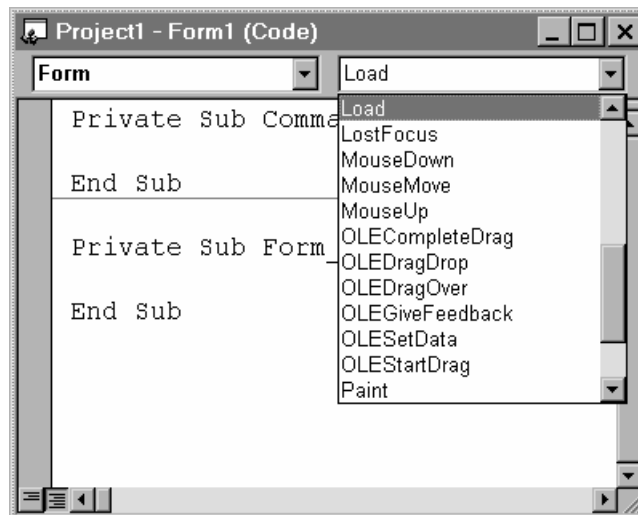
This is the real workplace for creating the user interface. It shows you the window that makes up the user interface. Here you place controls and other objects. To place a control simple click the desired control in the toolbox and then draw the boundary of it on your form. You can also double click an object in the object browser and then size it later.

2.2.8 Code Viewer



This is the second workplace in your program. Just like the form viewer shows the windows that make up your program, this viewer shows the underlying code.

To see the code that is attached to a control simply double-click the control in the form viewer. The code viewer will open up in the right place to show you the real code behind it. At the top you can find two selectors. The left list box allows you to select the object you want to modify. The right one allows you to select the event you want to attach code to. In the example below the `Form_Load` event will be modified.



2.2.9 The Help system

You can call in the help system whenever you want. Simply select the item you want help about and press F1. This works in every place of the environment. You can select a control and press F1 for help on the control. Or you can select a control, then select one of the properties, and then press F1 to get help about this particular property.

Another possibility is to highlight a word inside the code window and press F1: if it's a basic keyword you will get more information than you bargained for.

Furthermore the code editor has real-time Help. While you are typing code the system will show you your options for continuing your line of code. It will correct most of the syntax errors for you. Simple typing errors like missing quotes at the end of a string will automatically be corrected for you.

Chapter 3:

The Basic Objects and Controls

At the core of a Windows programs are always objects and Controls. A control is a type of object that is visible on the user interface.

3.1 The Form

The first object you have in almost every program is a *form*. This is the workspace for your entire program. Every control in your program needs to be placed on some form. Also this is the real thing that gets started. You can add forms at will by using the Project- Add form menu.

Your program has one and only one main form. This is the form that will be started first. You can select this in the project setup parameters (Project / project properties)

Just like any other object the form has **properties** and **events**. The properties can be set using the properties navigator or programmatically. The most important property of any object is it's name. This name is the unique designator used to refer to a particular object from within your code. All references to the objects properties and/or events will be made by this name. If you name your form Myform then you can refer to it as Myform. Suppose you wan to change the property *caption* from code then you simple type

```
MyForm.caption = "hello"
```

The click event will be known then as `MyForm_Click()`. All other events can be reached in a similar way.

Note:

When you double-click a control the code viewer will show you the default event for the control. You have to use the event browser on top of the code viewer to select the event you want to edit.

One of the events generated by a form is ***Load***. This is the event that is generated whenever your form is loaded the first time. In case of the main form this is at the start of a program. This procedure is the right place to load user settings etc. It's counterpart, the ***Form_Unload*** event is called upon termination of the program (in case of the main form) or when a form is unloaded.

3.2 The Controls

When your form has been given a name and you have set the properties to your taste then you can start adding controls to it. To do this, simply click one of the controls and then draw the outline of it on your form. Just like with the Form you can give your control a name. Let's call it ***MyButton***.

From now on you can access the control from code as ***MyButton***.

Note:

Visual basic is a case-insensitive language. This means that ***MyButton*** is the same as ***mybutton*** or ***MyBuTtOn***. Its case doesn't matter.

To this new object ***MyButton*** you could attach code for the ***Click*** event. Whenever you click the button that piece of code will be executed. You could for instance change its ***Caption*** property

```
Sub MyButton_Click ( )
```

```
MyButton.Caption = " You clicked me! "  
End Sub
```

The most basic program of them all could be created as follows:

Start a new project [FILE][New Project]. Select standard executable. Now put a label and a button on the form. Change the name of the button to ClickMe. Similar you name the label Hello. Change the caption property of the label to <empty>. Also change the caption of the button to 'Say Hi'. Now if you double click on the button you can attach the following code:

```
Sub ClickMe_Click ()  
    Hello.Caption = "Hello World"  
End Sub
```

By hitting [F5] you can run your first program. If you click on the button it will display the famous 'Hello World' in the label.



















And there you have it. With physically typing one line of code ('*Hello.Caption* = "Hello World", all the rest was generated by the Visual Basic environment) you have created your first object oriented, event driven, and GUI based windows program. You have written an event handler (*Clickme_click*), and controlled properties of objects.



And all of this with one line of code! How's that for Rapid Application development.

Of course you can do the above also with other programming languages. But it will cost you a great deal more effort to reach the same result.

3.3 The Standard controls inside Visual basic

Windows provides a set of universal controls that are always accessible. Any program uses these controls. Other controls are contained in separate control libraries.

Icon	Name	Function
	Picture Box	Allows you to display and edit graphics images
	Label	Displays text
	Textbox	Allows you to display and edit text, numbers and dates
	Frame	Provides a method for grouping other controls.
	Command Button	Provides a means to activate program functions
	Check Box	Displays or allows input of Boolean choices such as Yes -No, True - False or On - Off
	Option or Radio Button	Displays and allows a choice among multiple items
	Combo box	Allows the user to select an entry from a list or enter a new value
	List Box	Allows the user to select an entry
	Horizontal scrollbar	Allows the user to input numerical information
	Vertical scrollbar	Allows the user to input numerical information
	Timer	Provides a timed event. This can be used to fire actions on a timed basis.
	Drive List box	Displays and allows a user to choose from available disk drives in the computer
	Directory List Box	Displays and allows a user to choose from available directories on the computer
	File List Box	Displays and allows a user to choose from the available files in the computer
	Shape	Displays geometric shapes on the form
	Line	Displays line on the form
	Image	Displays graphic images











	Data Control	Provides a link to database files
	OLE Control	Provides a link to OLE servers (ActiveX)

The above sets are the controls made available by windows. Apart from these controls there are lots of controls that you can plug into the system. Some of these come shipped standard with Windows, some come together with the Visual basic language. One set of interesting controls is the **CommonDialog** collection.

This is a control that allows lots of common tasks to be performed by the system. Things like printer selection, color selection, file save and load forms are inside this control. You don't have to write code each time you want to give the user a file dialog box. You simply call in this control and you're done.

Another interesting set is the Windows Common controls. These include the new GUI style controls. These controls work only in the new GUI on WIN32 (Win95 and NT4.0 and up).

3.4 Common Controls

Icon	Name	Function
	UP-Down Control	This control 'glues' itself to another control. It allows you to changes numerical values up and down
	Animation	This allows you to specify an animation sequence. Like the flying paper when file copy is in progress.
	Slider	This allows the user to specify a numerical value.
	Listview	This looks like a file list-box
	Treelist	This displays and indexed list like in the windows explorer
	Imagelist	This allows you to specify a list of images. Can be used to make animations
	Progressbar	This can be used as a progress indicator
	Statusbar	A common control to make snazzy status bars
	Toolbar	Creating toolbars is a snap. Looks like the menu editor.
	Tabstrip	Allows you to make multi-panel forms

To insert these controls, and others, you can click on the control toolbox with the right mouse button. Then select Customize Toolbar. Here you will see all available controls on your system. Simple select the ones you find interesting and you're off. The library that holds the Common Controls is called Comdlg.dll. Like all components of windows it is subject to revisions. You have to keep this in mind when you distribute your program to an end-use. Make sure he has the same or a later version than yours. You can do this by creating a so-called distribution-pack. (This will be explained later).

3.5 Common Dialog Control

Icon	Name	Function
	CommDlg control	Allows you to perform serial communication

This control gives you a set of commonly used forms. It takes away the problem of having to reinvent the wheel all the time. Almost every program needs a file selector anyhow. This control accesses the standard windows methods of selecting a filer, printer, color etc.

3.6 Comm. Control

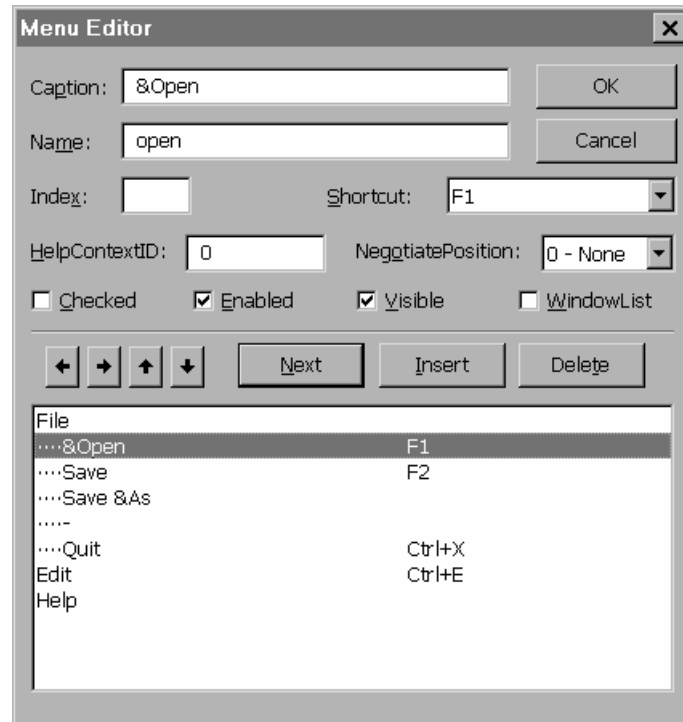
This control allows you to perform serial communications. This control is simply put on a form and given a name. Using the properties you can select things like baud rate and settings of the serial port. The control is invisible during runtime. You can have up to 16 of these Comm. controls on the form. The reason is the hard limit of the controls capabilities. It cannot handle more then 16 ports at a time.

Icon	Name	Function
	Comm. control	Allows you to perform serial communication

3.7 Menu's

The menu on a form is a control just like any other, except that it does not appear on the Control browser. The reason is that actually the menu is part of the form. You cannot have more than one menu on a form.

To create or edit a menu you have to start the menu editor. Tools - Menu editor or click on the Menu editor icon of the toolbar.



You start by typing the caption to appear on the menu bar. In the above case **&Open** is typed. The ampersand (&) means that the O should have a line under it. This will be the hotkey for the Open command. It also has the <F1> key assigned to it (Shortcut).

The next thing you have to do is give the item a name. In the shown example it is simply 'open'.

Any references to this menu item will be made by this name. So if I would like to have a checkmark appear before it I would execute `'open.checked = true'`.

To make multilevel menus you simply use the arrow buttons on the menu editor. You can move items using up down or shift them to a different level using left right.

Note:

To make a divider-line appear you have to specify a minus sign as the name for the entry. This kind of item cannot be activated and loses all other properties.

When you are done editing you simply click Close. The menu editor then compiles your menu and puts all controls in place on your form. Since every entry in your menu is actually a small control you can change certain properties of these entries. You can for instance change the checked or enabled property. In case of an error (you forgot to give a menu item a name, or the syntax is incorrect), you will not be able to close this window. You must first correct the errors before you can continue.

With checked a small checkmark could be made to appear before the menu item. You can use this to show the user which items he has selected. With the enabled property you can disable or enable certain menu options.

3.8 Properties

Every control has properties that accurately define how it looks and feels. It would lead us too far to explain all of them. The detailed help system inside visual basic is far more useful to explore them. However some basic properties need to be known. These properties exist and about 90 percent of all available controls.

3.8.1 Name

This is the single most important property a control can have. This property defines the handle you will use to access the object. You should set this property before you start writing any code for the object. The code editor will use this properties setting to generate the procedures for you.

Note:

If you change this property when code has been written for a control you will have to update all of you code relating to it manually!

3.8.2 Top, Left, Height, Width

These properties define the location of the control in relation to the form it is placed on. When you move the control in the form editor you will see that they are changed. You can also adjust them manually to create neatly aligned controls, or you can change them from within your program at runtime to make objects move on the screen.

3.8.3 Backcolor, ForeColor, Textcolor

With these properties you can modify how the control looks on the screen. You can set specific colors using the color selector or you can use the DOS based colors. Under Dos you had 16 colors available. You can still use these numbers to specify a color. To convert these numbers to the Windows coloring scheme there is a function called QBcolor.

When designing forms it is a very bad idea to freeze colors. The windows design guide describes that you should use the system colors instead of forcing your own. This can be done very easily. If you look at the color selector on the properties bar you will see that there are 2 panels. One is holding a color chart. The other is listing variables that refer to the system colors. You should use these variables instead. Whenever your program starts it will retrieve the system colors and use these for your program.

3.8.4 Caption and Text

The settings of these variables control what is displayed on the object. In general Caption is used for a static text display. This means the text is not changing a lot and the user needs not to edit the text. Text is a dynamic control. This means the user can change it, edit it.... whatever he could do with a text.

Most controls will thus have a Caption. Only Textbox, and Combobox have a Text property (there are others but they are not part of the standard windows control set)

3.8.5 Enabled and Visible

These properties define the active state of the control. The enabled property defines if the control will respond to events. If set to 'false' the object is detached from the message stream and it will do absolutely nothing. If you set it invisible however it just disappears from the screen. It does not get detached from the message queue.

3.8.6 Index

This specifies the objects place in a control array. If empty the control does not belong to an array. Otherwise it determines the position in the array.

3.8.7 Tabindex

This is also an index in an array but it determines the order in which controls are accessed .If you press the tab button you can switch from control to control. This is useful for users that don't have a mouse. The list is scanned in ascending order.

3.8.9 *ToolTipText*

This is a handy property that allows you to enter a few words of explanation about a control. During runtime whenever the user moves the mouse over a control and leaves it there for a few seconds this text will be displayed just below the mouse cursor. It gives somewhat instant help in your programs.

Chapter 4:

Events and Methods

As explained before, events are the driving power behind the OOP / GUI programming style. Whenever something happens, for instance the user clicks with the mouse, hit's a key or a character comes in over the serial port, an event is generated.

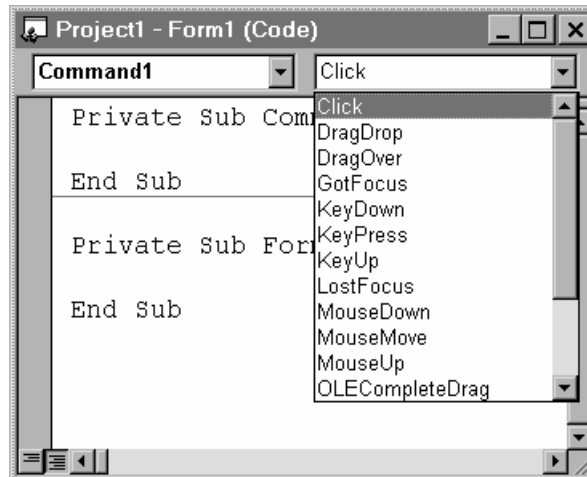
Now the above examples are only a small part of the possible events. Every tiny bit of action generates events. Even the mere fact that you move the mouse generates a stream of events.

Hitting a key alone generates 3 events. KeyDown, KeyUp and KeyPress

A simple thing like clicking the mouse can generate 4 events MouseUp, MouseDown Click and DoubleClick. While clicking you could have moved the mouse thus a stream of MouseMove events could have occurred.

4.1 Tapping into Events

You can tap into all of these events by using the properties browser in the code editor.



Every control has its own set of events that it can generate. The most useful events however are very limited. It's not until you start to do very complicated work that you will need the other events. Actually when you edit code using the method of double-clicking an object, the code editor will show only the most used event. The others have to be accessed using the event browser of the Code editor (see picture above).

4.1.1 Click (Most controls)

This is probably the most used event. Whenever the user clicks an object this event is fired. This is the place you will attach the real code of your program.

4.1.2 DblClick (Most controls)

The same as the Click event but it gets fired only when the user double-clicks. Important: double-clicking does not fire the Click event

4.1.3 KeyPress (Most controls)

Whenever you hit a key the object that has the focus fires this event. It is useful in combination with textboxes to make masking. Suppose you want the user to enter a number only. You could attach the following code to the KeyPress event of the textbox

```
Private Sub Text1_KeyPress (KeyAscii As Integer)
    Select Case Chr$(KeyAscii)
        Case "0" To "9"
            Text1.Text = _
Text1.Text + Chr$(KeyAscii)
        Case Else
            End Select
    End Sub
```

The KeyPress returns the ASCII value of the key that was hit. So if we detect that it lies between 0 and 9 (thus being a valid number) we allow it to go into the textbox.

Note:

Whenever you attach code to any of the Keypress events, the object will not handle the keyboard input for you. You are then responsible for taking appropriate action.

4.1.4 MouseMove (Most controls)

Whenever the user moves the mouse, the object under the mouse will fire the MouseMove event. You can use this event to retrieve the coordinates of the mouse. This can be useful if you want to make a small drawing program.

4.1.5 Activate (Form)

Only a Form generates these events. Whenever the user moves the focus to a form, it will fire this event. Suppose you have a program with 4 forms. You can only have one form active at a time. The form then becomes the new active form will fire this event. You can use this to update status bars, or to create context sensitive help.

4.1.6 Deactivate (Form)

This command is similar to the above. Whenever a form gets the focus, another one must lose it. The form that loses focus is generating the Deactivate event.

4.1.7 Load (Form)

This is probably the most useful form related event. Together with the unload event. If you are using local variable or need to load configuration or INI files this is the place to do it.

Whenever a form gets loaded the first time it fires this event. Since in a typical application a form only gets loaded once (during the start of the program) you can use this event to attach your startup code.

4.1.8 Unload (Form)

Similar to the **Load** event, the unload event gets fired when a form is destroyed (unloaded). Since this only happens during program termination you can use this event to store user preferences, or form size and position into an INI file, the registry or whatever.

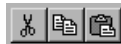
This is the place where you put your program's bailout code.

4.1.9 Change (Textbox)

The change event gets fired whenever the contents of the textbox change. This can be used to make terminal like programs.

Now there is one more thing about events you need to know. You can generate them also from within code! Since an event handler is nothing else then a subroutine you can call them from any part of your code belonging to the same form. You cannot call them from other forms, except if you declare them public. But declaring an event handler public should only be done in modules and it still is considered bad behavior. It can lead to very strange program behavior.

Suppose the following: You have a couple of buttons on a toolbar that allow you to Edit cut copy and paste text.



On your edit menu you have the same Functions, Cut Copy and Paste. Now the question is: are you going to write the cut copy and Paste code twice? Not in your life. The events generated by the toolbar buttons will simply generate the events like you would have clicked on the menu bar.

```
Sub Cut_Click ()  
    ' Cut text from textbox code ...  
End sub  
  
Sub ToolbarCut_click ()  
    Call Cut_click ()  
End sub
```

The First subroutine is the event handler for the Menu bar. The second Event handler is attached to the button on the toolbar. Whenever you click the button on the toolbar, it passes control to the event handler on the Menu bar. Done! No additional code no nothing.

The only pitfall in this is that you have to make sure your event handlers are not calling each other. This will create a recursive event and blow the stack sky-high. Result: The darned blue screen of death: A General Protection fault.

4.2 Methods

So far we have talked about objects their properties and the events they can generate. Now, an object has one last thing that is called Methods.

The easiest way to understand them is to think of them as built in procedures. Lets take the Move method. Almost any object supports it. Suppose you want to move a button. You could of course change the properties **Top** and **Left**, but that takes too much work (you have to calculate the absolute movement from the current and the new coordinate). Well you can use the **Move** method.

Object.move (x, y)

This will move the object to the new coordinates. Now the underlying code for a button is not the same for let's say a textbox. This is the strength of Methods. They have the same name, work on nearly all objects but are completely different internally.

A typical other method is the **Print** command. In normal basic **Print** is a keyword; well in VB it's a method. You might ask why? Well simple. You can pick an object and print to it. Like you can print to the form you can also print to a button, or a graphics box or even to the Printer object!

The internal workings are completely different yet the action has the same result. Except that, when printing to the form it ends up on the screen, and when printing to a printer it ends up on paper.

Now that we have discussed the nuts and bolts of the Visual basic programming language and the environment it works with, it's about time we start building a program. The next few chapters will show you how to build a user interface, how to attach code and how to get it up and running.

Later in this course more examples will be given to detail a bit on the most used features inside the programming language. But first something needs to be told

about the language itself.

Chapter 5:

The Basic language itself.

Before we can start writing code, we should know a little about the driving force behind all of this: the basic language. Originally conceived by Kemeny and Kurz in 1968 this language has often been regarded to as 'not useful for anyone beyond first grade'.

While true that Basic tended to lead to sloppy code, and the first interpreters were terribly slow, today's compilers can unleash the power of the machine. For technical environments basic is still the Number One language. Lots of research departments from outstanding universities solely depend on it to build the 'quick and dirty' problem solver they needed 3 weeks ago.

Since the compiler has to generate code which deals with whatever the programmer is cooking up, the resulting code will always be slower than a very low level language like 'C' and Assembler, or more structured language like Pascal.

However, the Visual basic compiler uses the same underlying technology as the Visual C++ compiler from Microsoft. This means that the same code generation process is used for both. This results in code that is heavily optimized and nearly as fast as the code generated by the Visual C compiler.

But the true power of basic lies in the possibility to develop a program in virtually no time. A basic program will be running and starting to do something

while in other languages you are still deciding what variable type to use, or checking out which library you need now.

A programming language is very similar to a human language. Before you can learn it you need to know the vocabulary (instruction set) and grammar (syntax). This will allow you to construct sentences (lines of code). The content of the text written in a language (the algorithm) is a different matter however. This is only acquired by practicing. Just like you can express something in different ways, you can solve problems programmatically in different ways.

Computer languages differ a bit from the human languages. They are much more organized. They need for instance ways to describe the data. So let's talk about that first.

5.1 Variables

A computer language uses variables to store data. They are a symbolic name used by the programmer to refer to data stored somewhere in memory. The compiler will allocate the necessary storage space and map it into the computers memory. Data comes in all sorts of colors and flavors. You can have numbers, letters, strings etc. ... So it is logical that there are different ways to store data. In most programming languages you have to explain to the compiler what kind of data to store. In Visual Basic you DON'T have to!

There is only one variable type. No integer, float, double quad. Just storage space. You need space? You got it. What do you want to store? Doesn't matter. How big is it? Of no importance.

Visual basic introduces the concept of a *Variant*.

A variant is a universal storage space. It is virtually unlimited in size '(16.777.216 bytes max) and can hold everything ranging from strings, numbers, pictures to even objects.

You even don't need to define it. Just use any name you want. In Basic it is the compiler's job to figure out how to store your data.

However over time the users of basic found out that if you are not careful enough you will end up with messy code that can be very buggy. Therefore Visual basic allows you to force yourself to program clean. You can use the Option Explicit command in the top of a module. Then you need to declare the variables and *typecast* them. This gives also some speed improvement since now the compiler can generate much more optimized code. Also you will have more storage space. When a variable is typecast then only the necessary amount of memory is allocated. A variant always uses at least 16 bytes of memory, even if it is empty.

5.1.1 Available Types in Visual Basic and how to declare them

Type	Character	Memory Requirement	Range of Values	Stores
Integer	%	2 byte	-32.768 to 32.767	Whole numbers
Long	&	4 byte	Approx 2 billion	Whole numbers
Single	!	4 byte	-1e-45 to 3e38	Decimal numbers
Double		8 byte	-5e-324 to 1.8e308	Decimal numbers
Currency		8 byte	-9e14 to +9e14	Numbers with up to 15 digits left and 4 digits right of the decimal
String	\$	1 byte + 1 byte per character	Up to 65000 character for fixed length and up to 2 billion for dynamics	Text information
Byte	None	1 byte	0 to 255	Whole numbers
Boolean	None	2 bytes	True or False	Logical values
Date	None	8 bytes	1/1/100 to 12/31/9999	Date and time information
Object	None	4 bytes	Not applicable	Pictures and OLE objects
Variant	None	16 bytes + 1 byte per character	Not applicable	Any of the above.

Typecasting a variable can be done in 2 ways. Either you declare a variable explicit or use the typecasting character (implicit).

```
a$ = "test" ' implicit declaration  
  
Dim a as string  
a = "test" ' explicit declaration
```

The net result is the same. By ending the name of a variable by the typecast character you force its type. In the second case you declare them using the **As** keyword. They both have their advantages and disadvantages. It's up to you which one you use.

If you decide to go for the explicit method then you have to use the **As** keyword

```
Dim account As Currency  
Dim a As Byte  
Dim power As Boolean
```

There is a little difference between implicit and explicit declaration. When using implicit declaration you add the type declaration character at the end of the variable name. This means that from now on you must reference the variable including the type character. Explicit declaration avoids this ,but then you can't immediately see what is the type of the variable. You have to look it up in the declaration clause of the variable. It's up to you what you want to use.

Note :

A variable name must start with a letter , the name cannot contain a period and can be no longer then 255 characters.

5.2 Arrays

Suppose you need to have more then one variable with the same name. For instance a table or an array. It would make programming easier if we could use

an index to refer to a set of variables. That's exactly what arrays are intended for. Arrays can be created for any kind of variable. You can even create arrays for objects (more on this later). Contrary to regular variable you need to declare them. Regular variables you can use on the fly. However you do not need to typecast them .

Declaring an array is done using the DIM keyword.

5.2.1 DIM

```
Dim myarray(5)  
Dim twodimensions(5,6)  
Dim ThreeD(5,10,100) as integer
```

The above examples are declarations for a number of arrays. The first two are not typecast (no explicit type for them is declared) . The last one is typecast as integer. When using big arrays it is useful to typecast them. Since arrays typically contains lots of variables (the multiplication of all dimensions : example a (5,10,10) contains $5*10*10 = 500$ elements) it is wishful to typecast them . Doing so will preserve a lot of memory.

If they are not typecast then they are assumed as Variant (which takes 16 byte per item if empty). You can declare arrays of up to 255 dimensions. Don't try to visualize how this would look , you can't. As a matter of fact nobody ever does this (except maybe some mathematicians or physicists).

An array is by default Zero-Based. This means if you declare an array of 5 elements you have access to elements 0 to 4. Suppose you want to store the years between 1900 and 2000. To conserve space you could declare an array that stores only the last two digits. (DON'T do this ! The Y2K bug will get you for this !). After all if you declare an array of 2000 elements you will waste the first 1900 of them. Well visual basic allows you to change this base.

```
Dim years (1900 to 2000)
```

Will declare an array with 100 elements. The first element will have an index of 1900 and the last will be 2000.

```
Dim MyMatrix(1 To 5, 4 To 9, 3 To 5) As Double
```

Will create an array with specified bounds.

Arrays created in this manner are called Static arrays. You lock the amount of memory at coding time. Equally you can create dynamic arrays

```
Dim myArray( )
```

Later in the code you can re-dimension your array with the REDIM command

5.2.2 ReDim

The ReDim statement is used to size or resize a dynamic array that has already been formally declared using a Private, Public, or Dim statement with empty parentheses (without dimension subscripts).

You can use the ReDim statement repeatedly to change the number of elements and dimensions in an array. However, you can't declare an array of one data type and later use ReDim to change the array to another data type, unless the array is contained in a Variant. If the array is contained in a Variant, the type of the elements can be changed using an As type clause, unless you're using the Preserve keyword, in which case, no changes of data type are permitted.

If you use the Preserve keyword, you can resize only the last array dimension and you can't change the number of dimensions at all. For example, if your array has only one dimension, you can resize that dimension because it is the last and only dimension. However, if your array has two or more dimensions, you can change the size of only the last dimension and still preserve the contents of the array. The following example shows how you can increase the size of the last

dimension of a dynamic array without erasing any existing data contained in the array.

```
ReDim X(10, 10, 10)
. . .
ReDim Preserve X(10, 10, 15)
```

Similarly, when you use Preserve, you can change the size of the array only by changing the upper bound; changing the lower bound causes an error.

If you make an array smaller than it was, data in the eliminated elements will be lost. If you pass an array to a procedure by reference, you can't re-dimension the array within the procedure.

When variables are initialized, a numeric variable is initialized to 0, a variable-length string is initialized to a zero-length string (""), and a fixed-length string is filled with zeros. Variant variables are initialized to 'Empty'. Each element of a user-defined type variable is initialized as if it were a separate variable. A variable that refers to an object must be assigned an existing object using the Set statement before it can be used. Until it is assigned an object, the declared object variable has the special value 'Nothing', which indicates that it doesn't refer to any particular instance of an object.

When you are writing procedures or functions that have to accept data in arrays it is always wise to query the array for its bounds.

There are commands Ubound and Lbound that allow you to do just that

5.2.3 Ubound

Return a Long containing the largest available subscript for the indicated dimension of an array.

Ubound (array name [, dimension])

Array name	Required. Name of the array variable; follows standard
------------	--

	variable naming conventions.
Dimension	Optional; Variant (Long). Whole number indicating which dimension's upper bound is returned. Use 1 for the first dimension, 2 for the second, and so on. If dimension is omitted, 1 is assumed.

The Ubound function is used with the Lbound function to determine the size of an array. Use the Lbound function to find the lower limit of an array dimension.

Ubound returns the following values for an array with these dimensions:

```
Dim A(1 To 100, 0 To 3, -3 To 4)
```

Statement	Return Value
Ubound(A, 1)	100
Ubound(A, 2)	3
Ubound(A, 3)	4

5.2.4 Lbound

Returns a Long containing the smallest available subscript for the indicated dimension of an array.

Lbound (array name [, dimension])

Array name	Required. Name of the array variable; follows standard variable naming conventions.
Dimension	Optional; Variant (Long). Whole number indicating which dimension's lower bound is returned. Use 1 for the first dimension, 2 for the second, and so on. If dimension is omitted, 1 is assumed.

The Lbound function is used with the Ubound function to determine the size of an array. Use the Ubound function to find the upper limit of an array dimension.

Lbound returns the values in the following table for an array with the following dimensions:

Dim A(1 To 100, 0 To 3, -3 To 4)

Statement	Return Value
Lbound(A, 1)	1
Lbound(A, 2)	0
Lbound(A, 3)	-3

The default lower bound for any dimension is either 0 or 1, depending on the setting of the Option Base statement. The base of an array created with the Array function is zero; it is unaffected by Option Base.

Arrays for which dimensions are set using the To clause in a Dim, Private, Public, ReDim, or Static statement can have any integer value as a lower bound.

5.2.5 Array

Array(argument list)

The required argument list is a comma-delimited list of values that are assigned to the elements of the array contained within the Variant. If no arguments are specified, an array of zero length is created.

The notation used to refer to an element of an array consists of the variable name followed by parentheses containing an index number indicating the desired element. In the following example, the first statement creates a variable named A as a Variant. The second statement assigns an array to variable A. The

last statement assigns the value contained in the second array element to another variable.

```
Dim A As Variant  
A = Array( 10 , 20 , 30 )  
B = A( 2 )
```

The lower bound of an array created using the Array function is determined by the lower bound specified with the Option Base statement, unless Array is qualified with the name of the type library (for example VBA.Array). If qualified with the type-library name, Array is unaffected by Option Base.

Note

A Variant that is not declared as an array can still contain an array. A Variant variable can contain an array of any type, except fixed-length strings and user-defined types. Although a Variant containing an array is conceptually different from an array whose elements are of type Variant, the array elements are accessed in the same way.

Example :

```
Dim MyWeek, MyDay
MyWeek = Array("Mon", "Tue", "Wed", "Thu",
               "Fri", _
               "Sat", "Sun")
' Assume lower bound set to 1 (using Option Base
')

MyDay = MyWeek(2) ' MyDay contains "Tue".
MyDay = MyWeek(4) ' MyDay contains "Thu".
```

This example uses the Array function to return a Variant containing an array.

5.3 Types

An array can be regarded to as a simple database. If you require storing data in a database like way you can create your own data types.

Type is used at module level to define a user-defined data type containing one or more elements.

```
[Private / Public] Type varname
    elementname [( [subscripts] )] As type
    [elementname [( [subscripts] )] As type]
    . . .
End Type
```

Public	Optional. Used to declare user-defined types that are available to all procedures in all modules in all projects.
Private	Optional. Used to declare user-defined types that are available only within the module where the declaration is made.
Varname	Required. Name of the user-defined type; follows standard variable naming conventions.
Elementname	Required. Name of an element of the user-defined type. Element names also follow standard variable naming conventions, except that keywords can be used.
Subscripts	Optional. Dimensions of an array element. Use only parentheses when declaring an array whose size can change. The subscripts argument uses the following syntax:
[lower To] upper [, [lower To] upper] . . .	When not explicitly stated in 'lower', the 'Option Base' statement controls the lower bound of an array. The lower bound is zero if no Option Base statement is present.

Type	Data type of the element; may be Byte, Boolean, Integer, Long, Currency, Single, Double, Decimal (not currently supported), Date, String (for variable-length strings), String * length (for fixed-length strings), Object, Variant, another user-defined type, or an object type.
------	--

The Type statement can be used only at module level. Once you have declared a user-defined type using the Type statement, you can declare a variable of that type anywhere within the scope of the declaration. Use Dim, Private, Public, ReDim, or Static to declare a variable of a user-defined type.

In standard modules, user-defined types are public by default. This visibility can be changed using the Private keyword. In class modules, however, user-defined types can only be private and the visibility can't be changed using the Public keyword.

Line numbers and line labels aren't allowed in Type...End Type blocks.

User-defined types are often used with data records, which frequently consist of a number of related elements of different data types.

The following example shows the use of fixed-size arrays in a user-defined type:

```
Type StateData
    CityCode (1 To 100) As Integer ' Declare a
    static array.
    County As String * 30
End Type

Dim Washington(1 To 100) As StateData
```

In the preceding example, StateData includes the CityCode static array, and the record Washington has the same structure as StateData.

When you declare a fixed-size array within a user-defined type, its dimensions must be declared with numeric literals or constants rather than variables. Also note that the setting of the Option Base statement determines the lower bound for arrays within user-defined types.

```
Type EmployeeRecord      ' Create user-defined
type.
    ID As Integer        ' Define elements of
data type.
    Name As String * 20
    Address As String * 30
    Phone As Long
    HireDate As Date
End Type

Sub CreateRecord()
    Dim MyRecord As EmployeeRecord ' Declare
variable.
    MyRecord.ID = 12003 ' Assign a value to an
element.
End Sub
```

This example uses the Type statement to define a user-defined data type. The Type statement is used at the module level only.

Now I have to be honest with you. You can perfectly do this but there are far more powerful ways to create this. Visual basic has direct access to ODBC objects (Open Database Control). This allows you to create and manipulate data far more efficiently than by coding everything yourself. For simple things it might be useful and faster not to use this heavy database engine. It's up to you to decide on this.

5.4 Scope of Variables



Vartype.vbp

When you define a variable (read declare it using DIM or simply start using a new variable name) it only exists locally.


```
Sub Button1_click()  
    Dim a as Integer  
End Sub  
  
Sub Button2_click()  
    Dim a as Long  
End Sub
```

In the above examples both variables A are independent variables. They have nothing to do with each other. When you exit a subroutine all variables are destroyed. When you allocate them the first time then they are created and reset. (The contents are set to zero for numbers or nothing for strings). Sometimes you might want a variable to exist outside of your procedures.

This can be done . There are 4 ways to preserve a variable : Static , Private , Public and Global. Public and Global are the same. The global keyword comes from old style basic and is maintained for compatibility reasons.

5.4.1 Public / Global

This kind of variable can be accessed anywhere in your program. It can be read and written from any form , module , procedure and function.

5.4.2 Private

The Private variable is a variable that only exists in the current portion of code. Only routines belonging to the same form or module as the one where the variable has been declared can access it.

5.4.3 Static

This is a variable that can only exist inside the function or procedure where it has been declared . Just like a normal variable . But it will not be destroyed upon exit of the procedure. Nobody outside the procedure can access it .

```
Function countup()  
    Static a As Integer  
    countup = a  
    a = a + 1  
End Function
```

If you would declare 'a' as a normal type then the function would always return 0 . Every time you call the function a storage space for 'a' is allocated , set to zero and upon exit the storage space is freed. By declaring the variable as Static it only gets created the first time you call the function. The next time you call the function then the variable A and its contents still exist and their contents have been unaltered.

5.5 Module level scope

Besides the static , public etc type variables can be bound to the code module they live in. A Module is a physical file that contains code. Forms , BAS files etc are all modules. A program is created from one or more of these.

Consider the following piece of code

```
Dim x as integer  
  
Sub form1_load()  
    Dim y as integer  
    Y = 2  
    X = 5  
    Call addup  
End sub  
Sub addup()  
    Debug.print x+y
```

End sub

**VarScope.vbp**

When this program is run it will always return 5. Why ? Well simple : the variable *X* is defined at module level. This means it is accessible from anywhere inside the same module. As long as you do not re-declare it you can reach it. The variable *Y* is created inside the `form1_load`. This means it is destroyed when exiting this part of the code. Referencing an in-existing variable will return 0. So the net result of $X+Y$ is $5 + 0 = 5$.

5.6 Subroutines and Functions

While programming you might have developed routines that are interesting to keep and that can be used in different portions of your programs. There is a neat way to store them and use them from any location.

5.6.1 Subroutines or Procedures

The simplest form is a procedure. It is a portion of code which performs a certain action based on the inputs you feed it. It does not generate any resulting output.

```
Sub DrawLine (x1,y1,x2,y2)  
End Sub
```

The procedure can work with the passed information and does something without returning an answer.

5.6.2 Functions



FunCall.vbp

A function is the same as a procedure but it returns a value that can be result of the functions operation.

```
Function Add (a,b)  
    Add = a + b  
End Function
```

The programming environment will automatically provide syntax help for any procedures or functions in your program. It shows you while typing what a certain procedure expects from you now. While declaring a procedure or function you can typecast the variables that need to be passed to them , or are returned from them..

```
Sub Test ( a As Integer , b as Currency )  
End Sub
```

```
Function addup ( a as integer , b as integer) as integer  
    Addup = a + b  
    Debug.print a + b  
End function
```

You can call a function without having to collect the result. You simply call it like you would call a subroutine

```
Addup 2,1
```

In the above example would call the function but not return a value. It would still get printed to the debug console. If I wanted the result of the function I would call it like a function

```
X = addup (5,5)
```

5.7 Scope of procedures



ProScope.vbp

Just like variables, subroutines or functions also obey to a certain scope. You can force this scope using the Public or Private modifiers. They cannot be made Static.

By default you can call any routine as long as it lives inside the same module. It is of no importance whether they are public or private. This scenario changes when you go to multi-module programs (multiple forms and or included files). The privately declared procedures are invisible to other modules. The public ones are visible. This means that you can have two modules called 'addup' in two different modules without any problem.

5.8 Constants

Storing often-used numbers in a variable makes programming easier. Furthermore it makes modifying and reading the program a lot easier afterwards. But it has one drawback : it eats memory. Therefore the concept of a constant has been defined. A constant is a placeholder for information. The only

difference is that it has no memory allocation during runtime. During compilation VB will replace all instances of the constant name with its contents.

Just like variables you can have Public (Global) or Private constants. Constants are declared with the CONST keyword

```
const version$ = "Version 1.0"  
const pi = 3.1415927
```

You can store anything in a constant. Since it has no real substance you do not have to specify the type. However , you have to take care when using them .

With the above constants this would yield an error :

```
result = pi * version  ' multiply a number with  
a string ??  
version = 12           ' error : you cannot change a  
constant
```

A special kind of constant is a so-called Enumerated Constant. While it can be used as a regular constant, it has special applications in Classes and User created objects.

Declaring it using the ENUM keyword creates an enumerated constant.

```
Enum weekdays  
Monday = 1  
Tuesday = 2  
Wednesday = 3  
Thursday = 4  
Friday = 5  
End Enum
```

You can refer to these constants simply by specifying their name.

```
Today = Wednesday
```

But what is the use then ?. Well, things change once you declare variables based from these enumerations.

```
Enum weekdays  
Monday = 1  
Tuesday = 2  
Wednesday = 3  
Thursday = 4  
Friday = 5  
End Enum  
  
Dim today as weekdays  
  
Today = wednesday
```

The moment you start typing 'Today =' you will get a pull down list with the possible choices. You have to think of an 'enumerate' as a way to aid you writing code. Your constants become part of the VB programming environment.

5.9 Numerical Operators

Visual basic supports the basic numerical operators. The conversion from one base to another is done automatically for you. The resulting number is stored into the format of the variable used to hold the result. If you use a variant then the number is automatically stored in the most precise format.

The order of execution obeys to the mathematical rules. You can force an execution order by placing calculations between round brackets (). Typically it is good programming practice to always write brackets. After all errors are quickly made to mathematical rules. This even gets worse when applying logical operators. After all can you tell what this results to ?

`X= 5*y-7 or int (sin z)-14 /12 ' ???`

If you change this to

`X= (((5*4)-7) or (int (sin z)-14)) / 12`

It gets a lot clearer.

Operator	Name	Function
+ - * /		Basic math operators
^	Exponent	Takes a number to the given exponent
Int	Integer	Strips off any digits after the decimal. It does not perform rounding !
Abs	Absolute value	Removes any sign from a number
Sgn	Sign	Extracts the sign from a number. 1 for positive -1 for negative and 0 for zero.
Exp	Antilog	Returns a number specifying e raised to a power
Sin	Sine	Std trig calculation Uses RADIANS !
Cos	Cosine	Std trig calculation Uses RADIANS !
Tan	Tangent	Std trig calculation Uses RADIANS !

Atn	Arctangent	Std trig calculation Uses RADIANS !
Log	Logarithm	Natural logarithm (e based 2.718282)
Rnd	Random	Generates a random positive number between 0 and 1
Sqr	Square root	Takes the square root of a number

The other operators can be derived from these. The VB help system has a complete list of thing like Sin-1 , cos-1 , Sin Hyp etc.

5.10 Base conversion

Sometimes you might feel the need to use hexadecimal or octal numbers. VB supports these bases as well. To specify such a number you add &h or &o in front to specify that a hex or octal number is following.

Example

```
X = &h3BC ' assigns hexadecimal 3BC to x  
Y = &o701 ' assigns octal 701 to y.
```

Note :

Visual Basic does not support the binary base !. However by including the VISION system you have access to this number format.

Note :

Converting strings to numbers and back is also considered base conversion. This is explained later on in the string manipulation chapter. The same rules apply. You can also put &h or &o in your strings.

5.11 Logical Operators

Apart from the standard mathematics operators Visual basic supports a set of logical operators. These operators only function on integer type numbers . You cannot use these operators on floating style numbers.

Command	Name	Description
NOT	Invert	In case of true false it inverts the condition. In case of a number every bit is flipped.
OR	Or	Standard logic operator Output is true when one input term is true
AND	And	Standard logic operator Output is true when all input terms are true
XOR	Xor	Standard logic operator Output is true if one and only one of the 2 inputs is true
EQV	Equivalent	Output is true if both inputs have the same state (this is actually the XNOR function)
IMP	Implication	consult the help system for more explanation

5.12 Flow Control

So far we have seen how to store data and perform basic mathematical functions with data. We also discovered how to group commands and expressions together in functions and procedures and how to pass data to these constructs and obtain results.

Besides these operations we need some way to control the flow of the program. Some kind of – what if – construct to decide which way the program should continue. That’s exactly what is coming next.

5.12.1 If then else

The most basic decision routine used in Visual basic is probably the if-then-else construction

```
if ( condition ) then
    ' if true
else
    ' if false
end if
```

Where condition is the result of an expression made up of 2 variables and a comparing function

Operator	Name	Operation
=	Equal	If the two variables contain exactly the same data this operator will set the condition to true
>	Bigger then	If the value contained in the first variable is bigger then the value of variable 2 then this expression will evaluate as true
<	smaller then	Like the previous but now var1 is less then var2
> =	More then or equal	
<=	Less then or equal	
<>	Different	

If the expression evaluates to true then the THEN clause will be executed. If it does not evaluate then the ELSE clause gets executed. This construct can be used to compare data and expressions and decide on which step to take next.

You can nest if-then-else clauses up to 255 levels deep. However , for a lot of these case you could use the if-then-elseif-else clause.

```
If ( x = 5 ) then
```

```
If y= 2 then  
    Debug.print "X=5 and Y=2"  
Else  
    Debug.print " X=5 and y <>2"  
End if  
Else  
    Debug.print " x <>5 . "  
End if
```

5.12.2 If-then-else / elseif

In some cases you will have a need for a more complex decision task. You can then use the elseif construct

```
if <condition1> then  
    <statement1>  
elseif <condition2> then  
    <statement2>  
elseif <condition3> then  
    <statement3>  
else  
    <default statement>  
end if
```

In most cases however it is easier to use the Select Case construct. This construct is explained next. However the If-Then-Else / ElseIf construct is required if you want to process objects. Select case cannot handle conditions resulting from object manipulation. This is a very rare condition however. It will be discussed later on.

5.12.3 Select case

If you need to do a lot of tests , or if a given expression can evaluate to a lot of different results then your best option is probable the Select case clause. The following example gives you an idea of the decision power the select case structure has aboard.

```
Select case (expression or variable)
  case 1
    print "You typed 1"
  case 2,3,4
    print "You typed 2 , 3 or 4"
  case 5 to 9
    print "You typed something between 5
and 9"
  case "A"
    print "You typed A"
  case "B","Z"
    print "You typed either B or Z"
  case "D" to "Y",
    print "You typed a letter between D
and Y"
  case "HELLO","HI","GOOD MORNING"
    print "Hello to you too"
  case "BYE","SEEYA"
    print "Goodbye , Have a nice day"
  case else
    print "huh?"
end select
```

As you can see you can specify values or strings , and ranges of values. When the expression or variable is checked against the Case clauses , the compiler will scan from top to bottom. The first clause to match the comparison will be executed. If no match is found the CASE ELSE clause will be executed.

5.12.4 Loop Constructions

Often you will require a process to be repeated a number of times or until a certain condition is met. This calls for LOOP constructions. There are three basic forms of loops. One that runs a given number of times, and one that runs until a condition is met.

5.12.5 For Next

This is the basic looping construction. You use this to execute a certain action a given amount of times. The For-Next clause can handle all variables.

```
For x = 1 to 10  
next x
```

Counts from 0 to 10

```
for y = 10 to 0 step -1  
next y
```

This counts from 10 to 0

```
for z = -1.5 to 125 step +0.01  
next z
```

Counts from -1.5 to 125 in steps of 0.01 : -1.5, -1.49, -1.48 ... 124.98, 124.99, 125.

Note :

It is not allowed to change the counter value from within the loop. This is a common mistake that is often made. The construct allows you to do this. However this can lead to system lock-ups. Programmers often use this technique to prematurely abort the execution of the FOR loop. DON'T do this. ! There is an Exit For statement

If you need to exit the For-Next loop prematurely then you should use the Exit-for statement.

```
For x = 0 to 100000  
    If x = 125 then exit for  
Next x
```

The counter would normally run from 0 to 10000. However if it reaches 125 the Exit-for command will abort it end resume execution after the Next x.

5.12.6 While wend

This is a looping procedure that runs as long as a certain condition is true.

The checking of the condition occurs before execution of the sequence. This means that if the condition is already met from the beginning , the sequence will not be executed.

```
While <condition>  
    < statements >  
Wend
```

If you need to abort this sequence you should use the exit-do statement.

```
While x <5  
    X=X+1  
    Debug.print x  
Wend
```

The above construction will run until $x = 5$ and then exit. If I set x to 6 initially the While-Wend will not be executed. The test is at the top. So X will not be incremented to 7. This is different in the next construction

5.12.7 Do Until

This is the other looping construct in Visual basic. It is similar to the While-Wend construct except that the testing of the condition happens at the end of the sequence. This means that, no matter what, the sequence will always be executed at least once.

```
Do  
    <statement>  
    <statement>  
    ...  
Loop until <condition>
```

Example :

```
Do  
    X =x +1  
Loop until x >5
```

If I start here with x initialized to 6 then it will be incremented to 7 before it is tested against the condition clause.

The same rule as for the While-wend applies. Never jump out of this construction. Use the exit-do command. The problem with jumping out is that some residual stuff is left on the stack of the program execution. The compiler

checks for this and inserts cleanup code to work around this. However this is not fail safe and might lead eventually to a system level crash.

5.13 String manipulation Left\$ - Right\$ - Ltrim\$ - Rtrim\$

When working with strings you will often manipulate their contents. In VB there is a rich instruction set to manipulate strings.

Suppose a\$ contains "How are you?"

5.13.1 Left\$

This command takes the left n characters from a given string

```
t$ = Left$ (a$,3) ' t$ now contains "How"
```

5.13.2 Right\$

This command takes the right n characters from a given string

```
t$ = Right$ ( a$, 4 ) ' t$ now contains "you?"
```

5.13.3 Mid\$

Just like it's smaller brother Left\$ and Right\$ this allows to you to extract a given amount of characters starting at an offset in a string

```
a$ = "How are you ?"  
t$ = Mid$ (a$, 4, 3 )    ' t$ will contain "are"
```

5.13.4 *Ltrim\$ / Rtrim\$ / Trim\$*

These functions remove 'whitespace' at the beginning and / or end of the string. 'whitespace' is any non-printable character. So everything that is not a letter , number or punctuation mark will be removed. These functions are very useful to manipulate user input. Or file input from an unknown origin.

```
A$ = LTrim$(b$)
```

5.13.5 *Ucase\$*

This function converts a given string to an all-uppercase string.

A very useful combination of these functions is often the following :

```
a$ = Ucase$ ( Ltrim$ ( Rtrim$ ( a$ ) ) )
```

This strips off any leading and trailing 'whitespace' and converts it to an uppercase only string. If you want to write a small command interpreter or macro tool you will use this construct very often.

5.13.6 VAL and STR\$

These functions are used to convert numbers to and from string. The VAL function extracts a number form a string. The routine stops scanning at the first encounter of a non-number character. VAL is also capable of recognizing scientific format numbers (-1.2 e-15) and different based numbers

```
print val ("10 hello")    ' prints 10
print val ("&h10 bye")    ' prints 16
print val ("-1.55 test")   ' prints -1.55
print val ("-1.5e-55 ok")  ' prints -1.5e-55
```

The STR\$ function has the opposite effect. It converts any number to its string style representation..

```
a$ = Str$( 125 )    ' will return as string
containing "123"
```

5.13.7 LEN

While not really a string manipulation function it is used in conjunction with these functions. LEN tells you exactly how long a string is. If it is empty a Zero is returned.

```
B$ = "HELLO"
For x = 1 to len(b$)
    Debug.print mid$(b$,x,1);" ";
next x
```

This will output 'H E L L O' to the debug windows. It checks how long the string is and then will extract the character one by one , print them and send a space behind

5.13.8 INSTR

This is a search routine that allows you to find strings in other strings. You can use this to search for words or special characters

```
X = instr("HELLO","E") ' will return 2
```

Upon execution X will contain 2. This means that an 'E' was found at position 2. You can also specify an offset

```
X = instr("HELLO THERE","E",3) ' will return 9
```

Now the result is 9 because you started searching at position 3 (the first L) and found an E at position 9.

5.14 File Manipulation (Open – Close – Print – Input)

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 be numerical , text , binary or even an entire database with linked lists , records , custom styles etc .. Well you could have not picked a simpler language . Basic in general is probably the only language where file manipulation is so simple , yet at the same time so extended.

Due to the numerous things you can do with files this topic will be covered many times over the course of this manual. In this section we will only cover basic file manipulations.

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. The handle itself 'points' to this information.

A handle is specified by using the # sign followed by a number. You can use your own numbering scheme or you can ask the system to give you a handle. The function '**freefile**' checks for a free handle, allocates it and returns it to you.

5.14.1 Basic structure to open a file.

Open <filename> **for** <mode> **as** <handle>

where:

<Filename> is any valid filename. This can include drive / network path / path / filename <mode> can be **input**, **output**, **random**, **binary**, **append** and <handle> any handle which is not already in use.

```
        outfile = freefile      ' retrieves a free
handle
        appfile = freefile      ' retrieves another free
handle
        myappfile = "appfile.txt"
        open "myfile.txt" for input as #100
        open "myfile.out" for output as outfile
        open myappfile for append as appfile
```

The above examples show you the basic modes of operation and the different ways you can specify a filename and handle. The open command also has some optional parameters that allow you to share or lock it while you are working with it. This is useful when running files over a network (databases). You can even specify if it is to be locked for write or read. This would lead us too far for now.

Whenever a file has been opened it should be closed when done with. Therefore you can use the close command. If you use CLOSE without parameters you will close ALL handles to any file currently open. If you use 'close' with a file handle it will close this file. If the handle does not exist it will do nothing.

```
Close #1 ' close file with handle 1  
Close ' close all files
```

15.4.2 Output mode

When you open a filename for output then 2 situations can occur : either the file exists or it doesn't. When it doesn't then it will be created . If it exists it , a copy will be made to the same name but with a .BAK extension and then the original file will be overwritten.

Data can be written to a file using the print command in conjunction with the handle

```
outfile = freefile ' retrieves a free  
handle  
open "myfile.txt" for output as outfile  
print #outfile,"HELLO"  
close outfile
```

5.14.3 Append mode

In case you don't want to overwrite an existing file but store more data into it you can use the append mode. The data you send to it will be written at the end of the current file. To store data you can again use the print method.

```
open "myfile.txt" for append as #1
```

5.14.4 Input mode

If you want to read something from a file this is one of the possible ways to retrieve the data. This opens a file for read.

```
open "myfile.txt" for input as #1
```

5.14.5 Storing something in a file

This is easy : just use Print to send it to the file

```
myfile = freefile  
open "test.txt" for output as #1  
print #1 , "Hello world"  
close #1
```

Any accepted print expression can be sent to a file.

Note :

This is the only place in visual basic where you can still use the Print statement like it used to be implemented in regular basic. All other cases treat Print as a method of an object. Even the DEBUG object .

Since I have not explained print yet this is a good point to do it.

5.14.6 PRINT constructions (file I/O)

Print is a tremendously versatile command. You can send nearly anything to it for printing. In Visual Basic you can only use the native print in combination with files. All other print statements are actually methods of objects (textboxes , printers , even the Debug object).

There are two basic ways of invoking print : send it a data list , or first build a string and then send it. The end result is the same but the execution is not. A data list requires the compiler to move all data bit by bit to the printing code. This takes time but does not use memory. The string-building way requires scratch memory to build the string but is a lot faster since only the entry point to the string is passed. Anyhow , in today's optimizing compilers the end result is the same.

5.14.6.1 Data list Style

```
Name$="USER"  
Number=1  
Print #1,"Hello ",name$,"You are my No",number
```

This will output "Hello USER You are my No 1". Strange ? No !. The comma really means 'move to the next tab'. If you do the following

```
Print #1,"Hello ";name$;" You are my No";number
```

You will get a string as you would expect it. The string printed automatically gets a CR/LF pair appended (Carriage return Line Feed). If you want to suppress this you simply append a semicolon at the end of the expression.

```
For a = 0 to 10  
    Print a  
Next a
```


Results in a list of numbers

```
For a = 0 to 10  
    Print a;  
Next a
```

Results in "12345678910"

5.14.6.2 String style

```
Name$="USER"  
Number=1  
Print #1,"Hello "+name$+"You are my  
No"+str$(number)
```

This first builds a complete string and passes it to the print command.

Note :

You can only pass strings to this style. So you must manually convert any numbers to strings before adding them with the + sign.

The same trick as with the 'Datalist' style applies here. By ending with a semicolon you can omit the CR/LF insertion. Instead of using the + operator you can now also insert the & operator. This is new in Visual Basic. The end result is the same.

5.14.7 Reading from a file

Here you have several options but the most used will be the **Line Input**. This retrieves an entire line from a file. It scans the file from the current location to the first occurrence of a CR/LF (carriage return / line feed). There are other

ways to retrieve data but they are used to extract records, a known amount of bytes or binary data. These functions will be discussed later on in this book.

```
Open "myfile.txt" for input as #1
while not eof ( 1 )
    line input #1,a$
    textbox1.text = textbox1.text +a$
+Chr$(13) +Chr$(10)
wend
close #1
```

The above piece of code will read an entire text file and dump it into a textbox. Since the **Line Input** statement reads a line , but removes the trailing CR/LF pair we have to add it to the textbox.

5.14.8 Determining file end

When you are reading from a file you should take care not to read beyond the end of it. This will result in an error. There is a function EOF that returns you whether you have reached the end of the file. The above example shows you how to use it. Not that you can use this only in conjunction with the input mode. If you are manipulating binary files you can read beyond the end. There you should use the LOF operator before performing any read. More on this later on in the book.

5.14.9 File names

Any valid windows file can be opened. Attempting to open a non-existing file can have two outcomes. In case the mode is output it will be created. In case the mode is input you will be stuck with a runtime error. You should especially take care with routines that allow the user to specify a file. There are ways to detect the validity of a filename. However ,there is a much simpler way. Simply use the CommonDialog FileOpen and FileSave to handle all of this for you. This object is dealt with in Part II.

Chapter 6 :

Creating a user interface

Lets take a look on how to make a form for our project.

6.1 Creating The Form

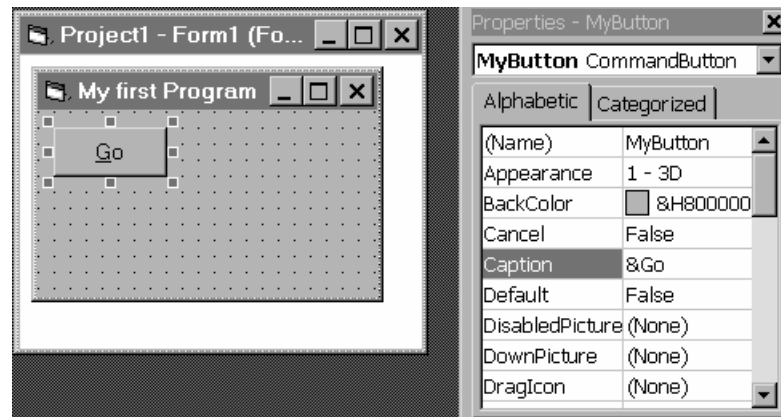
We start the VB environment and select standard EXE format. An empty form is being displayed. To add control you select a control out of the Control Toolbox and then drag the outline of it on your form.

All controls on a form take up control handle space available to the form . A form has a handle space limited to 255 (one byte) handles. This means a form can have a maximum of 255 controls. There are ways to extend this number . They will be discussed in the examples later in this guide.

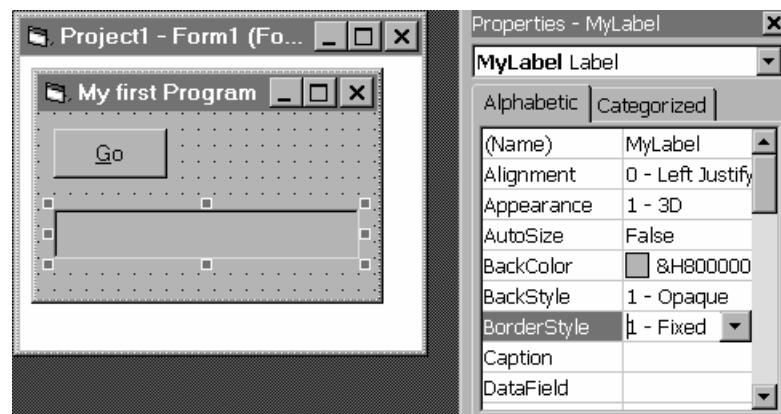
Note :

In a project containing multiple forms this still means that every form can have 255 controls.

Lets add a button to our form and edit the properties Caption and Name of it
The Name has been set to MyButton and the caption to Go. As you can see an ampersand (&) also denotes a hotkey , just like in the menu editor

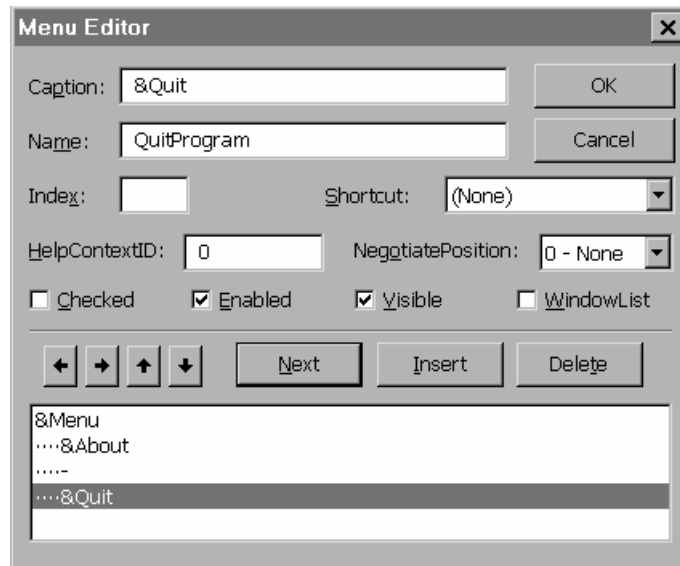


Now lets add another object. Lets say a label . We will call the label Mylabel and set its caption to an empty string



Now we have an object that can generate events (the Command button) , and an object that can be used to display something. We now still need a way to give

the user the option to terminate our program. A Windows program is not like a classic program with a beginning and an end. The flow of a windows program is not linear. When the program is started a lot of stuff is happening at the same time . It's the users option to terminate it. So you need to have an explicit means of terminating a program. We could give the user this means using another command button , but a menu looks nicer



This menu has 3 entries. The first will show the user a message about the program . The second entry is a divider line and the last one will allow the user to Quit the program. We have now come to the stage where the user interface is ready and we can start attaching code to our program.

6.2 Arrays of Objects and Controls



CTRLarray.vbp

In some cases you will have more than 255 controls. Or you might want to have an easy indexing system using controls. A typical example is a keypad. Instead placing 10 command-buttons on the form you can place an array of 10 command-buttons . This uses only one handle. Furthermore the attached code is common for all controls; You have to write only one procedure to handle all these events.

Creating arrays of objects is easily done by drawing the first object , giving it a name and then copying it. (*<CTRL> - C / <CTRL>- V*)

After you place the first copy the environment will ask you if you want to create a control array. You answer yes. You will notice that the **index** property will contain a value. This value indicates the position in an array.

For a keypad you would create a button called Keypad and give it as caption '0'. This will be the item 0 in the array. Then you copy it . Since you answered yes on the question Create a control array ' the newly placed object will get 1 as index. And so on. All you have to do now is change the caption accordingly.

When you click any of the controls in the control array you will see the same piece of code. But remark that now a parameter called index is being passed. This indicates you which of the elements in the array really generated the event. In case of our keypad the index relates directly to the number.

Another application of control arrays is to store labels. A typical form might contain a lot of labels that are wasting valuable resource space. Since labels have no use in generating events (nobody will ever do something with them apart from reading them) it's a good idea to store them in an array. That way you only use one handle.

One nice thing about control arrays is that you can add or remove elements at runtime. You can for instance add a button to the screen from within runtime. Or you can add an item to a menu bar (provided the part of the menu has been created as a control array).

Note :

You cannot remove the first item in a control array. Typically the first item is set invisible and then items are added as needed with the visible flag turned on.

The CD-ROM contains a sample project that shows you all of this functionality.. Feel free to explore this piece of code and or use it as a reference when creating your own programs

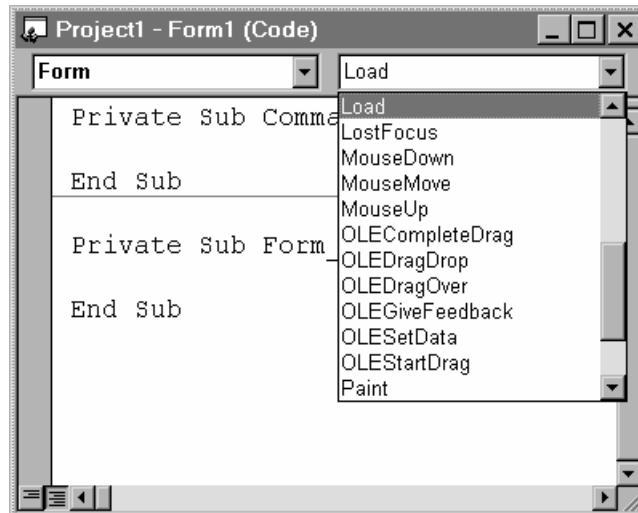
Chapter 7 :

Attaching code to your form

The events generated by user activity will invoke different parts of your program. To specify what you want to be done you have to start writing code. Since you are working in an event driven world the actual code writing will be limited to what cannot be done by the system.

7.1 Attaching code to objects

To add code to an object you simply double click it. A code editor window will open and show you the code attached to it. By default the most used event is displayed. In case of the button this is the Click. This can be different for every control. In case of text box the default event will be the change event. You can select events belonging to an object using the right pull-down menus on the code window. With the left pull-down menu you can also select an object .

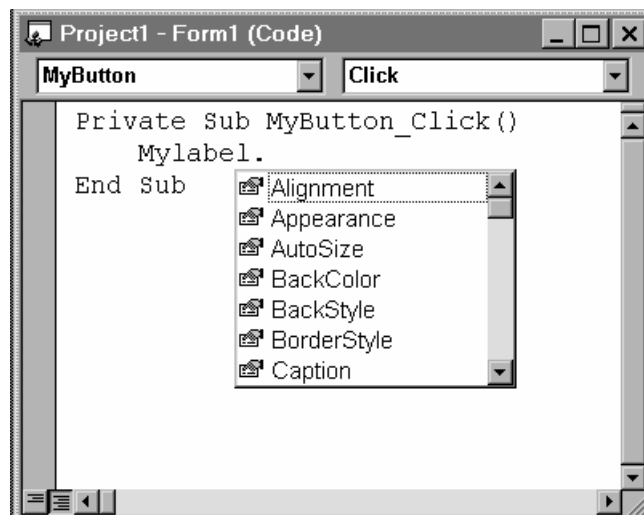


The code editor has a number of very interesting features which make programming very easy. As you are typing code the editor evaluates what you are typing and show online help. If you type the name of one of the objects and put a dot behind it, the system will automatically show you a list of all the properties and methods you can control from within code. If you are calling functions or procedures, it will automatically show you what they expect and in what order. This even works for your own defined procedures and functions.

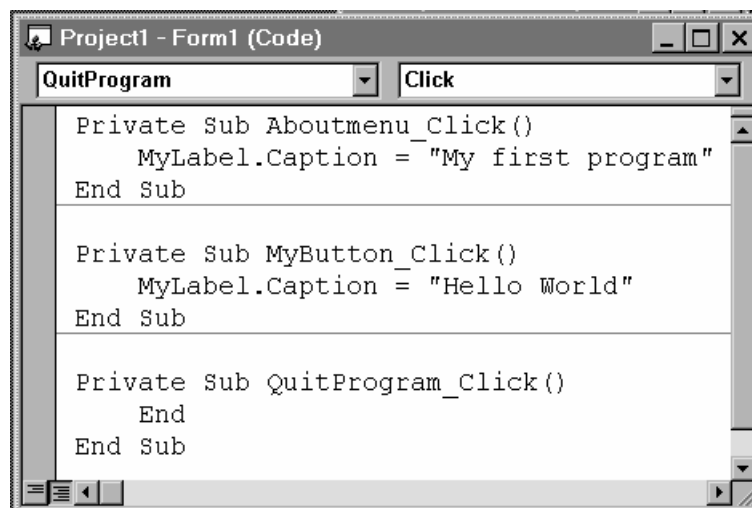
The moment you reach the end of the line and press return the editor evaluates what you just wrote and will comment if it finds any syntactical errors in your line. this will help you already eliminate typing errors and syntax errors before you even compile it the first time.

7.2 Let's Attach some code

So lets attach some code. First of all we want to display the text 'Hello World' when we click on the Go button. To do this you simply double click the Go button and the code viewer will present you with the correct routine.



While you are typing the editor evaluates what you are typing and attempts to assist you. Since we want to change a property (the Caption) of MyLabel , we start typing Mylabel. (note the dot !) , and then the editor kicks in and shows a list of what properties can be changed. It is sufficient to type now a few letters of the property name until the selector bar is right. You can now select this property by pressing the TAB button , or you can type the name completely.



The above figure shows you the complete program. When you click the 'Aboutmenu' item (on the Menu bar) then the caption of the Mylabel object is going to be changed into 'My first program'. The same goes for a click on Mybutton. Finally a click on the Quit button will terminate the execution of the program.

Note :

In general the code attached to a form should only contain directly linked subroutines or functions. If you need to define a dedicated function , store it in a **Module**. This keeps your code transparent , easier to read and maintain. For more information refer to chapter 9..

Now we are ready to run and compile our first program.

Chapter 8 :

Running and debugging a program

Now that you have created a user interface and attached code to it's about time we check if it actually does something . The VB environment allows you to run and debug your code in an easy to use way. This chapter will detail on how to trace and fix errors rapidly.

8.1 Running a program

To run a program you can either select the Run-Start (F5), Run-Start with full compile (shift-F5) or the Run button on the toolbar. When you press the Run button on the Toolbar you are actually just executing the Start command.

8.1.1 *Start , Break , Stop*



The difference with them is the following : When you come from the situation where your code is not running it doesn't really matter. Both options will

compile your code into an exe file and launch them. The difference is when the program aborts due to an error. If you debug your code and modify it you can continue the run by executing the Start command. If you select the ***Start with full compile*** your program will restart it's run completely.

When your program is running you can halt execution by using CTRL-Break. Or by clicking the Break button on the toolbar.

Upon Break , you will be shown your code . you can then edit and continue running the program. This allows you to make on-the-fly modifications and see their impact.

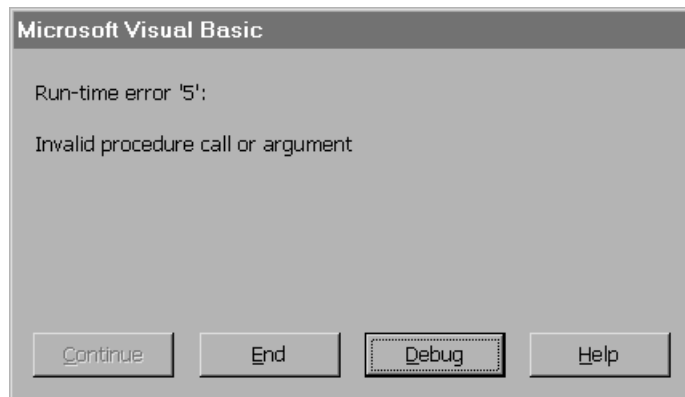
Note:

With break you cannot edit the user interface. Only the code can be modified. The reason is that the control structures need to be recompiled if you change the interface. Your code is interpreted on the fly. When you really make an EXE then your code also is compiled into machine language

The stop button finally ends the execution. If , during the execution of your program, a real error occurs then visual basic will halt execution and ask you what to do.

8.2 Debugging a program

If for some reason , something goes wrong (now how did that happen ? ☺) , you will be presented a warning and a number of options on how to proceed.



The runtime error-code is displayed together with a brief blurb on what the code means.

If you select End then the execution simply halts. Pressing help will display information about the nature of the error. The Debug button however is the most interesting.

When you press debug the code viewer will take you immediately to the line in your code where the error occurred. Now you are in Debug mode. Now you can trace the flow of your program, examine variables etc.

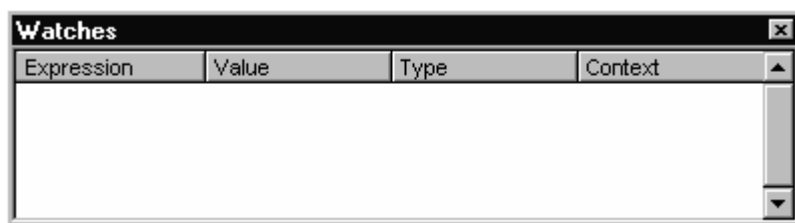
8.3 Examining Variables

You can examine the contents of the variables and properties you access inside the current procedure. This is useful to detect if some parameters are being passed correctly or if you don't misuse certain variables.

To do this just move your mouse over a variable or property and a small box will appear below the mouse-pointer to show you what is stored in the variable. Most problems are related to variable abuse.

8.4 Advanced Debugging : The Watch Window

This window appears automatically when watch expressions are defined in the project.



You can:

- Change the size of the column headers by dragging its border to the right to make it larger or to the left to make it smaller.
- Drag a selected variable to the Immediate window or the Watch window
- Close the window by clicking the Close box. If the Close box is not visible, double-click the Title bar to make the Close box visible, then click it.

8.4.1 Window Elements


Expression	Lists the watch expression with the Watch icon, on the left.
Value	List the value of the expression at the time of the transition to break mode. You can edit a value and then press ENTER, the UP ARROW key, the DOWN ARROW key, TAB, SHIFT+TAB, or click somewhere on the screen to validate the change. If the value is illegal, the Edit field remains active and the value is highlighted. A message box describing the error also appears. Cancel a change by pressing ESC.
Type	Lists the expression type.

Context	Lists the context of the watch expression.
---------	--

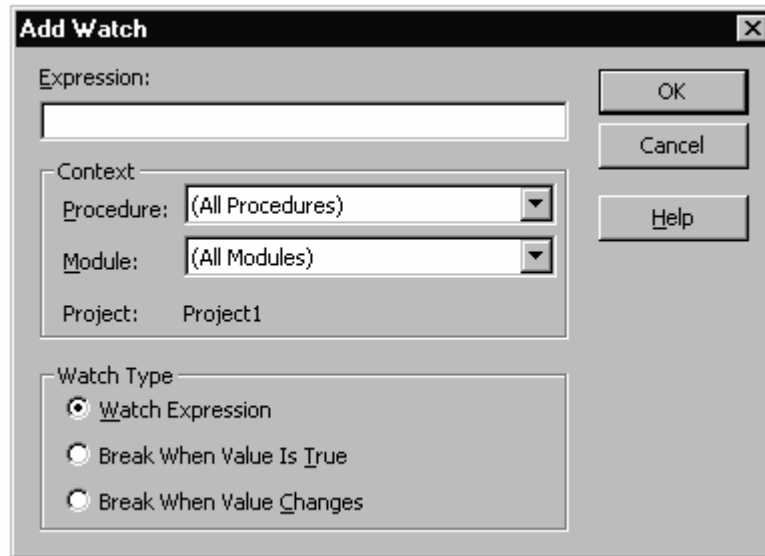
If the context of the expression isn't in scope when going to break mode, the current value isn't displayed. You can close the window by clicking the Close box. If the Close box is not visible, double-click the Title bar to make the Close box visible, then click it.

8.4.2 Add Watch command

At design time or in break mode, this command displays the Add Watch dialog box in which you enter a watch expression. The expression can be any valid Basic expression. Watch expressions are updated in the Watch window each time you enter break mode.

Toolbar shortcut: 

8.4.3 Add watch dialog box



Use to enter a watch expression. The expression can be a variable, a property, a function call, or any other valid Basic expression. Watch expressions are updated in the Watch window each time you enter break mode or after execution of each statement in the Immediate window.

You can drag selected expressions from the Code window into the Watch window.

Important When selecting a context for a watch expression, use the narrowest scope that fits your needs. Selecting all procedures or all modules could slow down execution considerably, since the expression is evaluated after execution of each statement. Selecting a specific procedure for a context affects execution only while the procedure is in the list of active procedure calls, which you can see by choosing the Call Stack command on the View menu.

Dialog Box Options :

Expression :

Displays the selected expression by default. The expression is a variable, a property, a function call, or any other valid expression. You may enter a different expression to evaluate.

Context:

Sets the scope of the variables watched in the expression.

- **Procedure** : Displays the procedure name where the selected term resides (default). Defines the procedure(s) in which the expression is evaluated. You may select all procedures or a specific procedure context in which to evaluate the variable.
- **Module** : Displays the module name where the selected term resides (default). You may select all modules or a specific module context in which to evaluate the variable.
- **Project** : Displays the name of the current project. Expressions can't be evaluated in a context outside of the current project.

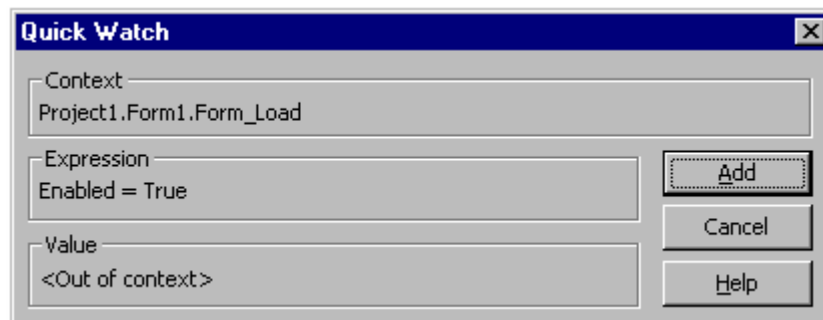
Watch Type Determines how Visual Basic responds to the watch expression.

- **Watch Expression** : Displays the watch expression and its value in the Watch window. When you enter break mode, the value of the watch expression is automatically updated.
- **Break When Value Is True** : Execution automatically enters break mode when the expression evaluates to true or is any nonzero value (not valid for string expressions).
- **Break When Value Changes** : Execution automatically enters break mode when the value of expression changes within the specified context.

8.4.4 Quick Watch command (Shift F9)

Displays the Quick Watch dialog box with the current value of the selected expression. This is only available in break mode. Use this command to check the current value of a variable, property, or other expression for which you have not defined a watch expression. Select the expression from either the Code window or the Immediate window, and then choose the Quick Watch command. To add a watch expression based on the expression in the Quick Watch dialog box, choose the Add button.

8.4.5 Quick watch dialog box



Displays the current value of a selected expression. This functionality is useful when debugging your code if you want to see the current value of a variable, property, or other expression.

Dialog Box Options

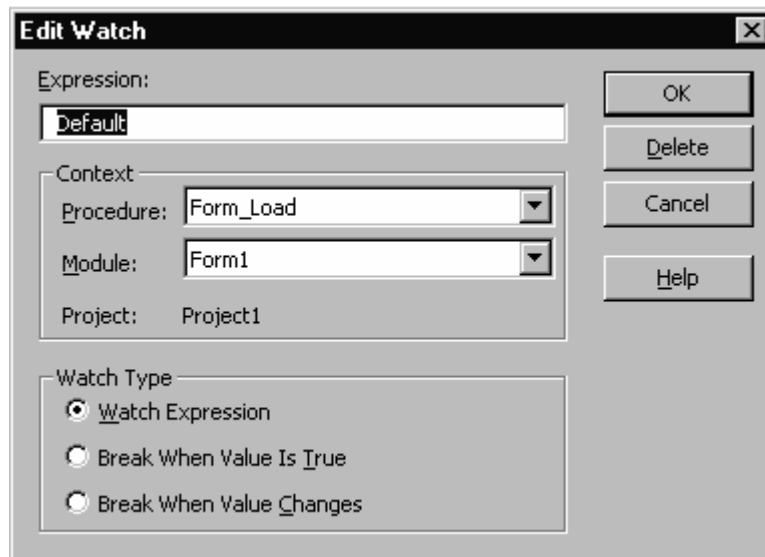
- **Current Context:** Lists the names of the project, module, and procedure where the watch expression resides.
 - **Expression :** Shows the selected expression.
 - **Value:** Shows the value of the selected expression. The current value isn't displayed if the expression context isn't within a procedure listed in the Calls dialog box.
-

8.4.6 Edit Watch command

Displays the Edit Watch dialog box in which you can edit or delete a watch expression. Available when the watch is set even if the Watch window is hidden. Not available at run time.

Toolbar shortcut:  . Keyboard shortcut: CTRL+W.

8.4.7 Edit Watch Window



Use to delete or edit the context or type of a watch expression.

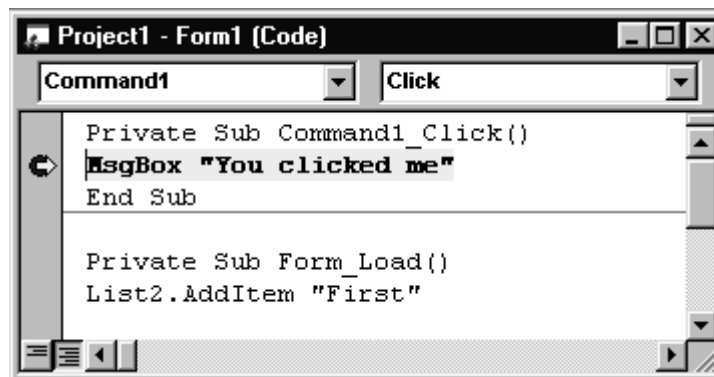
Important When selecting a context for a watch expression, use the narrowest scope that fits your needs. Selecting all procedures or all modules could slow down execution considerably, since the expression is evaluated after execution of each statement. Selecting a specific procedure for a context affects execution only while the procedure is in the list of active procedure calls.

8.5 Using Breakpoints

Breakpoints are maybe the most important trick in the debugger's hat. You can set a breakpoint on any **executable** line of code. When , during execution , the compiler reaches a line of code with a breakpoint set it will halt the execution. You can then examine variables or change the flow of the program.

Note:

Breakpoints are saved into you project. When you compile code containing breakpoints they are suppressed. Compiled code cannot be halted by breakpoints. They only work inside the IDE of Visual Basic



You can set a breakpoint by clicking in the column before the line you want to set it. A red dot will appear showing you a set breakpoint. Clearing it is equally simple. Simply repeat the action. When the code is running and reaches the breakpoint a yellow arrow will appear in front of the line where the run was halted. You can examine variables now. You can also move the arrow down or up. This way you can alter the program flow. Be careful however, this is tricky stuff and might not always lead to what you intended it to.

Note:

Breakpoints halt the tagged line. If there are multiple commands separated by semicolons on a single line then the first of them is halted. The others cannot be halted. In order to use breakpoints to their full extent you should make sure only one statement per line of code is present in your program.

8.6 the Debug Object

Another way of examining data and program flow is to insert calls to the DEBUG object. This is an embedded object of the IDE. When compiling code this object is made empty. This means that the calls are omitted. So the end user will not see these messages. You can call the immediate window (debug window) by pressing CTRL-G. You can simply print messages to debug by referring it as debug.print "something".

```
Sub Form1_load( )  
    Debug.print " program started"  
End sub  
  
Sub Quit_click( )  
    Debug.print " Bye !"  
    End  
End sub
```

The contents of this immediate window are preserved even after the run of the program is terminated. This means it can be used to track nasty bugs in bailout code , or things that happen even before a user interface is visible.

You cannot delete text from this window while the program is running. Once it is stopped you can select lines and use cut copy and paste commands just like you can with any regular window.

Chapter 9 :

Distributing a program

Now you have come to the point where you have program that is to your best knowledge bug free. The last step is to make a distributable version of the program.

9.1 The First steps ...

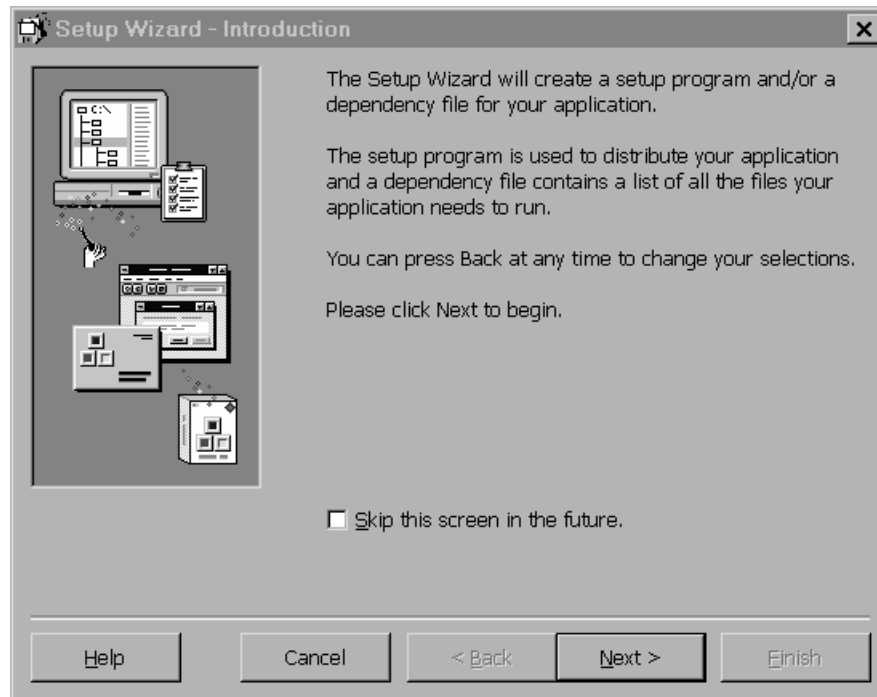
The first step is to compile it to an executable format. Clicking on the 'File' menu and selecting 'Make Executable' in the design environment can easily do this.

If you click this menu item the compiler will build an executable version of your program.

You can now run this project on your computer without needing Visual basic. However if you would like to distribute it you might run into some trouble. For starters , your project might use certain controls that reside in separate .VBX .OCX or .DLL files. Sometimes it's not always easy to figure out what exactly is needed. Furthermore your target user might not have the correct version of the files you use in your program.

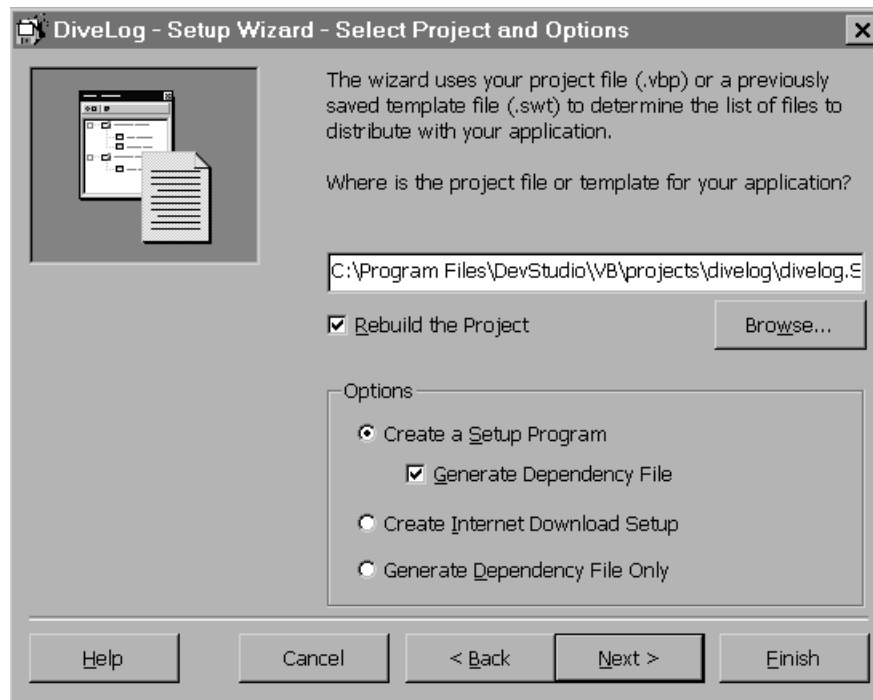
But , don't despair ! VB has a wizard aboard which does all of this work for you and creates a nice set of floppy disks you can use to install your program onto

another computer . More , it even creates a nice Setup and uninstall program that gives an extra Pro-touch to your application.



This wizard can be found in the program group of visual basic; It will guide you step-by-step through the creation of the distribution kit.

For the most part of the process all you have to do is clicking the Next button . However some pages are interesting , and will be detailed on next.



The first step you have to do is select the program's VBP file. It's a good idea to select the option 'Rebuild the project'. This will force a clean compile of all of your code. Furthermore you will be sure the latest changes and bindings are installed.

Note:

Bindings are the links between your program and external modules. These bindings contain also module version information. It's important to distribute the right version of the external files. Otherwise your program might not be able to run on someone else's computer.

In the Options section you can select the kind of operation you want to perform. For now the standard 'Create a setup program' will do just fine. However make sure the 'generate Dependency File' is switched on. This again is used for the bindings in your program.

9.2 Specifying the Media



The next screen allows you to specify the kind of setup you want.

If you specify floppy disk you will have to make sure to have enough empty and formatted floppy disks at hand. A better option in that case is to specify 'disk directories'. This will store the contents of the floppy disks in files on your hard disk. You can then later make the distribution floppies.

Single directory is used if you want to store everything in one huge file or want to write a CD with the software on.

The setup wizard will now ask you where you want the distribution files to be located. You can select any valid directory. It's a good idea to start in an empty directory. The wizard will not touch existing files in the directory (if any should exist). Therefore if the directory is not empty you might end up with more files than you bargained for.



The next screen will allow you to add custom files. This is interesting if you want to add 'readme' files or setup files to the distribution kit. These files will be packed also and installed on the target computer.

After this you will see a list of files that the wizard thinks are necessary for your program. You can edit this list at will. However this is not such a good idea since you might delete files that are really necessary.

For the remainder of the work you can simply click Next all the time. At a certain point you will see that the wizard starts gathering the required files and will compact them. In the final stage it will compile the actual Setup.exe program.

When all is done the only thing you have to do (depending on the output format you selected) is send the floppies to your user , or copy the files onto floppies , or maybe zip them and send them over the internet.

Chapter 10 :

Multi-module projects.

During your programming work situations will arise where you will need more than one form. You might want to give the user an options or setup form , or an about form. Sometimes your program will include also custom routines that are not directly related to events , but are called from other routines.

In any of these cases your project will be a multi-module project. To be honest , nearly every project , no matter how small , will most likely turn out to be a multi-module project.

You can add items using the Project menu. To create a form simply select 'Add Form'. Similarly a module can be created using the Add Module item.

10.1 Multiple Forms

Remember when we discussed the basic project form , we talked about the startup form. Typically the first form you ever draw in a project is the startup form. Other forms remain hidden until you call them. You can call in a form with the Show method. Similar , you can hide it using the Hide method.

```
MySecondform.Show
```

When the form is loaded then the **focus** is automatically set to the new form. This means that keyboard and mouse operations will refer to this form. You can select a different form by clicking it. In some cases you might want to 'lock' the form. This means the user can do nothing until ha closes the new form. This can be done using the vbModal option . If you specify this option then the user can only work with this form until it gets closed.

```
MySecondform.show vbModal
```

To ease the programming work Visual basic has also something called a MessageBox. This is a kind of predefined simple form that you can use to interrogate the user. A number of parameters allow you to change the look and feel of this form

```
MsgBox "Hello World", vbOKOnly + vbInformation,  
"My first Message"
```

This gives the following result :



10.2 Modules

Apart from forms there are also things called modules. A module is a piece of code that has no user interface. It generally contains variable definitions, and user subroutines and or functions. It provides a means to neatly organize your own functions.

A calculator program might have a custom function called calculate which takes in two numbers and an operator and returns the result.

```
Function Calculate (a, b, operator)
    select case operator
    case plus
        result = a + b
    case minus
        result = a - b
    end select
    calculate = result
end function
```

A module is also the place where you define your variables and constants. You have to consider a module as a separate process. When your application is compiled all modules are evaluated and their definitions are created. Then the remainder of the module is compiled to a library and linked to the other parts of the program.

10.3 Accessing items from other parts of the program

Since every form acts like a standalone unit this also means that 2 different forms can have a control with the same name. The controls are completely different since they have different handles.

Suppose a 3 form project and every form has a command button to close the form. Logically you would call every close button simply CloseForm. Now suppose you have a routine that closes all 3 forms. You could simply invoke the commands from this routine. The only problem is knowing which one.

Well the answer is simple. You just specify the parent object and then the desired object belonging to this parent. In our case the code would look like this

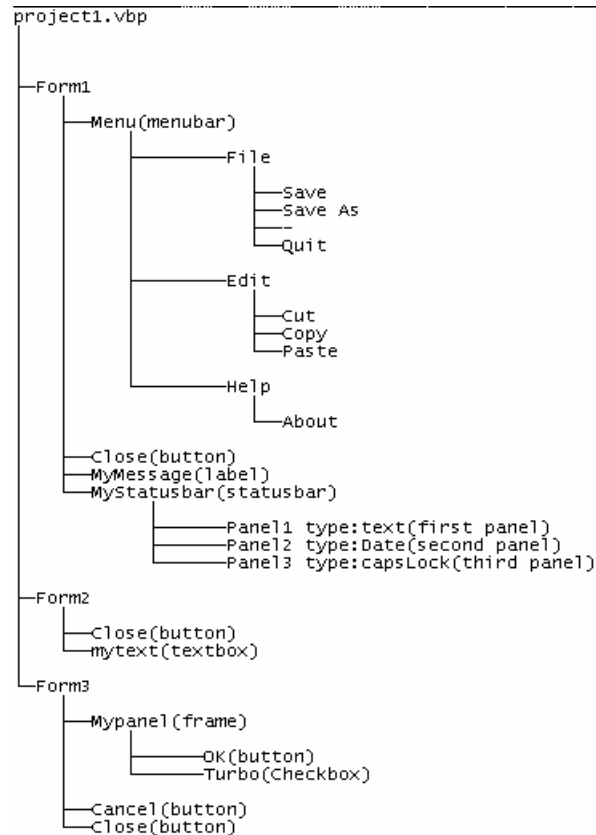
```
sub Closeall()  
    call form1.close_click ()  
    call form2.close_click ()  
    call form3.close_click ()  
end sub
```

This same rule applies to procedures inside a module. Actually your own built procedures are no different then the ones attached to objects. You have to specify the target object using its complete denominator. You can make an analogy between a project and a hard disk.

The hard disk is the project itself. The user interface is the root account. In the root of the project are forms (subdirectories) . Each form contains objects (files) . Every object on this form can contain further objects (another layer of directories) . To reach an object from any given location you have to specify the complete search path. Objects belonging to the same level (directory) can find each other since they reside at the same level.

The only quirk in this analogy is that in the root there can only be a special kind of object (forms) and they can only reside there.. once you go down then you can have directories made of other objects.

10.4 Root structure analogy of a project



As you can see in this graphical representation even a menu is a collection of objects. Keep in mind though that some objects can contain others. Some special objects like Forms and menus are 'awkward'. These objects have a special function. They are called parental object. This means that they form the basis from which the operating system detaches messages. A menu bar is also an object, but a special kind that can only be linked to a parent of the class 'Form'.

The particularities of this matter are dealt later on where the creation of custom objects will be discussed.

Chapter 11 :

A couple of case studies

This chapter will guide you step-by-step through the creation of a couple of small programs. This will provide you with a better understanding on how a program is written in Visual basic.

The first program is a small text viewer / editor. It allows you to view and edit files.

Topics such as insertable objects and system objects , textboxes and menus will be explained. You will see how to open , read and write , and close files. Furthermore it makes use of some of the components built into windows like the clipboard

The second program describes a calculator.

This will deal with arrays of controls and creating a multi-module project. It will show you how you can heavily optimize code by creating arrays of objects and writing your own custom functions and procedures.

11.1 Case Study 1 : A small Text Editor

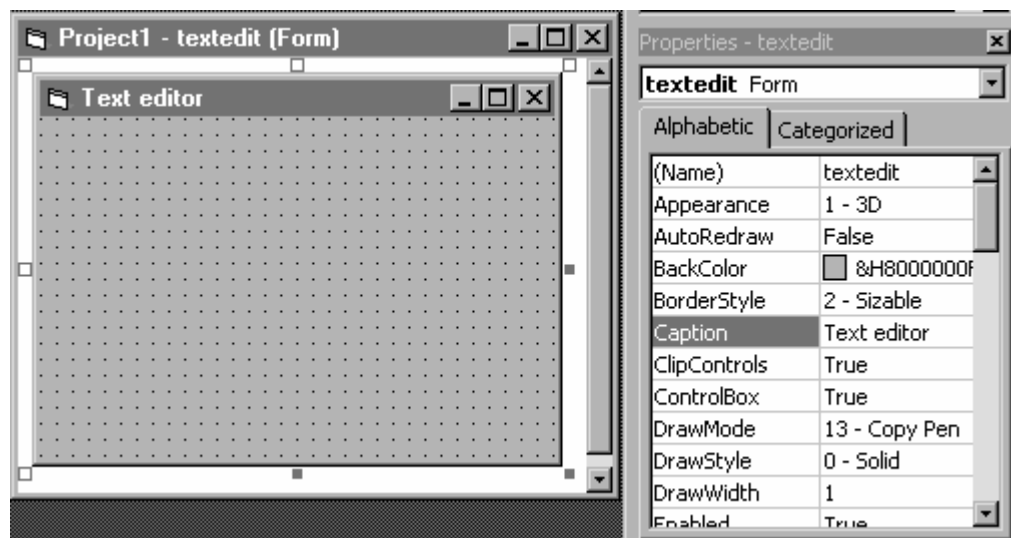


Textedit.vbp

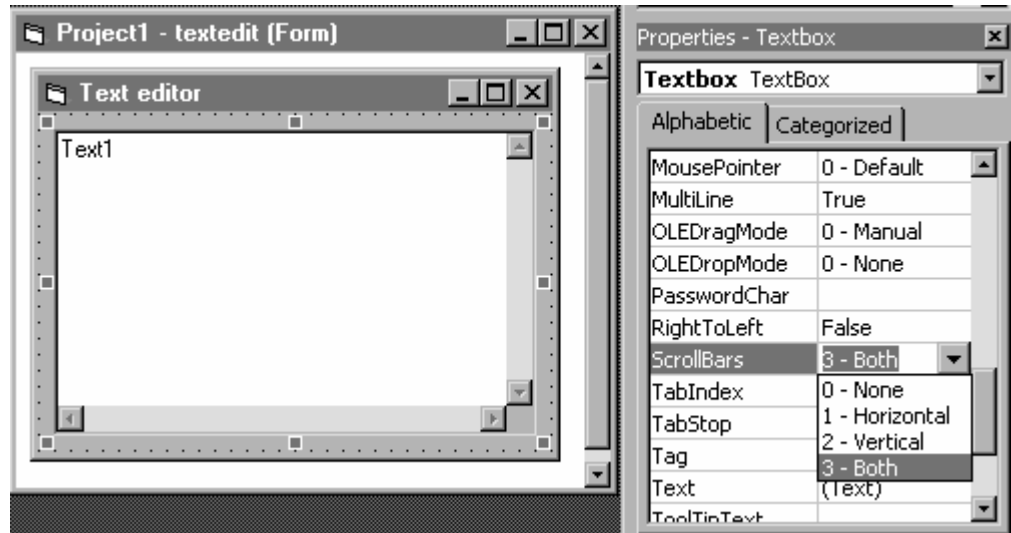
In this case study we will create a small text editor. Basic file manipulation , using the commondialog control and accessing the windows Clipboard will be explained.

Designing the user interface

As usual we first start visual basic and create a new Standard EXE project.

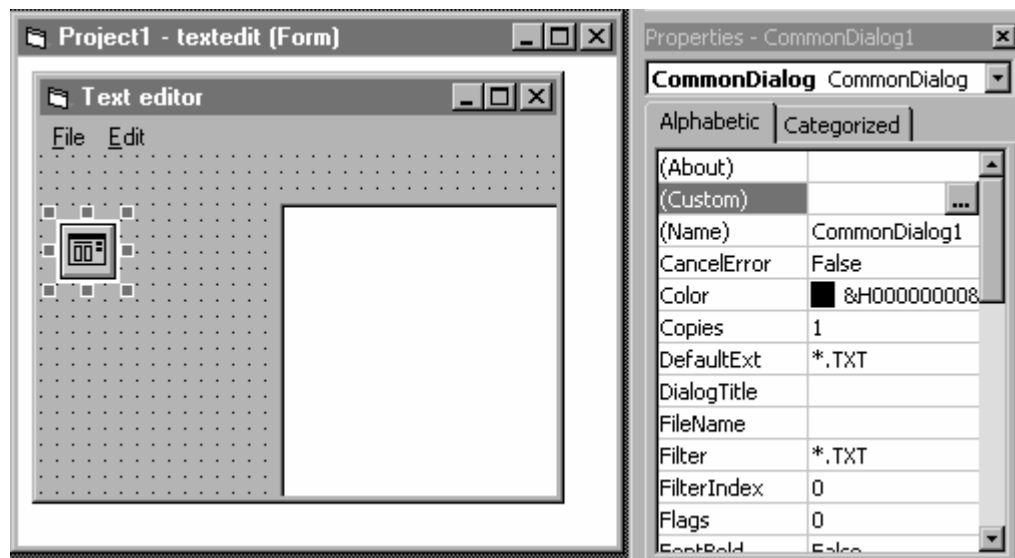


Since we will make a text editor the next logical step is to put a textbox on the form. If look to the properties of the textbox you will find something called 'multiline' . When you set this to true then the textbox can contain multiple lines of text. A CRLF will force the textbox to add a new line to its contents.



Since our edit can read files up to 32000 characters it might be a good idea if we would have some means to scroll through the text. Browsing through the properties quickly reveals the Scrollbars property. Setting this to Both displays both a horizontal and a vertical scrollbar.

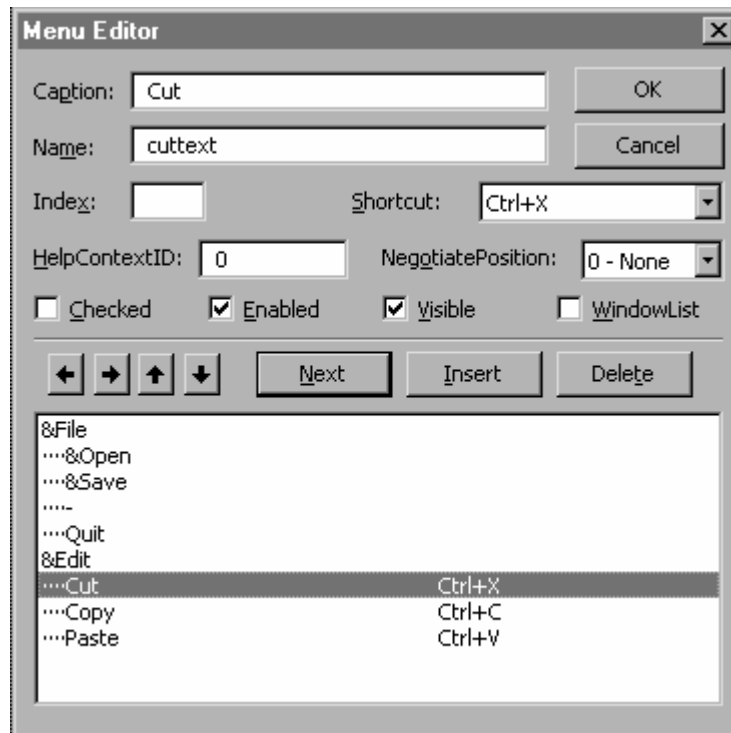
The next thing we should do is giving the user a means to load and save files. We could go on and design our own load and save forms but, since this is visual Basic, this already exists.



Placing the custom control commondialog on the form gives you instant access to such things as loading and saving files , selecting colors , selecting printers etc.

We give this control also a name. The control position doesn't matter . This is a kind of control that has no GUI element attached to it. This means that when your code starts running nothing appears. The interface of the control appears only when you access some of its methods .

Now we should build a menu to allow the user to access the file load/save and also to edit some text.



To ease the editing work we attach the standard windows hotkey's to the controls for cut , copy and paste.

If you would run your program now you would see that you already could type some text in the textbox . But this is of no much use since you could not save or retrieve any document.

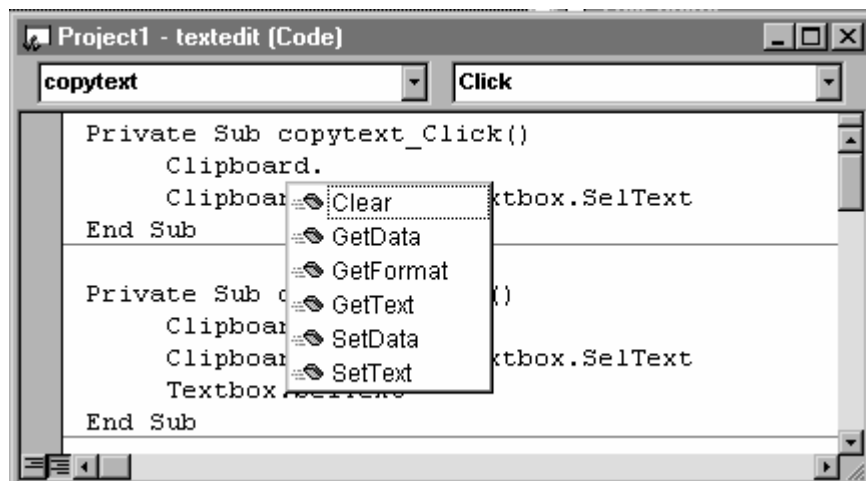
11.1.1 Attaching Code

The first thing to do is attach code to the Quit option on the menu. We will simply end the execution . Enhancements could be detecting if the user has not saved his work and display a warning that he might loose information.

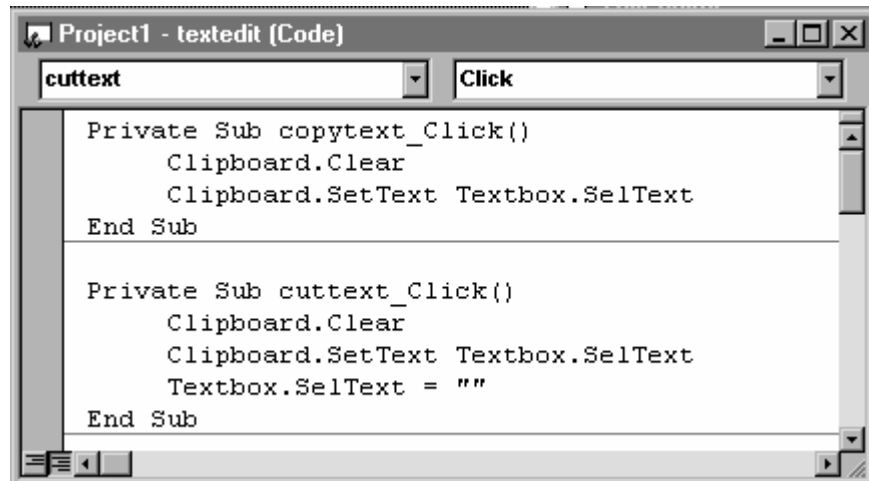
Now let's take a look into the Editing functions. The textbox control features a property called `SelectedText`. This property holds any text the user selects (highlighted text). You can read and write this property. This means that, when reading, you extract the text, and when writing, you change the selected text.

So all we need to do is store the contents of this property in a variable. Now we could do this but then we have to define a global variable etc. And what if we want to support copying and pasting across applications? Well windows has something called the 'clipboard'. We can use this clipboard from within visual basic. A virtual object called clipboard exists. This object need not to be put somewhere on your design form since it resides inside the operating system itself

To learn more about the clipboard object it suffices to type `clipboard` with a dot behind in the code window. VB will show you your options immediately.



As you can see the clipboard is a universal storage space for temporary data. You can store and retrieve texts, images and formatting commands with it.



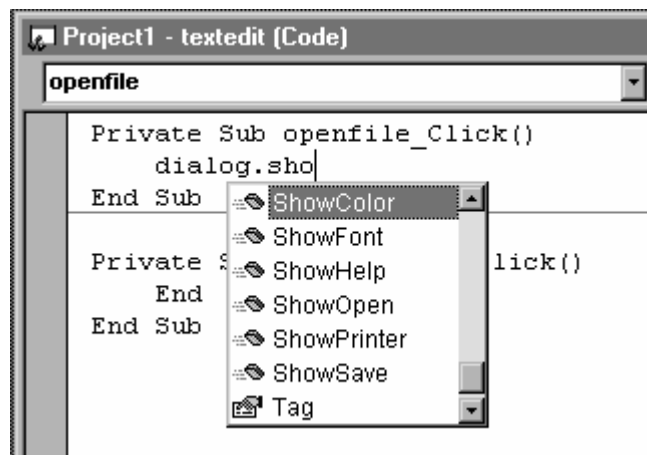
The copy routine clears the clipboard , then retrieves the text the user has selected and stores it onto the clipboard.

The Cut routine does exactly the same but afterwards sets the selected text to an empty string. This way the text disappears.

For the paste routine we only have to extract the stored text from the clipboard and dump it into the selected text. Now one nice thing about the Seltext property is that , when no text is selected , the text is dumped at the current cursor location. This means the user can insert text wherever he want by placing the cursor there , or overwriting text by selecting it.

Now that's done we can concentrate on loading and saving files. As explained before we will use the commondialog control to facilitate this operation.

The control has a several methods attached to it. The ones we are concentrating on are ShowOpen and ShowSave. Again when writing your code VB will assist you.



To ask the user for a filename you just call in the `commondialog.showopen`. The user can then navigate his hard disk and select a file. He can also cancel this operation

When a file has been successfully selected then the complete path and filename is stored in the property `Filename` of the `commondialog` control.

```
Private Sub openfile_Click()  
    dialog.ShowOpen          ' show the  
    commondialog  
    On Error GoTo invalidfile ' if no file should  
    be selected  
    filename$ = dialog.filename ' retrieve the  
    filename  
    Open filename$ For Input As #1 ' open the file  
    Textbox.Text = ""          ' clear contents of the  
    textbox  
    While Not EOF(1)           ' as long as not end of  
    file  
        Line Input #1, a$  
        Textbox.Text = Textbox.Text + a$ + vbCrLf  
    Wend  
  
invalidfile:  
    Close #1                   ' close  
    the file  
End Sub
```

When the user cancels the operation then no file has been selected and the property filename will be empty. In this case attempting to open a non-existing file will yield an error. Therefore we will test for errors during the execution of the code and takes measures to solve it.

Since a commondialog is always *appmodal* the user cannot do something else. He has to close the file selector first and then can continue to work.

The final code for the entire text editor looks like this :

```
Private sub QuitProgram_Click()  
    End  
End Sub  
  
Private sub savefile_Click()  
    dialog.ShowSave  
    On Error GoTo invalidfile  
    filename$ = dialog.filename  
    Open filename$ For output As #1  
    Print #1,Textbox.Text  
  
invalidfile:  
    Close #1  
End sub  
  
Private Sub openfile_Click()  
    dialog.ShowOpen          ' show the  
    commondialog  
    On Error GoTo invalidfile ' if no file should  
    be selected  
    filename$ = dialog.filename ' retrieve the  
    filename  
    Open filename$ For Input As #1 ' open the file  
    Textbox.Text = ""          ' clear contents of the  
    textbox  
    While Not EOF(1)          ' as long as not end of  
    file  
        Line Input #1, a$  
        Textbox.Text = Textbox.Text + a$ + vbCrLf  
    Wend  
  
invalidfile:  
    Close #1                  ' close  
    the file  
End Sub  
  
Private Sub Cuttext_Click()  
    Clipboard.Clear  
    Clipboard.Settext Textbox.Seltext  
    Textbox.Seltext=""  
End Sub
```

```
Private Sub CopyText_Click()  
    Clipboard.Clear  
    Clipboard.SetText Textbox.Seltext  
End Sub  
  
Private Sub PasteText_Click()  
    Textbox.Seltext=Clipboard.Gettext  
End Sub
```

Case Study 2 : A Calculator

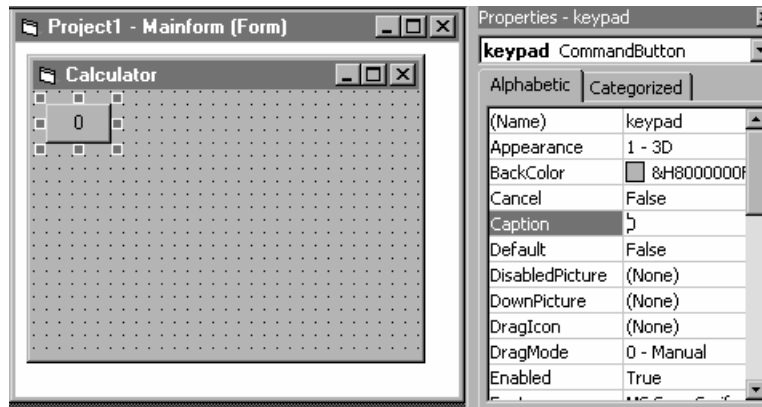


The work files for this project can also be found on the disk accompanying this manual.

The goal of this exercise is to show you how to create arrays of controls and how to create a project with multiple forms.

11.2.1 Designing the user interface

First of all you start up Visual basic and create a new standard exe project.



The main form is labeled 'calculator' and a commandbutton is created. The command button gets as caption '0' and as name Keypad.

The keypad we are about to design will be roughly divided in two sections. You will have the numerical field that will be designed as a control array and the other keys that are regular keys.

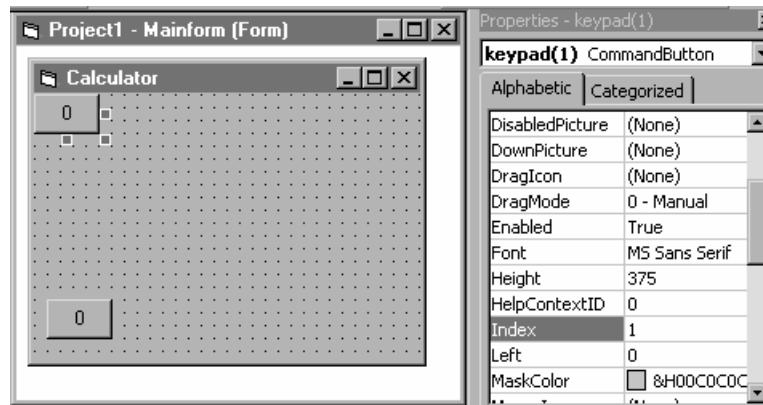
To create the array of objects you select the commandbutton and copy it (ctrl-c or copy on the edit menu)

Then you paste the object on the form by pressing ctrl-V or edit-Paste via the menubar.

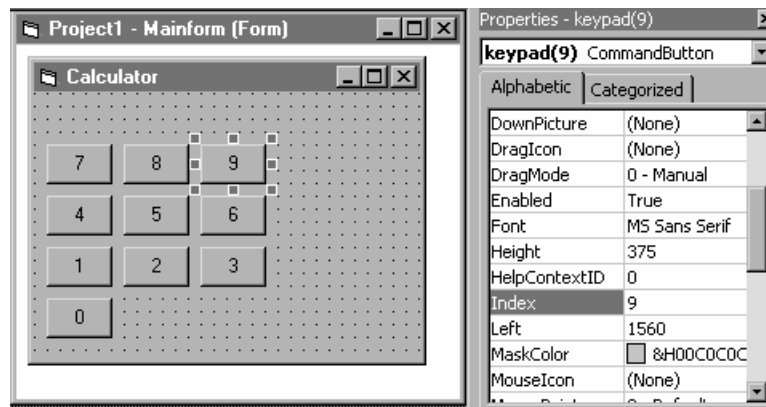
Visual basic will now ask you if you want to create a control array since you already have an object named 'keypad' on your form.



You click Yes and a new control will be placed on your form . This control is an exact copy of the original , except that it's *index* property has been set to 1.



You move the object now into place and select it. Change the caption property using the property navigator to '1'. Continue this until you have all the numbers from 0 to 9. The result should look like this:



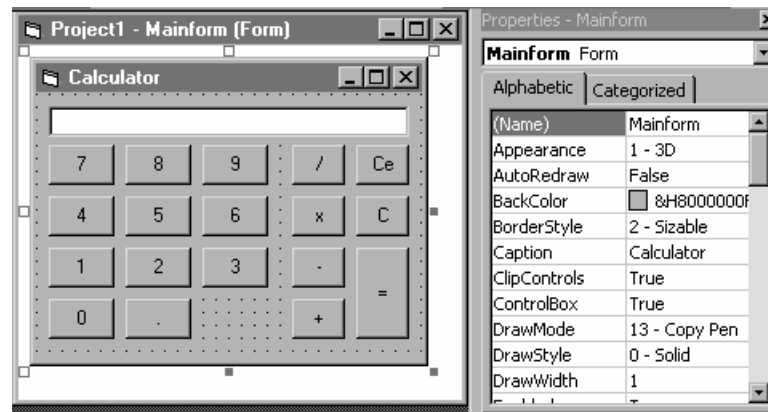
The last placed object will have index 9.

Now we can continue placing the other control buttons. The four operators will be called plus, minus, divide and multiply. A dot button will also be created. The next and last 3 buttons will be the CE , C and = button. These will allow you to correct errors and to actually execute the calculation.

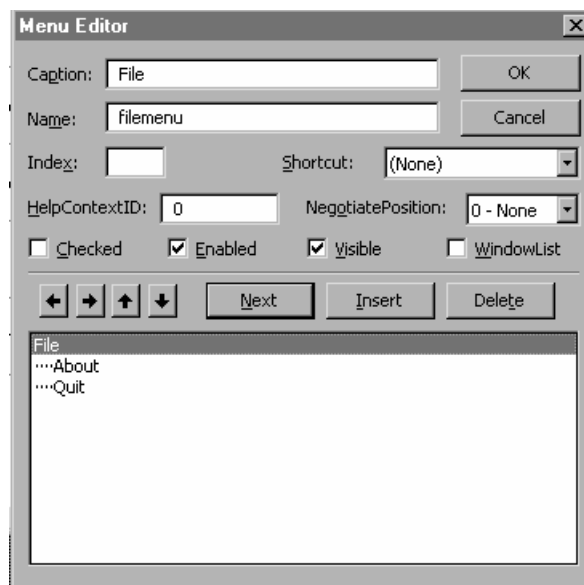
CE will be named clearerror , C will be named clearall and = is called calculate.

Finally a textbox is placed on the form and named 'display'.

The result should look somewhat like this:



The last thing we should do is creating a small menu that allows the user to exit the program. To do this you start the menu editor and build a small menu.



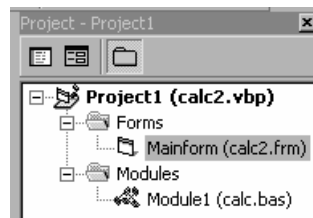
Now that we have everything in place we can start writing code for our application.

11.2.2 Writing Code

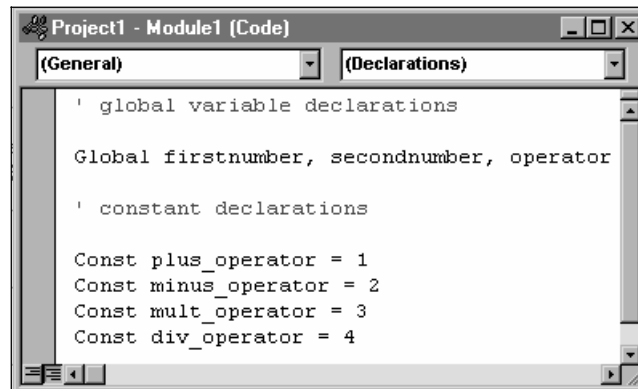
The first thing that needs to be done is deciding if we need any variables and or constants. If yes they should be stored in a module.

It might be a good idea to store entered values in two variables. Also the selected operator should be stored somewhere. To make the code readable we will define constants for the operators. This will allow us to refer to names instead of numbers. The code will be easier to understand later .

To do this you select the Project menu and click on ‘Add-Module” . In the project browser you will see that an empty module has been created and attached to your project.



In this module we will define the variables and constants discussed above.



The variables firstnumber, secondnumber and operator have been defined as global. This means that they can be accessed and modified from anywhere in the program. The same goes for the operators that are stored as global constants.!

As you type you will see again that the VB code editor will color the text. This gives you immediate feedback on the correct syntax of your code.

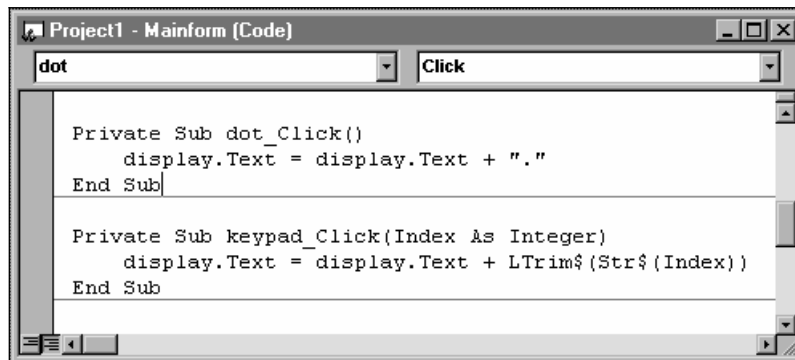
Now that this is done we can start creating code for our project.

11.2.3 Attaching code to the user interface

Let's start with the keypad. Double click any of the buttons of the keypad. The code editor will open up and show you the appropriate section of the program code.

As you can see the keypad click event returns an index to show you which one of the keys in the array of objects actually invoked the event. We will use this index value to update the contents of the display.

All we do is simply convert the number to a string using the STR\$ function. Since the STR\$ returns a string beginning with a leading space we strip off this space using the LTRIM\$ function



The dot operator will simply add a dot to the display. A point for improvement would be to check if there is already a dot and decide whether to put it or not. But this would take us too far from the basic programming course. Nah .. it won't. Since we are programming in Basic this is dead easy.

```
if instr ( display.text, "." ) = 0 then
    display.text = display.text + "."
end if
```

These two functions will allow you to enter a number using the mouse and the keypad.

So far you have written exactly 3 lines of code. Pretty neat huh ? So now lets create the code for the operators and other buttons. To do this simply double click any of the remaining controls and you will be able to attach the rest of the code. For the operators we will set the variable operator to one of our constants.

Later in the real calculation part we will use these constants again to decide what needs to be done.

The CE and C buttons do nothing else then clear the display (CE) and the variables holding the 2 numbers and the operator (C)

The code for the other buttons will look like this :

```
Private Sub clear_Click()
    display.Text = ""
End Sub
Private Sub clearerror_Click()
    display.Text = ""
    firstnumber = 0
    secondnumber = 0
    operator = 0
End Sub
Private Sub divide_Click()
    firstnumber = Val(display.Text)
    display.Text = ""
    operator = div_operator
```

```
End Sub
Private Sub minus_Click()
    firstnumber = Val(display.Text)
    display.Text = ""
    operator = minus_operator
End Sub
Private Sub multiply_Click()
    firstnumber = Val(display.Text)
    display.Text = ""
    operator = mult_operator
End Sub
Private Sub plus_Click()
    firstnumber = Val(display.Text)
    display.Text = ""
    operator = plus_operator
End Sub
Private Sub quitProgram_Click()
    End
End Sub
```

Now for the real workhorse. The = button . This button will actually calculate and display the result. This takes some programming logic to figure out exactly what to do and how to do it. Fortunately there is the Select Case construction that will help us out here


```
Private Sub calculate_Click()  
    secondnumber = Val(display.Text)  
    Select Case operator  
        Case plus_operator  
            result = firstnumber + secondnumber  
        Case minus_operator  
            result = firstnumber - secondnumber  
        Case mult_operator  
            result = firstnumber * secondnumber  
        Case div_operator  
            result = firstnumber / secondnumber  
    End Select  
    display.Text = result  
    firstnumber = result  
    secondnumber = 0  
    operator = 0  
End Sub
```

The last part of our work is to attach a few blurbs of code to the 2 menu items. The about item will simply brag a bit about our program and the Quit menu will neatly terminate our program.

When finished the complete code should look like this :

```
Private Sub aboutprogram_Click()  
    display.Text = "Calculator v1.0"  
End Sub  
Private Sub calculate_Click()  
    secondnumber = Val(display.Text)  
    Select Case operator  
        Case plus_operator  
            result = firstnumber + secondnumber  
        Case minus_operator  
            result = firstnumber - secondnumber  
        Case mult_operator  
            result = firstnumber * secondnumber  
        Case div_operator
```

```
        result = firstnumber / secondnumber
    End Select
    display.Text = result
    firstnumber = result
    secondnumber = 0
    operator = 0
End Sub
Private Sub clear_Click()
    display.Text = ""
End Sub
Private Sub clearerror_Click()
    display.Text = ""
    firstnumber = 0
    secondnumber = 0
    operator = 0
End Sub
Private Sub divide_Click()
    firstnumber = Val(display.Text)
    display.Text = ""
    operator = div_operator
End Sub
Private Sub dot_Click()
    display.Text = display.Text + "."
End Sub
Private Sub keypad_Click(Index As Integer)
    display.Text = display.Text +
LTrim$(Str$(Index))
End Sub
Private Sub minus_Click()
    firstnumber = Val(display.Text)
    display.Text = ""
    operator = minus_operator
End Sub
Private Sub multiply_Click()
    firstnumber = Val(display.Text)
    display.Text = ""
    operator = mult_operator
End Sub
```

```
Private Sub plus_Click()  
    firstnumber = Val(display.Text)  
    display.Text = ""  
    operator = plus_operator  
End Sub  
Private Sub quitProgram_Click()  
    End  
End Sub
```

Well that's it. We've just made a calculator with only 37 lines of code , that has a complete GUI and is entirely event driven.

Visual Basic

For Electronics Engineering Applications

Part II

*The Advanced World of Visual **B**asic*

Visual Basic

For Electronics Engineering Applications

Part II :

The Advanced World of Visual Basic

Introduction to Part II

Well hello. You made it this far. Hope you enjoyed the first part. In this part I'll take you further into the nuts and bolts of VB programming.

In the first part you have seen the fundamentals of Visual Basic programming. This course will build on the previous knowledge and dig deeper into the world of Windows and VB programming.

The main block of this part will explore the richness of the standard objects. Things like Popup menus , MDI forms , Menu lists ,and Timers and more will be explained. A section is dedicated to database manipulation and data access. I will also show you how to embed other applications inside your programs.

The rest of this part will dig deeper in Windows and the things it is composed of. The goal is to expose the inner workings of windows and of what use they can be to a programmer.. Topics such as API accessing , DLL accessing will be extensively covered.

The Last sections of this part will dig deeper into the computer. Topics such as serial communication , and WinSock operations will be explained.

Enjoy !

Chapter 12 :

One step beyond.

So far you have learned about objects, methods, properties and events. You have seen what they are and touched some of the things you can do with them. You also studied the command set of the Basic language. By this time you probably will have written a small program yourself. Now it is time to dig a little bit deeper in this new world ..

12.1 Forms

Lets take a closer look at the forms and what you can do with them. Typically a program will consist of multiple forms. You should already know that you can 'show' and 'hide' a form. You can also Load and unload a form. Now what is the real difference. The Form is typically the startup place of your program. Almost 100% of the applications have at least one form.

Typically the program begins with the form_load code of the main form. You do not have to call this yourself. The compiler defaults to this procedure upon executing the code. You can change this in the projects property dialog of Visual Basic.

12.2.1 Load

Loading a form means that windows allocated memory for the graphical part , the event processors , and the message queue. The moment a form is loaded it starts consuming resources. The more controls there are on a form the more resources will be allocated and the more time windows has to divert to it even if the form is not being used.

When you execute the `Form.Show` command then windows checks if the referred form is already loaded. If not it loads it and then shows it. If you hide a form it only disappears from the screen. All the resources allocated to it still remain locked.

From the above the following should be observed : if you extensively use forms then you should take care to unload the unnecessary forms whenever you can. Don't simply hide them.

You don't need to use the `Load` statement with forms unless you want to load a form without displaying it. Any reference to a form automatically loads it if it's not already loaded. For example, the `Show` method loads a form before displaying it. Once the form is loaded, its properties and controls can be altered by the application, whether or not the form is actually visible. Under some circumstances, you may want to load all your forms during initialization and display them later as they're needed.

If you load a Form whose `MDIChild` property is set to `True` (in other words, the child form) before loading an `MDIForm`, the `MDIForm` is automatically loaded before the child form. `MDI` child forms cannot be hidden, and thus are immediately visible after the `Form_Load` event procedure ends.

The standard dialog boxes produced by Visual Basic functions such as `MsgBox` and `InputBox` do not need to be loaded, shown, or unloaded, but can simply be invoked directly.

12.2.2 Unload

Unloads a form or control from memory.

Unload object

The object placeholder is the name of a Form object or control array element to unload.

Unloading a form or control may be necessary or expedient in some cases where the memory used is needed for something else, or when you need to reset properties to their original values.

Before a form is unloaded, the Query_Unload event procedure occurs, followed by the Form_Unload event procedure. Setting the cancel argument to True in either of these events prevents the form from being unloaded. For MDIForm objects, the MDIForm object's Query_Unload event procedure occurs, followed by the Query_Unload event procedure and Form_Unload event procedure for each MDI child form, and finally the MDIForm object's Form_Unload event procedure.

When a form is unloaded, all controls placed on the form at run time are no longer accessible. Controls placed on the form at design time remain intact; however, any run-time changes to those controls and their properties are lost when the form is reloaded. All changes to form properties are also lost. Accessing any controls on the form causes it to be reloaded.

Note:

When a form is unloaded, only the displayed component is unloaded. The code associated with the form module remains in memory.

Only control array elements added to a form at run time can be unloaded with the Unload statement. The properties of unloaded controls are reinitialized when the controls are reloaded.

12.2.3 Show

Shows a form or MDI form. If it is not loaded it will be loaded automatically

The Show method syntax has these parts:

Part	Importance	Description
Object	Optional.	An object expression that evaluates to an object in the Applies To list. If object is omitted, the form associated with the active form module is assumed to be object.
Style	Optional.	Integer that determines if the form is modal or modeless. If style is 0, the form is modeless; if style is 1, the form is modal.
Ownerform	Optional.	A string expression that specifies the component that "owns" the form being shown. For standard Visual Basic forms, use the keyword Me

If the specified form isn't loaded when the Show method is invoked, Visual Basic automatically loads it. When Show displays a modeless form, subsequent code is executed as it's encountered. When Show displays a modal form, no subsequent code is executed until the form is hidden or unloaded.

Note:

A form is bound to a certain mode. Typically a form is 'modeless' . this means that acts just as any other from . You can however force a certain 'mode'. You can stop windows until a particular form gets closed. This is explained in detail later on

When Show displays a modal form, no input (keyboard or mouse click) can occur except to objects on the modal form. The program must hide or unload a modal form (usually in response to some user action) before input to another form can occur. An MDIForm can't be modal.

Although other forms in your application are disabled when a modal form is displayed, other applications aren't. The startup form of an application is automatically shown after its Load event is invoked. Here is an example of how the Ownerform argument is used with the Show method:

```
Private Sub cmdShowResults_Click()  
    ' Show a modal form named frmResults.  
    frmResults.Show vbModal, Me  
End Sub
```

12.2.4 Hide.

Hides an MDIForm or Form object but doesn't unload it.

`object.Hide`

If object is omitted, the form with the focus is assumed to be object.

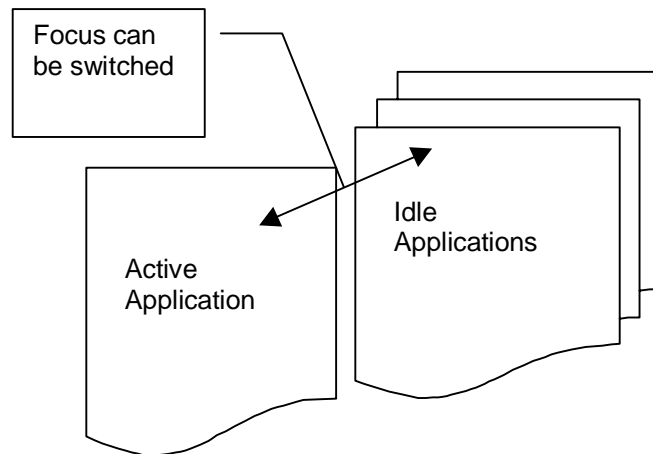
When a form is hidden, it's removed from the screen and its Visible property is set to False. A hidden form's controls aren't accessible to the user, but they are available to the running Visual Basic application, to other processes that may be communicating with the application through DDE, and to Timer control events.

When a form is hidden, the user can't interact with the application until all code in the event procedure that caused the form to be hidden has finished executing.

If the form isn't loaded when the Hide method is invoked, the Hide method loads the form but doesn't display it.

12.2.5 Modal / Modeless forms

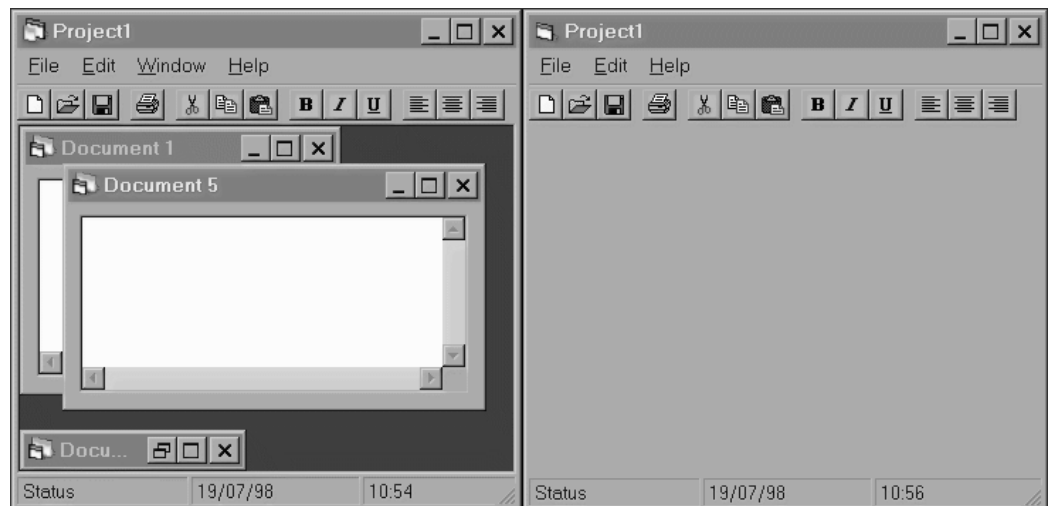
In generic a form is Modeless. This means that it is just a window on the GUI. If you want to create You can also create a Modal form. When a form is Modal this means that it has the focus for input. All other forms belonging to the same project are disabled. You can use this to notify the user of something and waiting for a response. The user cannot deny the information since the program stalls until he does something. Visual Basic supports the 2 types of Modal forms. VbAppModal means that the other windows of the application are



disabled . vbSystemModal means that all applications are disabled until the form is hidden.

12.2.6 MDI forms

Besides the standard look you can also create interfaces in what is called the Multiple Document Interface or MDI.



The left part shows an MDI interface. The right part shows a standard interface. Creating a program that does handle MDI is a bit more complex than a normal program. Fortunately VB has a wizard that enables you to build an MDI program very fast. To access this simply start the VB program wizard (File - New application and select VB application wizard. In the first form specify a MDI interface. The wizard will generate all necessary stuff for you.

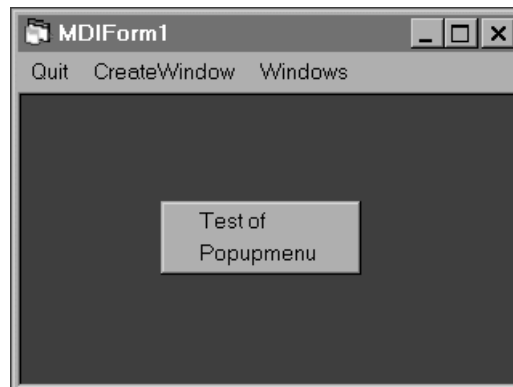
12.2 Menu's

Let's talk a bit more about menus. You have seen how to create them and how to assign hotkeys to them. There is more stuff you can do with menus. Creating a popup menu for instance, or adding items at runtime.

12.2.1 Popup menu's



I'm sure you have seen a lot of programs that have popup menus. Generally when you click with the right mouse button a menu of some sort pops up at the current cursor location.



Well you can do this in Visual basic too.

```
object.PopupMenu menuname, flags, x, y,  
boldcommand
```

The PopupMenu method syntax has these parts:

Part	importance	Description
object	Optional.	An object expression that evaluates to

		an object in the Applies To list. If object is omitted, the form with the focus is assumed to be object.
Menuname	Required.	The name of the pop-up menu to be displayed. The specified menu must have at least one submenu.
Flags	Optional.	A value or constant that specifies the location and behavior of a pop-up menu, as described in Settings.
X	Optional.	Specifies the x-coordinate where the pop-up menu is displayed. If omitted, the mouse coordinate is used.
Y	Optional.	Specifies the y-coordinate where the pop-up menu is displayed. If omitted, the mouse coordinate is used.
Boldcommand	Optional.	Specifies the name of a menu control in the pop-up menu to display its caption in bold text. If omitted, no controls in the pop-up menu appear in bold.

The settings for flags are:

Constant (location)	Value	Description
vbPopupMenuLeftAlign	0	(Default) The left side of the pop-up menu is located at x.
vbPopupMenuCenterAlign	4	The pop-up menu is centered at x.
vbPopupMenuRightAlign	8	The right side of the pop-up menu is located at x.
Constant (behavior)	Value	Description
vbPopupMenuLeftButton	0	(Default) An item on the pop-up menu reacts to a mouse click only when you use the left mouse button.
vbPopupMenuRightButton	2	An item on the pop-up menu reacts to a mouse click when you use either the right or the left mouse button.

Note

The flags parameter has no effect on applications running under Microsoft Windows version 3.0 or earlier. To specify two flags, combine one constant from each group using the Or operator.

These constants are listed in the Visual Basic (VB) object library in the Object Browser.

You specify the unit of measure for the x and y coordinates using the ScaleMode property. The x and y coordinates define where the pop-up is displayed relative to the specified form. If the x and y coordinates aren't included, the pop-up menu is displayed at the current location of the mouse pointer.

When you display a pop-up menu, the code following the call to the popup menu method isn't executed until the user either chooses a command from the menu (in which case the code for that command's Click event is executed before the code following the PopupMenu statement) or cancels the menu. In addition, only one pop-up menu can be displayed at a time; therefore, calls to this method are ignored if a pop-up menu is already displayed or if a pull-down menu is open.

12.2.2 Adding images to menu's

You might have seen already menus that contain bitmaps. This is not directly possible from Visual basic (or from any other language for that matter). The problem is that you manually need to write code for this. You cannot simply hide this in some compiler option. You need to generate explicit program startup and exit code. The icons or bitmaps need to be unloaded and destroyed upon program exit.

However it can be done using API calls. API calls are explained in Part III. An example will be given there on how to do this.

12.3 Modifying menus from code

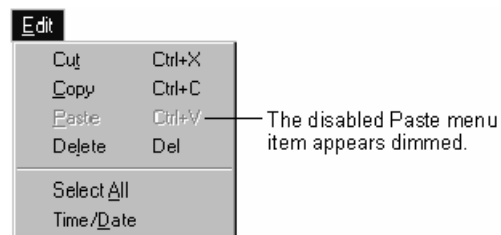
The menus you create at design time can also respond dynamically to run-time conditions. For example, if a menu item action becomes inappropriate at some point, you can prevent users from selecting that menu item by disabling it. In the MDI NotePad application, for example, if the clipboard doesn't contain any text, the Paste menu item is dimmed on the Edit menu, and users cannot select it.

You can also dynamically add menu items, if you have a menu control array. This is described in "Adding Menu Controls at Run Time," later in this topic.

You can also program your application to use a check mark to indicate which of several commands was last selected. For example, the Options, Toolbar menu item from the MDI NotePad application displays a check mark if the toolbar is displayed. Other menu control features described in this section include code that makes a menu item visible or invisible and that adds or deletes menu items.

12.3.1 Enabling and Disabling Menu Commands

All menu controls have an Enabled property, and when this property is set to False, the menu is disabled and does not respond to user actions. Shortcut key access is also disabled when Enabled is set to False. A disabled menu control appears dimmed.



For example, this statement disables the Paste menu item on the Edit menu of the MDI NotePad application:

```
mnuEditPaste.Enabled = False
```

Disabling a menu title in effect disables the entire menu, because the user cannot access any menu item without first clicking the menu title. For example, the following code would disable the Edit menu of the MDI Notepad application:

```
mnuEdit.Enabled = False
```

12.3.2 Displaying a Check Mark on a Menu Control

Using the Checked property, you can place a check mark on a menu to:

- Tell the user the status of an on/off condition. Choosing the menu command alternately adds and removes the check mark.
- Indicate which of several modes is in effect. The Options menu of the MDI Notepad application uses a check mark to indicate the state of the toolbar.



You create check marks in Visual Basic with the Checked property. Set the initial value of the Checked property in the Menu Editor by selecting the check box labeled Checked. To add or remove a check mark from a menu control at run time, set its Checked property from code. For example:

```
Private Sub mnuOptions_Click ()  
    ' Set the state of the check mark based on  
    ' the Visible property.  
    mnuOptionsToolbar.Checked =  
picToolbar.Visible  
End Sub
```

12.3.3 Making Menu Controls Invisible

In the Menu Editor, you set the initial value of the Visible property for a menu control by selecting the check box labeled Visible. To make a menu control visible or invisible at run time, set its Visible property from code. For example:

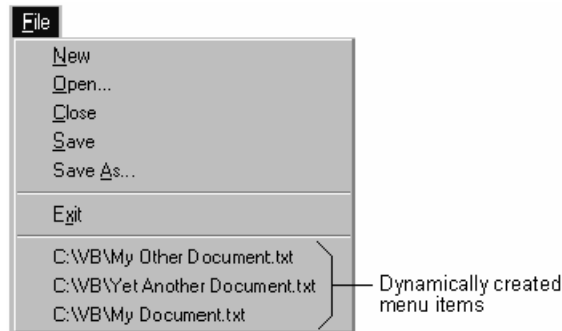
```
mnuFileArray(0).Visible = True    ' Make the  
control                          ' visible.  
mnuFileArray(0).Visible = False  ' Make the  
control                          ' invisible.
```

When a menu control is invisible, the rest of the controls in the menu move up to fill the empty space. If the control is on the menu bar, the rest of the controls on the menu bar move left to fill the space.

Note Making a menu control invisible effectively disables it, because the control is inaccessible from the menu, access or shortcut keys. If the menu title is invisible, all the controls on that menu are unavailable.

12.3.4 Adding Menu Controls at Run Time

A menu can grow at run time. In the image below, for example, as files are opened in the SDI NotePad application, menu items are dynamically created to display the path names of the most recently opened files.



You must use a control array to create a control at run time. Because the `mnuRecentFile` menu control is assigned a value for the `Index` property at design time, it automatically becomes an element of a control array — even though no other elements have yet been created.

When you create `mnuRecentFile(0)`, you actually create a separator bar that is invisible at run time. The first time a user saves a file at run time, the separator bar becomes visible, and the first file name is added to the menu. Each time you save a file at run time, additional menu controls are loaded into the array, making the menu grow.

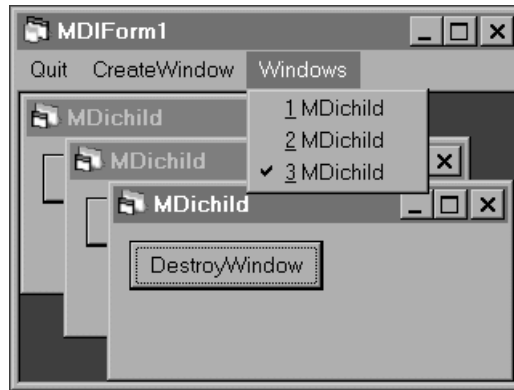
Controls created at run time can be hidden by using the `Hide` method or by setting the control's `Visible` property to `False`. If you want to remove a control in a control array from memory, use the `Unload` statement.

12.4 Special Menu features

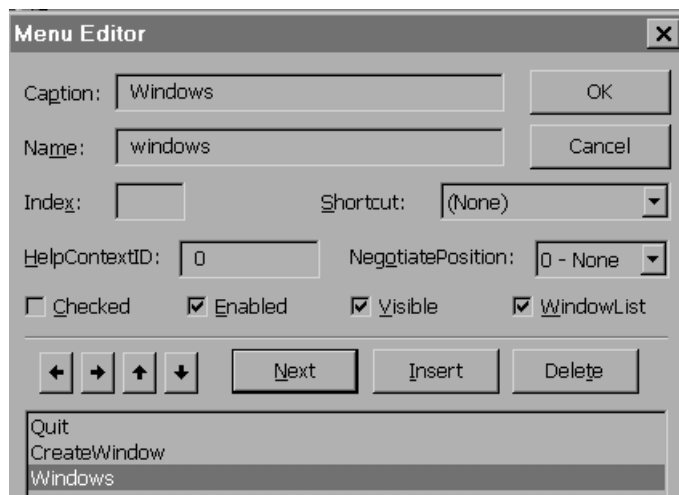
The menu system has a number of interesting features that can make life a lot easier for the programmer.

12.4.1 WindowList

A WindowList is a menu entry that automatically displays a list of available windows in your program. This is only useful if you are programming MDI interface. It allows the user to quickly jump from one window to another.



To add this to your application you simply select an entry in the menu editor and check the WindowList checkbox.



Note however that you can have only one WindowList in your menu bar.

12.4.2 Negotiating menu's

When programming MDI style programs you can use a feature called `NegotiateMenus`. You will find this property on any normal (non mdi-main) form. When this is set to true the menu of the child window will be displayed on the parent window.

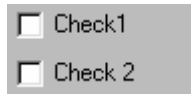


12.5 Option Selectors

Option selectors are simple visual components that allow you to specify certain selections or options. The simplest are Radio buttons and checkmarks. More advanced selectors let you select from a list (Combobox and Listbox). Finally I will show how to group selectors by using the Frame object.

Another type of selectors are listboxes. These allow the user to select an item from a predefined list. Or in case of the ComboBox he can also type in an existing item.

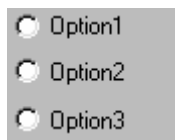
12.5.1 Checkboxes



The checkbox is one of the simplest selectors in Windows. It allows you to turn options on and off. A CheckBox control displays an X when selected; the X disappears when the CheckBox is cleared. Use this control to give the user a True/False or Yes/No option. You can use CheckBox controls in groups to display multiple choices from which the user can select one or more. To display text next to the CheckBox, set the Caption property. Use the Value property to determine the state of the control—selected, cleared, or unavailable.

You can also set the value of a CheckBox programmatically with the Value property. A value of 0 means it is not checked. A one means Checked and 2 means it is grayed out. If you set the value to 2 it will not respond to Click actions.

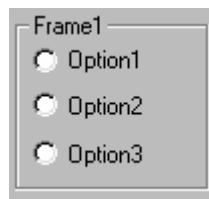
12.5.2 OptionButtons or Radio Buttons



An OptionButton control displays an option that can be turned on or off. Usually, OptionButton controls are used in an option group to display options from which the user selects only one.

Note :

CheckBox and OptionButton controls function similarly but with an important difference: Any number of CheckBox controls on a form can be selected at the same time. In contrast, only one OptionButton in a group can be selected at any given time.

12.5.3 Grouping Radio Buttons.

You group OptionButton controls by drawing them inside a container such as a Frame control, a PictureBox control, or a form. To group OptionButton controls in a Frame or PictureBox, draw the Frame or PictureBox first, and then draw the OptionButton controls inside. All OptionButton controls within the same container act as a single group.

12.5.4 Listboxes.

There are 2 standard windows listboxes you can use to allow the user to select something from a list. ListBox , FileListBox , DirListBox, DiskListbox are simple list boxes. ComboBox is a more versatile ListBox variant. This discussion will focus on the Combobox used as a plain listbox. For the full usage Combobox read on in the next chapter.



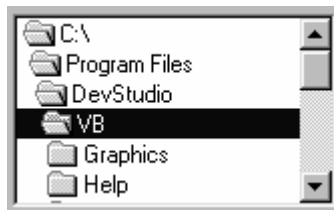
A `ListBox` control displays a list of items from which the user can select one or more. If the number of items exceeds the number that can be displayed, a scroll bar is automatically added to the `ListBox` control.

If no item is selected, the `ListIndex` property value is -1. The first item in the list is `ListIndex` 0, and the value of the `ListCount` property is always one more than the largest `ListIndex` value.

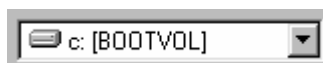
To add or delete items in a `ListBox` control, use the `AddItem` or `RemoveItem` method. Set the `List`, `ListCount`, and `ListIndex` properties to enable a user to access items in the `ListBox`. Alternatively, you can add items to the list by using the `List` property at design time.



A `FileListBox` control locates and lists files in the directory specified by the `Path` property at run time. Use this control to display a list of files selected by file type. You can create dialog boxes in your application that, for example, enable the user to select a file or group of files.



A `DirListBox` control displays directories and paths at run time. Use this control to display a hierarchical list of directories. You can create dialog boxes that, for example, enable a user to open a file from a list of files in all available directories.



A **DriveListBox** control enables a user to select a valid disk drive at run time. Use this control to display a list of all the valid drives in a user's system. You can create dialog boxes that enable the user to open a file from a list of files on a disk in any available drive.



A **ComboBox** control combines the features of a **TextBox** control and a **ListBox** control—users can enter information in the text box portion or select an item from the list box portion of the control.

To add or delete items in a **ComboBox** control, use the **AddItem** or **RemoveItem** method. Set the **List**, **ListCount**, and **ListIndex** properties to enable a user to access items in the **ComboBox**. Alternatively, you can add items to the list by using the **List** property at design time.

Note

A **Scroll** event will occur in a **ComboBox** control only when the contents of the dropdown portion of the **ComboBox** are scrolled, not each time the contents of the **ComboBox** change. For example, if the dropdown portion of a **ComboBox** contains five items and the top item is highlighted, a **Scroll** event will not occur until you press the down arrow six times (or the PGDN key once). After that, a **Scroll** event occurs for each press of the down arrow key. However, if you then press the up arrow key, a **Scroll** event will not occur until you press the up arrow key six times (or the PGUP key once). After that, each up arrow key press will result in a **Scroll** event.

The Style property settings for the ComboBox control are:

Constant	Value	Description
VbComboDropDown	0	(Default) Dropdown Combo. Includes a drop-down list and a text box. The user can select from the list or type in the text box.
VbComboSimple	1	Simple Combo. Includes a text box and a list, which doesn't drop down. The user can select from the list or type in the text box. The size of a Simple combo box includes both the edit and list portions. By default, a Simple combo box is sized so that none of the list is displayed. Increase the Height property to display more of the list.
VbComboDropDownList	2	Dropdown List. This style allows selection only from the drop-down list.

12.6 Timer objects

The Timer control allows you to generate timed events. There is no practical limitation to the amount of timers you can have running at the same time. The most important property of timer is the interval. It returns or sets the number of milliseconds between calls to a Timer control's Timer event.

object.Interval [= milliseconds]

The settings for milliseconds are:

Setting	Description
0	(Default) Disables a Timer control.
1 to 65,535	Sets an interval (in milliseconds) that takes effect when a Timer control's Enabled property is set to

True. For example, a value of 10,000 milliseconds equals 10 seconds. The maximum, 65,535 milliseconds, is equivalent to just over 1 minute.

You can set a Timer control's Interval property at design time or run time. When using the Interval property, remember:

The Timer control's Enabled property determines whether the control responds to the passage of time. Set Enabled to False to turn a Timer control off, and to True to turn it on. When a Timer control is enabled, its countdown always starts from the value of its Interval property setting.

12.7 User entry objects

12.7.1 Textboxes



Textboxes can be useful to allow the user to – fill in the blanks -. In some cases you want the user to see information but modify it only when a certain condition is met. The textbox has two interesting properties :

12.7.1.1 Locked and Enabled.

Locked means that the textbox remains as it is. It does not become grayed out. But the user cannot change the contents. No Change event is generated. You can change the contents from code. Enabled means that it will become grayed out. If this is the case you will not be able to change its contents , neither the user nor the program code.



Other interesting features are the Multiline and Scrollbar properties. By switching on Multiline you allow the user to type multiple lines of text. If you also cared to set the scrollbar property to anything else then none , then scrollbars of the selected style will automatically appear when the text no longer fits in the visible portions of the textbox. The user can then use these to walk through whatever input he made in the textbox.

12.7.1.2 Keypress Event

This is an event generated by every keypress when the textbox has the focus. It will return the ASCII key code. You can use this to make textboxes that behave in a particular way.



Sample :

```
Private Sub Text7_KeyPress(KeyAscii As Integer)  
    Static password$  
    If KeyAscii = 13 Then  
        MsgBox "Password :" + password$  
        Text7.Text = ""  
        password$ = ""  
    Else  
        password$ = password$ + Chr$(KeyAscii)  
        x = Len(Text7.Text)  
        Text7.Text = String(x, "*" )  
        KeyAscii = Asc ("*" )  
    End If  
End Sub
```

The above routine will react to any keypress. If enter (Carriage return = ASCII 13) is detected then a messagebox is displayed that shows the type password

If the character is not a CR then the character is added to the password. Finally A number of stars representing the length of the typed password are printed. But why on earth do I assign a star to the keyascii code ? Well simple. Reading the Value returned from this routine does not prevent it from getting sent to the textbox. Furthermore it will overwrite the first character in the textbox. So if I set it to an asterisk it will simply overwrite the first asterisk that was already there.

12.7.1 Combobox

I already explained the basics of a ComboBox when used as a simple listbox. However it goes far beyond that. The user can also type something . So when using combo-boxes you should retrieve the text property of the ComboBox. You can then match it against whatever is in the ComboBox. If it is not in there it means the user made a totally new selection. You can then decide to either reject it or maybe create something new.

A ComboBox control combines the features of a TextBox control and a ListBox control—users can enter information in the text box portion or select an item from the list box portion of the control.

To add or delete items in a ComboBox control, use the `AddItem` or `RemoveItem` method. Set the `List`, `ListCount`, and `ListIndex` properties to enable a user to access items in the ComboBox. Alternatively, you can add items to the list by using the `List` property at design time.

Note

A Scroll event will occur in a ComboBox control only when the contents of the dropdown portion of the ComboBox are scrolled, not each time the contents of the ComboBox change. For example, if the dropdown portion of a ComboBox contains five items and the top item is highlighted, a Scroll event will not occur until you press the down arrow six times (or the PGDN key once). After that, a Scroll event occurs for each press of the down arrow key. However, if you then press the up arrow key, a Scroll event will not occur until you press the up arrow key six times (or the PGUP key once). After that, each up arrow key press will result in a Scroll event.

By modifying the `Style` property settings for the ComboBox you can change its behavior. You will probably most of the time use it as Dropdown List. When styled to this mode it is easier to work with than the Listbox.

Possible settings are :

Constant	Value	Description
vbComboDropDown	0	(Default) Dropdown Combo. Includes a drop-down list and a text box. The user can select from the list or type in the text box.
vbComboSimple	1	Simple Combo. Includes a text box and a list, which doesn't drop down. The user can select from the list or type in the text box. The size of a Simple combo box includes both the edit and list portions. By default, a Simple combo box is sized so that none of the list is displayed. Increase the Height property to display more of the list.
vbComboDropDownList	2	Dropdown List. This style allows selection only from the drop-down list.

12.8 Printing

The Printer object enables you to communicate with a system printer (initially the default system printer). The Printers collection enables you to gather information about all the available printers on the system.

You can use graphics methods to draw text and graphics on the Printer object. Once the Printer object contains the output you want to print, you can use the EndDoc method to send the output directly to the default printer for the application.

You should check and possibly revise the layout of your forms if you print them. If you use the PrintForm method to print a form, for example, graphical images may be clipped at the bottom of the page and text carried over to the next page.

The Printers collection enables you to query the available printers so you can specify a default printer for your application. For example, you may want to find out which of the available printers uses a specific printer driver.

The following code searches all available printers to locate the first printer with its page orientation set to portrait, then sets it as the default printer:

```
Dim X As Printer  
For Each X In Printers  
    If X.Orientation = vbPRORPortrait Then  
        ' Set printer as system default.  
        Set Printer = X  
        ' Stop looking for a printer.  
        Exit For  
    End If  
Next
```

You designate one of the printers in the Printers collection as the default printer by using the Set statement. The preceding example designates the printer identified by the object variable X, the default printer for the application.

Note If you use the Printers collection to specify a particular printer, as in Printers(3), you can only access properties on a read-only basis. To both read and write the properties of an individual printer, you must first make that printer the default printer for the application.

12.9 Taking Advantage of the Windows95 Look

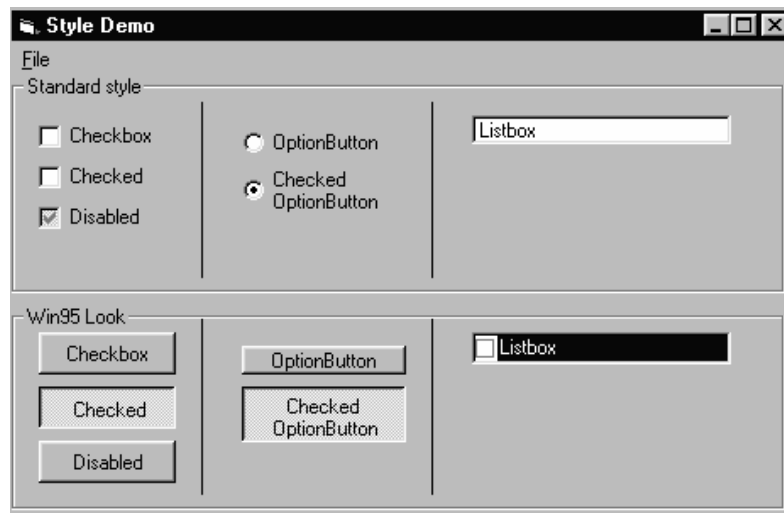


WinStyle.vbp

SO far I have shown you how to use the embedded controls that windows provides us. As you know Windows has a long history and the user interface has changed from time to time. The Windows 95 look brought some new graphical features. The controls available to us can often be switched from style.

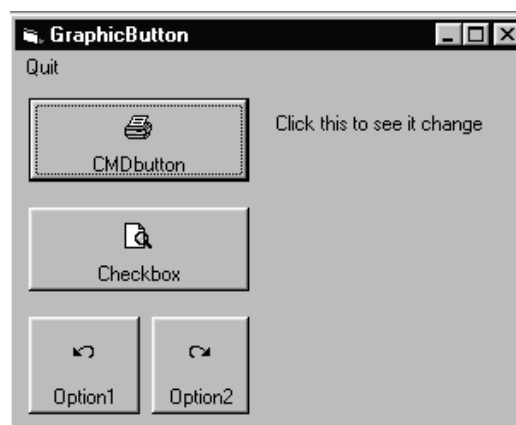
Optionbuttons , RadioButtons , Comboboxes and listboxes can either adopt the standard look or a newer Win95 look. In order to change the look there is a property called Style.

The look differs rather drastically



Changing the Checkbox and Optionbutton changes them to buttons that assume depressed or unpressed states. The listbox changes by adding a checkbox in front of an item. Using this you can make multiple selections (normally not possible with a listbox).

Other interesting options are the features that allow you to insert images on controls. This is a new thing that the Win95 GUI brings with it . For Win NT user : You need Release 4 or later.



You can assign a picture to the Down position , the Disabled and to the standard look. The standard picture (when nothing is happening or has happened with

the control) is set with the Picture property. The properties DownPicture and DisabledPicture allow you to set the image that should appear when they are selected and disabled.

Chapter 13 :

Graphics.

You can also create graphics using VB. Graphics can be drawn on almost any control. Visual Basic supports a set of methods that allow you to create drawings very easily. All drawing commands use an object called the Brush. This brush is an internal windows object that determines what drawing will look like. The brush is has a coordinate system and properties that specify color , style , fillstyle etc.

A number of basic statements are available to change these properties.

13.1 Basic coordinate operations

Drawing requires the use of a coordinate system of some kind. Windows handles the translation between your coordinate system and the physical display all by itself. The CurrentX and CurrentY parameters know the point where the next drawing will commence. You can also use these to set a new point.

13.1.1 *CurrentX, CurrentY*

Return or set the horizontal (*CurrentX*) or vertical (*CurrentY*) coordinates for the next printing or drawing method. Not available at design time.

```
object.CurrentX [= x]  
object.CurrentY [= y]
```

The *CurrentX* and *CurrentY* properties syntax have these parts:

Part	Description
Object	An object expression that evaluates to an object in the Applies To list.
X	A number that specifies the horizontal coordinate.
Y	A number that specifies the vertical coordinate.

Coordinates are measured from the upper-left corner of an object. The *CurrentX* property setting is 0 at an object's left edge, and the *CurrentY* property setting is 0 at its top edge. Coordinates are expressed in twips, or the current unit of measurement defined by the *ScaleHeight*, *ScaleWidth*, *ScaleLeft*, *ScaleTop*, and *ScaleMode* properties.

13.2 Drawing setup

13.2.1 *Drawwidth*

Returns or sets the line width for output from graphics methods.

```
object.Drawwidth [= size]
```

The DrawWidth property syntax has these parts:

Part	Description
Object	An object expression that evaluates to an object in the Applies To list.
Size	A numeric expression from 1 through 32,767. This value represents the width of the line in pixels. The default is 1; that is, 1 pixel wide.

Increase the value of this property to increase the width of the line. If the DrawWidth property setting is greater than 1, DrawStyle property settings 1 through 4 produce a solid line (the DrawStyle property value isn't changed). Setting DrawWidth to 1 allows DrawStyle to produce the results shown in the DrawStyle property table.

13.2.2 Drawmode

Returns or sets a value that determines the appearance of output from graphics method or the appearance of a **Shape** or **Line** controls.

object.DrawMode [= number]

The **DrawMode** property syntax has these parts:

Part	Description
Object	An object expression that evaluates to an object in the Applies To list.
Number	An integer that specifies appearance, as described in Settings.
Settings	

The settings for number are:

Constant	Setting	Description
VbBlackness	1	Blackness.
VbNotMergePen	2	Not Merge Pen — Inverse of setting 15 (Merge Pen).
VbMaskNotPen	3	Mask Not Pen — Combination of the colors common to the background color and the inverse of the pen.
VbNotCopyPen	4	Not Copy Pen — Inverse of setting 13 (Copy Pen).
VbMaskPenNot	5	Mask Pen Not — Combination of the colors common to both the pen and the inverse of the display.
VbInvert	6	Invert — Inverse of the display color.
VbXorPen	7	Xor Pen — Combination of the colors in the pen and in the display color, but not in both.
VbNotMaskPen	8	Not Mask Pen — Inverse of setting 9 (Mask Pen).
VbMaskPen	9	Mask Pen — Combination of the colors common to both the pen and the display.
VbNotXorPen	10	Not Xor Pen — Inverse of setting 7 (Xor Pen).
VbNop	11	Nop — No operation — output remains unchanged. In effect, this setting turns drawing off.
VbMergeNotPen	12	Merge Not Pen — Combination of the display color and the inverse of the pen color.
VbCopyPen	13	Copy Pen (Default) — Color specified by the ForeColor property.
VbMergePenNot	14	Merge Pen Not — Combination of the pen color and the inverse of the display color.
VbMergePen	15	Merge Pen — Combination of the pen color and the display color.
VbWhiteness	16	Whiteness.

Use this property to produce visual effects with Shape or Line controls or when drawing with the graphics methods. Visual Basic compares each pixel in the

draw pattern to the corresponding pixel in the existing background and then applies bit-wise operations. For example, setting 7 (Xor Pen) uses the Xor operator to combine a draw pattern pixel with a background pixel.

The exact effect of a DrawMode setting depends on the way the color of a line drawn at run time combines with colors already on the screen. Settings 1, 6, 7, 11, 13, and 16 yield the most predictable results.

13.2.3 DrawStyle

Returns or sets a value that determines the line style for output from graphics methods.

object.DrawStyle [= number]

The **DrawStyle** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
number	An integer that specifies line style, as described in Settings.

The settings for number are:

Constant	Setting	Description
vbSolid	0	(Default) Solid
vbDash	1	Dash
vbDot	2	Dot
vbDashDot	3	Dash-Dot
vbDashDotDot	4	Dash-Dot-Dot
vbInvisible	5	Transparent
vbInsideSolid	6	Inside Solid

If **DrawWidth** is set to a value greater than 1, **DrawStyle** settings 1 through 4 produce a solid line (the **DrawStyle** property value isn't changed). If **DrawWidth** is set to 1, **DrawStyle** produces the effect described in the preceding table for each setting.

13.2.4 Fillcolor

Returns or sets the color used to fill in shapes; **Fill Color** is also used to fill in circles and boxes created with the **Circle** and **Line** graphics methods.

object.FillColor [= value]

The **FillColor** property syntax has these parts:

Part	Description
Object	An object expression that evaluates to an object in the Applies To list.
Value	A value or constant that determines the fill color, as described in Settings.
Settings	

The settings for value are:

Setting	Description
Normal RGB colors	Colors set with the RGB or QBColor functions in code.
System default colors	Colors specified with the system color constants in the Visual Basic (VB) object library in the Object Browser. The Microsoft Windows operating environment substitutes the user's choices, as specified by the user's Control Panel settings.

By default, **FillColor** is set to 0 (Black). Except for the Form object, when the **FillStyle** property is set to its default, 1 (Transparent), the **FillColor** setting is ignored.

Note

As with all color settings you can still use the old DOS style colors by calling the Qbcolor function

13.2.5 FillStyle

Returns or sets the pattern used to fill Shape controls as well as circles and boxes created with the Circle and Line graphics methods.

object.FillStyle [= number]

The FillStyle property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
number	An integer that specifies the fill style, as described in Settings.

The number settings are:

Constant	Setting	Description
vbFSSolid	0	Solid
vbFSTransparent	1	(Default) Transparent
vbHorizontalLine	2	Horizontal Line
vbVerticalLine	3	Vertical Line
vbUpwardDiagonal	4	Upward Diagonal
VbDownwardDiagonal	5	Downward Diagonal
VbCross	6	Cross
VbDiagonalCross	7	Diagonal Cross

When **FillStyle** is set to 1 (Transparent), the **FillColor** property is ignored, except for the **Form** object.

13.3 Drawing primitives

13.3.1 PSet

Sets a point on an object to a specified color.

object.PSet [Step] (x, y), [color]

The **PSet** method syntax has the following object qualifier and parts:

Part	importance	Description
object	Optional.	Object expression that evaluates to an object in the Applies To list. If object is omitted, the Form with the focus is assumed to be object.
Step	Optional.	Keyword specifying that the coordinates are relative to the current graphics position given by the CurrentX and CurrentY properties.
(x, y)	Required.	Single values indicating the horizontal (x-axis) and vertical (y-axis) coordinates of the point to set.
color	Optional.	Long integer value indicating the RGB color specified for point. If omitted, the current ForeColor property setting is used. You can use the RGB function or QBColor function to specify the color.

The size of the point drawn depends on the setting of the DrawWidth property. When **DrawWidth** is 1, PSet sets a single pixel to the specified color. When DrawWidth is greater than 1, the point is centered on the specified coordinates.

The way the point is drawn depends on the setting of the **DrawMode** and **DrawStyle** properties. When **PSet** executes, the **CurrentX** and **CurrentY** properties are set to the point specified by the arguments.

To clear a single pixel with the **PSet** method, specify the coordinates of the pixel and use the **BackColor** property setting as the color argument.

13.3.2 Line

The **Line** command allows you to draw lines , boxes , filled boxes , shaded etc.

```
object.Line [Step] (x1, 1) [Step] (x2, y2),  
[color], [B][F]
```

The Line method is built of following parts :

Part	importance	Description
Object	Optional.	Object expression that evaluates to an object in the Applies To list. If object is omitted, the Form with the focus is assumed to be object.
Step	Optional.	Keyword specifying that the starting point coordinates is relative to the current graphics position given by the CurrentX and CurrentY properties.
(x1, y1)	Optional.	Single values indicating the coordinates of the starting point for the line or rectangle. The ScaleMode property determines the unit of measure used. If omitted, the line begins at the position indicated by CurrentX and CurrentY.
Step	Optional.	Keywords specifying that the end point coordinates are relative to the line starting point.
(x2, y2)	Required.	Single values indicating the coordinates of the end point for the line being drawn.
Color	Optional.	Long integer value indicating the RGB color used to draw the line. If omitted, the ForeColor property setting is used. You can use the RGB function or QBColor function to specify the color.
B	Optional.	If included, causes a box to be drawn using the coordinates to specify opposite corners of the box.
F	Optional.	If the B option is used, the F option specifies that the box is filled with the same color used to draw the box. You cannot use F without B. If B is used without F, the box is filled with the current FillColor and FillStyle. The default value for FillStyle is transparent.

To draw connected lines, begin a subsequent line at the end point of the previous line.

The width of the line drawn depends on the setting of the **DrawWidth** property. The way a line or box is drawn on the background depends on the setting of the **DrawMode** and **DrawStyle** properties. When **Line** executes, the **CurrentX** and **CurrentY** properties are set to the end point specified by the arguments.

13.3.3 Circle

This command allows you to create circles and ellipses.

```
object.Circle [Step](x,  
y),radius,[color,start,end, aspect]
```

The **Circle** method syntax has the following object qualifier and parts.

Part	importance	Description
object	Optional.	Object expression that evaluates to an object in the Applies To list. If object is omitted, the Form with the focus is assumed to be object.
Step	Optional.	Keyword specifying that the center of the circle, ellipse, or arc is relative to the current coordinates given by the CurrentX and CurrentY properties of object.
(x, y)	Required.	Single values indicating the coordinates for the center point of the circle, ellipse, or arc. The ScaleMode property of object determines the units of measure used.
radius	Required.	Single value indicating the radius of the circle, ellipse, or arc. The ScaleMode property of object determines the unit of measure used.
color	Optional.	Long integer value indicating the RGB color of the circle's outline. If omitted, the value of the ForeColor property is used. You can use the RGB function or QBColor function

		to specify the color.
start, end	Optional.	Single-precision values. When an arc or a partial circle or ellipse is drawn, start and end specify (in radians) the beginning and end positions of the arc. The range for both is -2 pi radians to 2 pi radians. The default value for start is 0 radians; the default for end is 2 * pi radians.
aspect	Optional.	Single-precision value indicating the aspect ratio of the circle. The default value is 1.0, which yields a perfect (non-elliptical) circle on any screen.

To fill a circle, set the **FillColor** and **FillStyle** properties of the object on which the circle or ellipse is drawn. Only a closed figure can be filled. Closed figures include circles, ellipses, or pie slices (arcs with radius lines drawn at both ends).

When drawing a partial circle or ellipse, if start is negative, **Circle** draws a radius to start, and treats the angle as positive; if end is negative, **Circle** draws a radius to end and treats the angle as positive. The **Circle** method always draws in a counter-clockwise (positive) direction.

The width of the line used to draw the circle, ellipse, or arc depends on the setting of the **DrawWidth** property. The way the circle is drawn on the background depends on the setting of the **DrawMode** and **DrawStyle** properties.

When drawing pie slices, to draw a radius to angle 0 (giving a horizontal line segment to the right), specify a very small negative value for start, rather than zero.

You can omit an argument in the middle of the syntax, but you must include the argument's comma before including the next argument. If you omit an optional argument, omit the comma following the last argument you specify.

When **Circle** executes, the **CurrentX** and **CurrentY** properties are set to the center point specified by the arguments.

13.4 Saving and loading graphics

Every time you need a graphic you could of course build it from scratch. You must be joking right ? There are functions that allow you to store and retrieve graphics from disk.

13.4.1 Saving Graphics

So you have created a nice graphic and would like to save it. Well nothing is simpler. Typically you use a PictureBox or an Image control to doodle on. But you can also use other objects to draw on. As long as an object has a Picture or Image property you can extract the graphical data from it. The problem is extracting this data and storing it in the appropriate format. To do this there is a function built into WINDOWS ! . After all the GUI system knows how to treat the graphics . Visual basic gives you direct access to this via the SavePicture procedure

SavePicture *picture, stringexpression*

Picture:	Picture or Image control from which the graphics file is to be created.
Stringexpression:	Filename of the graphics file to save.

If a graphic was loaded from a file to the Picture property of an object, either at design time or at run time, and it's a bitmap, icon, metafile, or enhanced metafile, it's saved using the same format as the original file. If it is a GIF or JPEG file, it is saved as a bitmap file.

Graphics in an Image property are always saved as bitmap (.bmp) files regardless of their original format. Any image that has been made with the drawing controls can be stored in this format.

Example

```
Private Sub Form_Click ()  
    ' Declare variables.  
    Dim CX, CY, Limit, Radius as Integer, Msg  
    as String  
    ScaleMode = vbPixels      ' Set scale to  
    pixels.  
    AutoRedraw = True ' Turn on AutoRedraw.  
    Width = Height      ' Change width to match  
    height.  
    CX = ScaleWidth / 2      ' Set X position.  
    CY = ScaleHeight / 2     ' Set Y position.  
    Limit = CX ' Limit size of circles.  
    For Radius = 0 To Limit ' Set radius.  
        Circle (CX, CY), Radius, RGB(Rnd *  
255, _  
                                Rnd * 255, Rnd * 255)  
        DoEvents      ' Yield for other processing.  
    Next Radius  
    Msg = "Choose OK to save the graphics from  
this form " Msg = Msg & "to a bitmap file."  
    MsgBox Msg  
    SavePicture Image, "TEST.BMP" ' Save  
    picture to file.  
End Sub
```

13.4.2 Loading Graphics

If you can save graphics it should be equally possible to load graphics. That's exactly what the LoadPicture is intended for. Any object supporting the Picture or Image property can be used as target for this operation

LoadPicture([stringexpression])

The stringexpression argument is the name of a graphics file to be loaded.

Graphics formats recognized by Visual Basic include bitmap (.bmp) files, icon (.ico) files, run-length encoded (.rle) files, metafile (.wmf) files, enhanced metafiles (.emf), GIF files, and JPEG (.jpg) files.

Graphics are cleared from forms, picture boxes, and image controls by assigning LoadPicture with no argument. To load graphics for display in a PictureBox control, Image control, or as the background of a form, the return value of LoadPicture must be assigned to the Picture property of the object on which the picture is displayed. For example:

```
Set Picture = LoadPicture( "PARTY.BMP" )  
Set Picture1.Picture = LoadPicture( "PARTY.BMP" )
```

To assign an icon to a form, set the return value of the LoadPicture function to the Icon property of the Form object:

```
Set Form1.Icon = LoadPicture( "MYICON.ICO" )
```

Icons can also be assigned to the DragIcon property of all controls except Timer controls and Menu controls. For example:

```
Set Command1.DragIcon =  
LoadPicture( "MYICON.ICO" )
```

Load a graphics file into the system Clipboard using LoadPicture as follows:

```
Clipboard.SetData LoadPicture( "PARTY.BMP" )
```

That's it. Very easy.

Example :

```
Private Sub Form_Click ( )  
    Dim MSG as String ' Declare variables.  
    On Error Resume Next ' Set up error  
handling.  
    Height = 3990  
    Width = 4890 ' Set height and width.  
    Set Picture = LoadPicture("PAPER.BMP")  
    ' Load bitmap.  
    If Err Then  
        MSG = "Couldn't find the .BMP file."  
        MsgBox MSG ' Display error message.  
        Exit Sub ' Quit if error occurs.  
    End If  
    MSG = "Choose OK to clear the bitmap from  
the form."  
    MsgBox MSG  
    Set Picture = LoadPicture( ) ' Clear  
form.  
  
End Sub
```

13.5 Coordinate systems



Bars.vbp

So far I have covered the basic drawing operations you can perform . When making drawing you always are doing this relative to a coordinate system. I have shown you how to specify where you want to draw in the coordinate system , but I have not explained you how to set it up. You can impose your own coordinate system using the Scale, ScaleMode, ScaleHeight , ScaleWidth , ScaleLeft and ScaleTop properties of the object you are drawing on.

13.5.1 Scale

Defines the coordinate system for a Form, PictureBox, or Printer. Doesn't support named arguments.

object.**Scale** (*x1*, *y1*) - (*x2*, *y2*)

Part	Description
object	Optional. An object expression that evaluates to an object in the Applies To list. If object is omitted, the Form object with the focus is assumed to be object.
x1, y1	Optional. Single-precision values indicating the horizontal (x-axis) and vertical (y-axis) coordinates that define the upper-left corner of object. Parentheses must enclose the values. If omitted, the second set of coordinates must also be omitted.
x2, y2	Optional. Single-precision values indicating the horizontal and vertical coordinates that define the lower-right corner of object. Parentheses must enclose the values. If omitted, the first set of coordinates must also be omitted.

The Scale method enables you to reset the coordinate system to any scale you choose. Scale affects the coordinate system for both run-time graphics statements and the placement of controls. If you use Scale with no arguments (both sets of coordinates omitted), it resets the coordinate system to twips.

You can specify scales as you please. You can also set the scale using the ScaleLeft, ScaleHeight, ScaleWidth and ScaleTop properties. Sometimes the plain Scale method is easier than setting each of these properties manually. It depends on what you want to do.

13.5.2 Scalemode

Returns or sets a value indicating the unit of measurement for coordinates of an object when using graphics methods or when positioning controls.

object.ScaleMode [= value]

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
value	An integer specifying the unit of measurement, as described in Settings.

The possible settings for value are:

Constant	Setting	Description
vbUser	0	Indicates that one or more of the ScaleHeight, ScaleWidth, ScaleLeft, and ScaleTop properties are set to custom values.
VbTwips	1	(Default) Twip (1440 twips per logical inch; 567 twips per logical centimeter).
VbPoints	2	Point (72 points per logical inch).
VbPixels	3	Pixel (smallest unit of monitor or printer resolution).
vbCharacters	4	Character (horizontal = 120 twips per unit; vertical = 240 twips per unit).
VbInches	5	Inch.
VbMillimeters	6	Millimeter.
VbCentimeters	7	Centimeter.

Using the related ScaleHeight, ScaleWidth, ScaleLeft, and ScaleTop properties, you can create a custom coordinate system with both positive and negative coordinates. These four Scale properties interact with the ScaleMode property in the following ways:

- Setting the value of any other Scale property to any value automatically sets ScaleMode to 0. A ScaleMode of 0 is user-defined.
- Setting the ScaleMode property to a number greater than 0 changes ScaleHeight and ScaleWidth to the new unit of measurement and sets

ScaleLeft and ScaleTop to 0. The CurrentX and CurrentY property settings change to reflect the new coordinates of the current point.

13.5.3 ScaleHeight , Scalewidth

Return or set the number of units for the horizontal (ScaleWidth) and vertical (ScaleHeight) measurement of the interior of an object when using graphics methods or when positioning controls. For MDIForm objects, not available at design time and read-only at run time.

```
object.ScaleHeight [= value]  
object.ScaleWidth [= value]
```

The ScaleHeight and ScaleWidth property syntaxes have these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
value	A numeric expression specifying the horizontal or vertical measurement.

You can use these properties to create a custom coordinate scale for drawing or printing. For example, the statement ScaleHeight = 100 changes the units of measure of the actual interior height of the form. Instead of the height being n current units (twips, pixels, ...), the height will be 100 user-defined units. Therefore, a distance of 50 units is half the height/width of the object, and a distance of 101 units will be off the object by 1 unit.

Use the ScaleMode property to define a scale based on a standard unit of measurement, such as twips, points, pixels, characters, inches, millimeters, or centimeters.

Setting these properties to positive values makes coordinates increase from top to bottom and left to right. Setting them to negative values makes coordinates increase from bottom to top and right to left.

Using these properties and the related `ScaleLeft` and `ScaleTop` properties, you can set up a full coordinate system with both positive and negative coordinates. All four of these `Scale` properties interact with the `ScaleMode` property in the following ways:

- Setting any other `Scale` property to any value automatically sets `ScaleMode` to 0. A `ScaleMode` of 0 is user-defined.
- Setting `ScaleMode` to a number greater than 0 changes `ScaleHeight` and `ScaleWidth` to the new unit of measurement and sets `ScaleLeft` and `ScaleTop` to 0. In addition, the `CurrentX` and `CurrentY` settings change to reflect the new coordinates of the current point.

You can also use the `Scale` method to set the `ScaleHeight`, `ScaleWidth`, `ScaleLeft`, and `ScaleTop` properties in one statement.

Note

The `ScaleHeight` and `ScaleWidth` properties aren't the same as the `Height` and `Width` properties.

For `MDIForm` objects, `ScaleHeight` and `ScaleWidth` refer only to the area not covered by `PictureBox` controls in the form. Avoid using these properties to size a `PictureBox` in the `Resize` event of an `MDIForm`.

13.5.4 `ScaleLeft` and `ScaleTop`

Return or set the horizontal (`ScaleLeft`) and vertical (`ScaleTop`) coordinates for the left and top edges of an object when using graphics methods or when positioning controls.

```
object.ScaleLeft [= value]  
object.ScaleTop [= value]
```

The `ScaleLeft` and `ScaleTop` property syntaxes have these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
value	A numeric expression specifying the horizontal or vertical coordinate. The default is 0.

Using these properties and the related ScaleHeight and ScaleWidth properties, you can set up a full coordinate system with both positive and negative coordinates. These four Scale properties interact with the ScaleMode property in the following ways:

- Setting any other Scale property to any value automatically sets ScaleMode to 0. A ScaleMode of 0 is user-defined.
- Setting the ScaleMode property to a number greater than 0 changes ScaleHeight and ScaleWidth to the new unit of measurement and sets ScaleLeft and ScaleTop to 0. The CurrentX and CurrentY property settings change to reflect the new coordinates of the current point.

You can also use the Scale method to set the ScaleHeight, ScaleWidth, ScaleLeft, and ScaleTop properties in one statement.

Note

The ScaleLeft and ScaleTop properties aren't the same as the Left and Top properties.

Chapter 14:

Communicating to the world around us

What good is a program if it cannot communicate. Well so far we have concentrated on communicating with the user of the program. Maybe it's time to have a look at what else we could possibly communicate with

14.1 SendKeys ...: a simple way of communicating

Yes... you can emulate sending keystrokes to another program. This is perhaps the simplest way of talking to other applications.. SendKeys can send one or more keystrokes to the active window as if typed at the keyboard.

SendKeys *string* [, *wait*]

String	Required. String expression specifying the keystrokes to send.
Wait	Optional. Boolean value specifying the wait mode. If False (default), control is returned to the procedure

	immediately after the keys are sent. If True, keystrokes must be processed before control is returned to the procedure.
--	---

One or more characters represent each key. To specify a single keyboard character, use the character itself. For example, to represent the letter A, use "A" for string. To represent more than one character, append each additional character to the one preceding it. To represent the letters A, B, and C, use "ABC" for string.

The plus sign (+), caret (^), percent sign (%), tilde (~), and parentheses () have special meanings to SendKeys. To specify one of these characters, enclose it within braces ({}). For example, to specify the plus sign, use {+}. Brackets ([]) have no special meaning to SendKeys, but you must enclose them in braces. In other applications, brackets do have a special meaning that may be significant when dynamic data exchange (DDE) occurs. To specify brace characters, use {{}} and {}.

To specify characters that aren't displayed when you press a key, such as ENTER or TAB, and keys that represent actions rather than characters, use the codes shown in the following table :

Key	Code
BACKSPACE	{BACKSPACE}, {BS}, or {BKSP}
BREAK	{BREAK}
CAPS LOCK	{CAPSLOCK}
DEL or DELETE	{DELETE} or {DEL}
DOWN ARROW	{DOWN}
END	{END}
ENTER	{ENTER} or ~
ESC	{ESC}

HELP	{HELP}
HOME	{HOME}
INS or INSERT	{INSERT} or {INS}
LEFT ARROW	{LEFT}
NUM LOCK	{NUMLOCK}
PAGE DOWN	{PGDN}
PAGE UP	{PGUP}
PRINT SCREEN	{PRTSC}
RIGHT ARROW	{RIGHT}
SCROLL LOCK	{SCROLLLOCK}
TAB	{TAB}
UP ARROW	{UP}
FUNCTION KEYS (F1.. F16)	{F1}{F2}{F3} ... etc

To specify keys combined with any combination of the SHIFT, CTRL, and ALT keys, precede the key code with one or more of the following codes:

Key	Code
SHIFT	+
CTRL	^
ALT	%

To specify that any combination of SHIFT, CTRL, and ALT should be held down while several other keys are pressed, enclose the code for those keys in parentheses. For example, to specify to hold down SHIFT while E and C are pressed, use "+(EC)". To specify to hold down SHIFT while E is pressed, followed by C without SHIFT, use "+EC".

To specify repeating keys, use the form {key number}. You must put a space between key and number. For example, {LEFT 42} means press the LEFT ARROW key 42 times; {h 10} means press H 10 times.

NOTE :

You can't use SendKeys to send keystrokes to an application that is not designed to run in Microsoft Windows. Sendkeys also can't send the PRINT SCREEN key {PRTSC} to any application.



Sendkeys.vbp

The following example uses the Shell function to run the Calculator application included with Microsoft Windows. It uses the SendKeys statement to send keystrokes to add some numbers, and then quit the Calculator. (To see the example, paste it into a procedure, then run the procedure. Because AppActivate changes the focus to the Calculator application, you can't single step through the code.)

```
Dim ReturnValue, I
ReturnValue = Shell("CALC.EXE", 1) ' Run
Calculator.
AppActivate ReturnValue          ' Activate the
Calculator.
For I = 1 To 100 ' Set up counting loop.
    SendKeys I & "{+}", True
    ' Send keystrokes to Calculator
Next I ' to add each value of I.
SendKeys "=", True ' Get grand total.
SendKeys "%{F4}", True ' Send ALT+F4 to close
Calculator.
```

14.1.1 AppActivate

Activates an application window.

AppActivate *title[, wait]*

The AppActivate statement syntax has these named arguments:

Part	Description
Title	Required. String expression specifying the title in the title bar of the application window you want to activate. The task ID returned by the Shell function can be used in place of title to activate an application.
Wait	Optional. Boolean value specifying whether the calling application has the focus before activating another. If False (default), the specified application is immediately activated, even if the calling application does not have the focus. If True, the calling application waits until it has the focus, then activates the specified application.

The AppActivate statement changes the focus to the named application or window but does not affect whether it is maximized or minimized. Focus moves from the activated application window when the user takes some action to change the focus or close the window. Use the Shell function to start an application and set the window style.

In determining which application to activate, title is compared to the title string of each running application. If there is no exact match, any application whose title string begins with title is activated. If there is more than one instance of the application named by title, one instance is arbitrarily activated.

14.1.2 Shell

Runs an executable program and returns a Variant (Double) representing the program's task ID if successful, otherwise it returns zero.

Shell(*pathname*[,*windowstyle*])

pathname	Required; Variant (String). Name of the program to execute and any required arguments or command-line switches; may include directory or folder and drive.
windowstyle	Optional. Variant (Integer) corresponding to the style of the window in which the program is to be run. If windowstyle is omitted, the program is started minimized with focus.

The windowstyle named argument can have these values:

vbHide	0	Window is hidden and focus is passed to the hidden window.
vbNormalFocus	1	Window has focus and is restored to its original size and position.
vbMinimizedFocus	2	Window is displayed as an icon with focus.
vbMaximizedFocus	3	Window is maximized with focus.
vbNormalNoFocus	4	Window is restored to its most recent size and position. The currently active window remains active.
vbMinimizedNoFocus	6	Window is displayed as an icon. The currently active window remains active.

If the Shell function successfully executes the named file, it returns the task ID of the started program. The task ID is a unique number that identifies the running program. If the Shell function can't start the named program, an error occurs.

Note

The Shell function runs other programs asynchronously. This means that a program started with Shell might not finish executing before the statements following the Shell function are executed.

14.2 DDE : another means of inter-program communication

DDE ? Never heard of ! . Well this is one of the cornerstones of Windows and Windows programming. It allows you to send information across programs.

You can actually write programs that talk to each other. This is an interesting feature that allows you to communicate , not only data , but also commands. DDE can be done with virtually any windows application. Most programs support it in one way or another. The only program is finding out the command for that particular program.

Communicating over DDE is done via a number of commands.

First of all we need to set up the communication. This requires specifying the application and target of the DDE conversation.

14.2.1 *LinkMode*:

Returns or sets the type of link used for a DDE conversation and activates the connection as follows:

Control	Allows a destination control on a Visual Basic form to initiate a conversation, as specified by the control's LinkTopic and LinkItem properties.
Form	Allows a destination application to initiate a conversation with a Visual Basic source form, as specified by the destination application's application!topic!item expression.

object.LinkMode [= number]

The LinkMode property syntax has these parts:

object	An object expression that evaluates to an object in the Applies To list.
number	An integer that specifies the type of connection, as described in Settings.

For controls used as destinations in DDE conversations, the settings for number are:

Constant	Setting	Description
vbLinkNone	0	(Default) None — No DDE interaction.
vbLinkAutomatic	1	Automatic — Destination control is updated each time the linked data changes.
vbLinkManual	2	Manual — {Destination control is updated only when the LinkRequest method is invoked.
vbLinkNotify	3	Notify — A LinkNotify event occurs whenever the linked data changes, but the destination control is updated only

		when the LinkRequest method is invoked.
--	--	---

For forms used as sources in DDE conversations, the settings for number are:

Constant	Setting	Description
vbLinkNone	0	(Default) None — No DDE interaction. No destination application can initiate a conversation with the source form as the topic, and no application can poke data to the form. If LinkMode is 0 (None) at design time, you can't change it to 1 (Source) at run time.
vbLinkSource	1	Source — Allows any Label, PictureBox, or TextBox control on a form to supply data to any destination application that establishes a DDE conversation with the form. If such a link exists, Visual Basic automatically notifies the destination whenever the contents of a control are changed. In addition, a destination application can poke data to any Label, PictureBox, or TextBox control on the form. If LinkMode is 1 (Source) at design time, you can change it to 0 (None) and back at run time.

The following conditions also apply to the LinkMode property:

Setting LinkMode to a nonzero value for a destination control causes Visual Basic to attempt to initiate the conversation specified in the LinkTopic and LinkItem properties. The source updates the destination control according to the type of link specified (automatic, manual, or notify).

If a source application terminates a conversation with a Visual Basic destination control, the value for that control's LinkMode setting changes to 0 (None).

If you leave `LinkMode` for a form set to the default 0 (None) at design time, you can't change `LinkMode` at run time. If you want a form to act as a source, you must set `LinkMode` to 1 (Source) at design time. You can then change the value of `LinkMode` at run time.

Note

Setting a permanent data link at design time with the Paste Link command from the Edit menu also sets the `LinkMode`, `LinkTopic`, and `LinkItem` properties. This creates a link that is saved with the form. Each time the form is loaded, Visual Basic attempts to re-establish the conversation.

14.2.2 *Linktopic*

For a destination control :

returns or sets the source application and the topic (the fundamental data grouping used in that application). Use `LinkTopic` with the `LinkItem` property to specify the complete data link.

For a source form:

returns or sets the topic that the source form responds to in a DDE conversation.

object.LinkTopic [= value]

The `LinkTopic` property syntax has these parts:

object	An object expression that evaluates to an object in the Applies To list.
--------	--

value	A string expression specifying a DDE syntax element.
-------	--

The LinkTopic property consists of a string that supplies part of the information necessary to set up either a destination link or source link. The string you use depends on whether you're working with a destination control or a source form. Each string corresponds to one or more elements of standard DDE syntax, which include application, topic, and item.

Note

While the standard definition for a DDE link includes the application, topic, and item elements, the actual syntax used within applications for a destination link to a source application may vary slightly.

For example, within Microsoft Excel, you use the syntax:

application/topic!item

Within Microsoft Word for Windows, you use:

application topic item

Don't use the pipe character [|] or exclamation mark [!].

Within a Visual Basic application, you use:

application/topic

The exclamation mark for topic is implicit.

14.2.2.1 Destination Control

To set LinkTopic for a destination control, use a string with the syntax application|topic as follows:

- application is the name of the application from which data is requested, usually the executable filename without an extension — for example, Excel (for Microsoft Excel).
- The pipe character (|, or character code 124) separates the application from the topic.
- Topic is the fundamental data grouping used in the source application — for example, a worksheet in Microsoft Excel.

In addition, for a destination control only, you must set the related `LinkItem` property to specify the item element for the link. A cell reference, such as `R1C1`, corresponds to an item in a Microsoft Excel worksheet.

14.2.2.2 Source Form

To set `LinkTopic` for a source form, set value to an appropriate identifier for the form. A destination application uses this string as the topic argument when establishing a DDE link with the form. Although this string is all you need to set `LinkTopic` within Visual Basic for a source form, the destination application also needs to specify:

- The application element that the destination application uses, which is either the Visual Basic project filename without the `.vbp` extension (if you're running your application in the Visual Basic development environment) or the Visual Basic application filename without the `.exe` extension (if you're running your application as a stand-alone executable file). The `EXENAME` property of the `App` object provides this string in your Visual Basic code unless the user changed the filename. (`EXENAME` always returns the actual filename of the application on disk; DDE always uses the original name that was specified in the Project Properties dialog box.)
- The item element that the destination application uses, which corresponds to the `Name` property setting for the `Label`, `PictureBox`, or `TextBox` control on the source form.

The following syntax is an example of a valid reference from Microsoft Excel to a Visual Basic application acting as a source:

```
=VizBasicApplication/FormN!TextBox1
```

You could enter this reference for a destination cell in the Microsoft Excel formula bar.

To activate the data link set with `LinkTopic`, set the `LinkMode` property to the appropriate nonzero value to specify the type of link you want. As a general rule, set `LinkMode` after you set `LinkTopic`. For a destination control, changing `LinkTopic` breaks an existing link and terminates the DDE conversation. For a source form, changing `LinkTopic` breaks all destination links that are using that topic. For these reasons, always set the `LinkMode` property to 0 before changing `LinkTopic`. After changing `LinkTopic` for a destination control, you must set `LinkMode` to 1 (Automatic), 2 (Manual), or 3 (Notify) to establish a conversation with the new topic.

Note

Setting a permanent data link at design time with the Paste Link command on the Edit menu also sets the `LinkMode`, `LinkTopic`, and `LinkItem` properties. This creates a link that is saved with the form. Each time the form is loaded, Visual Basic attempts to reestablish the conversation.

14.2.3 *LinkItem*

Returns or sets the data passed to a destination control in a DDE conversation with another application.

```
object.LinkItem [= string]
```

object	An object expression that evaluates to an object in the Applies To list.
string	A string expression that specifies the data to be passed to the destination control.

This property corresponds to the item argument in the standard DDE syntax, with application, topic, and item as arguments. To set this property, specify a recognizable unit of data in an application as a reference — for example, a cell reference such as "R1C1" in Microsoft Excel.

Use LinkItem in combination with the LinkTopic property to specify the complete data link for a destination control to a source application. To activate this link, set the LinkMode property.

You set LinkItem only for a control used as a destination. When a Visual Basic form is a source in a DDE conversation, the name of any Label, PictureBox, or TextBox control on the form can be the item argument in the application|topic|item string used by the destination. For example, the following syntax represents a valid reference from Microsoft Excel to a Visual Basic application:

=VizBasicApplication/MyForm!TextBox1

You could enter the preceding syntax for a destination cell in the Microsoft Excel formula bar.

A DDE control can potentially act as destination and source simultaneously, causing an infinite loop if a destination-source pair is also a source-destination pair with itself. For instance, a TextBox control may be both a source (through its parent form) and destination of the same cell in Microsoft Excel. When data in a Visual Basic TextBox changes, sending data to Microsoft Excel, the cell in Microsoft Excel changes, sending the change to the TextBox, and so on, causing the loop.

To avoid such loops, use related but not identical items for destination-source and source-destination links in both directions between applications. For example, in Microsoft Excel, use related cells (precedents or dependents) to link

a worksheet with a Visual Basic control, avoiding use of a single item as both destination and source. Document any application|topic pairs you establish if you include a Paste Link command for run-time use.

Note

Setting a permanent data link at design time with the Paste Link command from the Edit menu also sets the LinkMode, LinkTopic, and LinkItem properties. This creates a link that is saved with the form. Each time the form is loaded, Visual Basic attempts to re-establish the conversation.

Example

```
Private Sub Form_Click ( )  
    Dim CurRow As String  
    Static Row ' Worksheet row number.  
    Row = Row + 1 ' Increment Row.  
    If Row = 1 Then ' First time only.  
        ' Make sure the link isn't active.  
        Text1.LinkMode = 0  
        ' Set the application name and topic  
        name.  
        Text1.LinkTopic = "Excel|Sheet1"  
        Text1.LinkItem = "R1C1" ' Set  
        LinkItem.  
        Text1.LinkMode = 1 ' Set LinkMode to  
        Automatic.  
    Else  
        ' Update the row in the data item.  
        CurRow = "R" & Row & "C1"  
        Text1.LinkItem = CurRow ' Set  
        LinkItem.  
    End If  
End Sub
```

In the example, each mouse click causes a cell in a Microsoft Excel worksheet to update the contents of a Visual Basic TextBox control. To try this example, start Microsoft Excel, open a new worksheet named Sheet1, and put some data in the first column. In Visual Basic, create a form with a TextBox control. Paste the code into the Declarations section, and then press F5 to run the program.

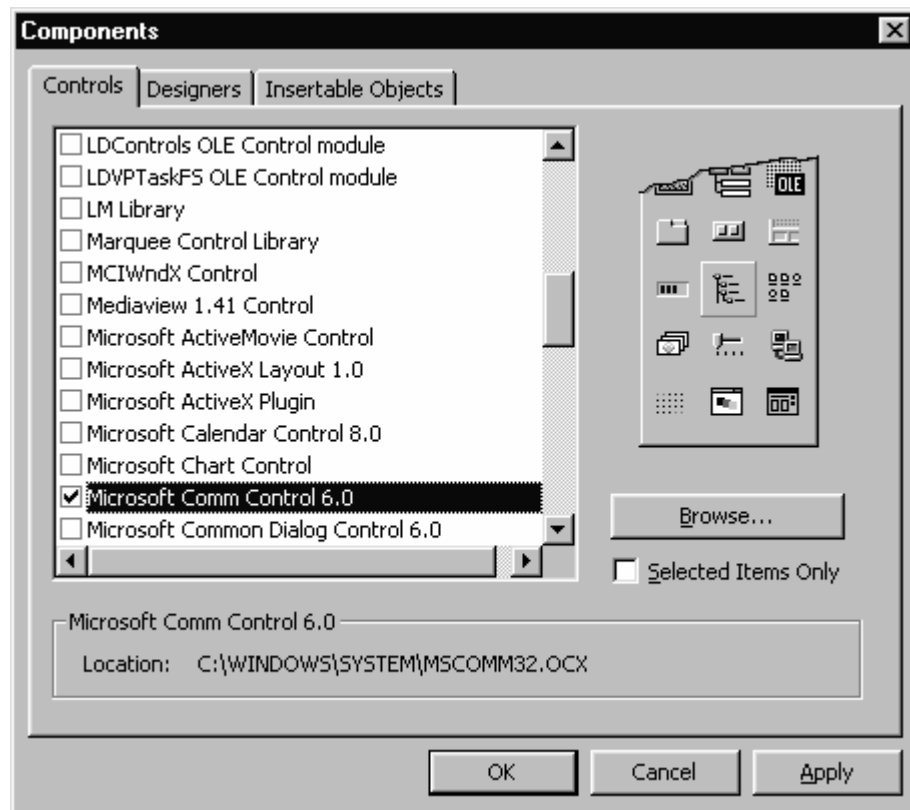
That's it for DDE. There are a lot more commands and events you can hook into. However if you want to learn more about this I can only suggest you to read a dedicated book about it. Check the suggested reading list for more information.

14.3 Serial IO : Talking to world beyond the port.

This is probably the most obscure part of windows. Most people still regard to this as a tricky thing. Well it isn't. Windows includes a standard object called MSCOMM that handles all the down-to-earth stuff for you. From programming the serial interface card to handshaking and exchanging data. All you have to do is set it up correctly

14.3.1 Inserting the object

First of all you should insert the MsComm object into your program.



To do this right click on the object browser and select Add-Component. Then browse for the MsComm Object. And make sure it is checked. It will then appear on the object browser.

The image of this object looks like this :



You can put up to 16 of these in a project (Windows directly supports up to 16 Serial ports) . If you need more ports you will have to install additional drivers into the operating system. Unfortunately there is no direct and easy way to detect the number of serial ports in a computer system. There are some API calls that allow you to extract the so-called system-metrics. But this leads us too far. There are dedicated books to API programming (Check the suggested reading list at the back of this book).

Simple communications can be set up in a snap. All we have to do is select the comport, specify operating parameters and open it. Done. So let's have a look at the most used parameters.

14.3.2 Portopen

Sets and returns the state of the communications port (open or closed). Not available at design time.

`object.PortOpen [= value]`

Where value is either True or False

Setting the PortOpen property to True opens the port. Setting it to False closes the port and clears the receive and transmit buffers. The MSComm control automatically closes the serial port when your application is terminated.

Make sure the CommPort property is set to a valid port number before opening the port. If the CommPort property is set to an invalid port number when you try to open the port, the MSComm control generates error 68 (Device unavailable).

In addition, your serial port device must support the current values in the Settings property. If the Settings property contains communications settings that your hardware does not support, your hardware may not work correctly.

If either the DTREnable or the RTSEnable properties is set to True before the port is opened, the properties are set to False when the port is closed. Otherwise, the DTR and RTS lines remain in their previous state.

Note

Never ever attempt to close or open a port two times in a row. This yields an error. You should first check the state of the port.

You can only change its state. You can't set it to the state it already has. You can make a construction like this

```

If comport.portopen=true then
    Comport.portopen=false
Else
    Comport.portopen=true
End if

```

14.3.3 Handshaking

Sets and returns the hardware handshaking protocol.

```
object.Handshaking [ = value ]
```

The Handshaking property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
value	An integer expression specifying the handshaking protocol, as described in Settings.

Setting	Value	Description
comNone	0	(Default) No handshaking.
comXOnXOff	1	XON/XOFF handshaking.
comRTS	2	RTS/CTS (Request To Send/Clear To Send) handshaking.
comRTSXOnXOff	3	Both Request To Send and XON/XOFF handshaking.

Handshaking refers to the internal communications protocol by which data is transferred from the hardware port to the receive buffer. When a character of data arrives at the serial port, the communications device has to move it into the receive buffer so that your program can read it. If there is no receive buffer and your program is expected to read every character directly from the hardware, you will probably lose data because the characters can arrive very quickly.

A handshaking protocol insures data is not lost due to a buffer overrun, where data arrives at the port too quickly for the communications device to move the data into the receive buffer.

Hardware (RTS/CTS) handshaking is done by the UART itself. But this requires the other side to have these capabilities as well. Furthermore this requires 2 extra wires in the serial cable. Simple devices such as microcontroller boards might not have these lines available.

Software handshaking is a protocol whereby 2 characters are reserved for handshaking. One character XON enables transmission and the other (XOFF) disables it. Whenever the slave is ready for data it sends XON to the master . If its input buffer is full it sends XOFF and processes the buffer.

14.3.4 Settings

Sets and returns the baud rate, parity, data bit, and stop bit parameters.

object.Settings [= value]

If value is not valid when the port is opened, the MSComm control generates error 380 (Invalid property value).

Value is composed of four settings and has the following format:

"BBBB,P,D,S"

Where BBBB is the baud rate, P is the parity, D is the number of data bits, and S is the number of stop bits. The default value of value is:

"9600,N,8,1"

The following table lists the valid settings.

BBBB Setting	P setting	Databits	Stopbits
--------------	-----------	----------	----------

110	E	Even	4	1
300	M	Mark	5	1.5
600	N	None	6	2
1200	O	Odd	7	
2400	S	Space	8 (Default)	
9600 (Default)				
14400				
19200				
28800				
38400				
(reserved)				
56000				
(reserved)				
128000				
(reserved)				
256000				
(reserved)				

Reserved means in this case that it depends on the speed of your computer if you can get there or not. IF you have a UART with a buffer inside you will be able to get these speeds. Else this setting will produce an error.

14.3.5 OutbufferSize , InbufferSize

Sets and returns the size, in bytes, of the transmit buffer.

object.OutBufferSize = number

OutBufferSize refers to the total size of the transmit buffer. The default size is 512 bytes. Do not confuse this property with the OutBufferCount which reflects the number of bytes currently waiting in the transmit buffer.

Note

The larger you make the transmit buffer, the less memory you have available to your application. However, if your buffer is too small, you run the risk of overflowing unless you use handshaking. As a general rule, start with a buffer size of 512 bytes. If an overflow error occurs, increase the buffer size to handle your application's transmission rate.

14.3.6 OutbufferCount, Inbuffercount

Returns the number of characters waiting in the transmit buffer. You can also use it to clear the transmit buffer. This property is not available at design time.

object.OutBufferCount [= value]

You can clear the transmit buffer by setting the OutBufferCount property to 0.

Note

Do not confuse the OutBufferCount property with the OutBufferSize property which reflects the total size of the transmit buffer.

14.3.7 Parityreplace

Sets and returns the character that replaces an invalid character in the data stream when a parity error occurs.

object.ParityReplace [= value]

Where value is a string expression representing a character, as described below.

The parity bit refers to a bit that is transmitted along with a specified number of data bits to provide a small amount of error checking. When you use a parity bit, the MSComm control adds up all the bits that are set (having a value of 1) in the data and tests the sum as being odd or even (according to the parity setting used when the port was opened).

By default, the control uses a question mark (?) character for replacing invalid characters. Setting ParityReplace to an empty string ("") disables replacement of

the character where the parity error occurs. The OnComm event is still fired and the CommEvent property is set to comEventRXParity.

The ParityReplace character is used in a byte-oriented operation, and must be a single-byte character. You can specify any ANSI character code with a value from 0 to 255.

14.3.8 DTREnable

Determines whether to enable the Data Terminal Ready (DTR) line during communications. Typically, the Data Terminal Ready signal is sent by a computer to its modem to indicate that the computer is ready to accept incoming transmission.

```
object.DTREnable[ = value ]
```

Where value is either True or False

When DTREnable is set to True, the Data Terminal Ready line is set to high (on) when the port is opened, and low (off) when the port is closed. When DTREnable is set to False, the Data Terminal Ready always remains low.

Note

When talking to a modem , in most cases setting the Data Terminal Ready line to low hangs up the telephone.

14.3.9 Rthreshold

Sets and returns the number of characters to receive before the MSComm control sets the CommEvent property to comEvReceive and generates the OnComm event.

object.Rthreshold [= value]

Setting the RThreshold property to 0 (the default) disables generating the OnComm event when characters are received. Setting RThreshold to 1, for example, causes the MSComm control to generate the OnComm event every time a single character is placed in the receive buffer.

14.3.10 OnComm Event

The OnComm event is generated whenever the value of the CommEvent property changes, indicating that either a communication event or an error occurred.

Private Sub object_OnComm ()

The CommEvent property contains the numeric code of the actual error or event that generated the OnComm event. Note that setting the RThreshold or SThreshold properties to 0 disables trapping for the comEvReceive and comEvSend events, respectively.

14.3.11 Commevent

Returns the most recent communication event or error. This property is not available at design time and is read-only at run time.

X = object.CommEvent

Although the OnComm event is generated whenever a communication error or event occurs, the CommEvent property holds the numeric code for that error or event. To determine the actual error or event that caused the OnComm event, you must reference the CommEvent property.

The CommEvent property returns one of the following values for communication errors or events. These constants can also be found in the Object Library for this control.

Communication errors include the following settings:

Constant	Value	Description
ComEventBreak	1001	A Break signal was received.
ComEventCTSTO	1002	Clear To Send Timeout. The Clear To Send line was low for the system specified amount of time while trying to transmit a character.
ComEventDSRTO	1003	Data Set Ready Timeout. The Data Set Ready line was low for the system specified amount of time while trying to transmit a character.
ComEventFrame	1004	Framing Error. The hardware detected a framing error.
ComEventOverrun	1006	Port Overrun. A character was not read from the hardware before the next character arrived and was lost.
ComEventCDTO	1007	Carrier Detect Timeout. The Carrier Detect line was low for the system specified amount of time while trying to transmit a character. Carrier Detect is also known as the Receive Line Signal Detect (RLSD).
ComEventRxOver	1008	Receive Buffer Overflow. There is no room in the receive buffer.
ComEventRxParity	1009	Parity Error. The hardware detected a parity error.
ComEventTxFull	1010	Transmit Buffer Full. The transmit buffer was full while trying to queue a character.
ComEventDCB	1011	Unexpected error retrieving Device Control Block (DCB) for the port.

Communications events include the following settings:

Constant	Value	Description
----------	-------	-------------

ComEvSend	1	There are fewer than Sthreshold number of characters in the transmit buffer.
ComEvReceive	2	Received Rthreshold number of characters. This event is generated continuously until you use the Input property to remove the data from the receive buffer.
ComEvCTS	3	Change in Clear To Send line.
ComEvDSR	4	Change in Data Set Ready line. This event is only fired when DSR changes from 1 to 0.
ComEvCD	5	Change in Carrier Detect line.
ComEvRing	6	Ring detected. Some UARTs (universal asynchronous receiver-transmitters) may not support this event.
ComEvEOF	7	End Of File (ASCII character 26) character received.

That's about it concerning serial communications. A simple program is available that allows you to send and receive character over a serial port. An in depth hardware approach to serial programming will be handle in Part IV of this Book.. The examples on how to make use of this powerful object will be explained there.

14.4 Winsock : The world is not enough ...

The Winsock control, invisible to the user, provides easy access to TCP and UDP network services. Microsoft Access, Visual Basic, Visual C++, or Visual FoxPro developers can use it. To write client or server applications you do not need to understand the details of TCP or to call low-level Winsock APIs. By setting properties and invoking methods of the control, you can easily connect to a remote machine and exchange data in both directions.

14.4.1 TCP Basics

The Transfer Control Protocol allows you to create and maintain a connection to a remote computer. Using the connection, both computers can stream data between themselves.

If you are creating a client application, you must know the server computer's name or IP address (RemoteHost property), as well as the port (RemotePort property) on which it will be "listening." Then invoke the Connect method.

If you are creating a server application, set a port (LocalPort property) on which to listen, and invoke the Listen method. When the client computer requests a connection, the ConnectionRequest event will occur. To complete the connection, invoke the Accept method within the ConnectionRequest event.

Once a connection has been made, either computer can send and receive data. To send data, invoke the SendData method. Whenever data is received, the DataArrival event occurs. Invoke the GetData method within the DataArrival event to retrieve the data.

14.4.2 UDP Basics

The User Datagram Protocol (UDP) is a connectionless protocol. Unlike TCP operations, computers do not establish a connection. Also, a UDP application can be either a client or a server.

To transmit data, first set the client computer's LocalPort property. The server computer then needs only to set the RemoteHost to the Internet address of the client computer, and the RemotePort property to the same port as the client computer's LocalPort property, and invoke the SendData method to begin sending messages. The client computer then uses the GetData method within the DataArrival event to retrieve the sent messages.

14.4.3 RemoteHost

This returns or sets the remote machine to which a control sends or receives data. You can either provide a host name, for example,

"FTP://ftp.microsoft.com," or an IP address string in dotted format, such as "100.0.1.1".

object.RemoteHost = string

When this property is specified, the URL property is updated to show the new value. Also, if the host portion of the URL is updated, this property is also updated to reflect the new value.

The RemoteHost property can also be changed when invoking the OpenURL or Execute methods.

At run time, changing this value has no effect until the next connection.

14.4.4 Protocol

Returns or sets the protocol, either TCP or UDP, used by the Winsock control.

object.Protocol [=protocol]

The possible settings for protocol are:

Constant	Value	Description
SckTCPProtocol	0	Default. TCP protocol.
SckUDPProtocol	1	UDP protocol.

The control must be closed (using the Close method) before this property can be reset.

14.4.5 State

Returns the state of the control, expressed as an enumerated type. Read-only and unavailable at design time.

X = object.State

The possible results for the State property are:

Constant	Value	Description
SckClosed	0	Default. Closed
SckOpen	1	Open
SckListening	2	Listening
SckConnectionPending	3	Connection pending
SckResolvingHost	4	Resolving host
SckHostResolved	5	Host resolved
SckConnecting	6	Connecting
SckConnected	7	Connected
SckClosing	8	Peer is closing the connection
SckError	9	Error

14.4.6 Accept

This is for TCP server applications only. This method is used to accept an incoming connection when handling a ConnectionRequest event.

object.Accept requestID

The Accept method is used in the ConnectionRequest event. The ConnectionRequest event has a corresponding argument, the RequestID parameter that should be passed to the Accept method. An example is shown below:

```

Private Sub Winsock1_ConnectionRequest _
  (ByVal requestID As Long)
  ' Close the connection if it is currently
  open
  ' by testing the State property.
  If Winsock1.State <> sckClosed Then
    Winsock1.Close

    ' Pass the value of the requestID
    parameter to the
    ' Accept method.
    Winsock1.Accept requestID
  End Sub

```

The Accept method should be used on a new control instance (other than the one that is in the listening state.)

14.4.7 GetData

Retrieves the current block of data and stores it in a variable of type variant.

object.GetData data, [type,] [maxLen]

Data	Where retrieved data will be stored after the method returns successfully. If there is not enough data available for requested type, data will be set to Empty.
Type	Optional. Type of data to be retrieved, as shown in Settings.
MaxLen	Optional. Specifies the desired size when receiving a byte array or a string. If this parameter is missing for byte array or string, all available data will be retrieved. If provided for data types other than byte array and string, this parameter is ignored.

Data can be returned in a type of your choice. This is accomplished by specifying this in the Type argument. The settings for type are:

Description	Constant
Byte	vbByte
Integer	vbInteger
Long	vbLong
Single	vbSingle
Double	vbDouble
Currency	vbCurrency
Date	vbDate
Boolean	vbBoolean
SCODE	vbError
String	vbString
Byte Array	vbArray + vbByte

It's common to use the GetData method with the DataArrival event, which includes the totalBytes argument. If you specify a maxlen that is less than the totalBytes argument, you will get the warning 10040 indicating that the remaining bytes will be lost.

14.4.8 Connectionrequest

Occurs when a remote machine requests a connection.

This is for TCP server applications only. The event is activated when there is an incoming connection request. RemoteHostIP and RemotePort properties store the information about the client after the event is activated.

object_ConnectionRequest (requestID As Long)

The ConnectionRequest event syntax has these parts:

The requestID parameter is the incoming connection request identifier. This argument should be passed to the Accept method on the second control instance.

The server can decide whether or not to accept the connection. If the incoming connection is not accepted, the peer (client) will get the Close event. Use the Accept method (on a new control instance) to accept an incoming connection.

14.4.9 DataArrival

Occurs when new data arrives.

*object_***DataArrival** (*bytesTotal As Long*)

This event will not occur if you do not retrieve all the data in one GetData call. It is activated only when there is new data. Use the BytesReceived property to check how much data is available at any time.

In the appendix of this part examples will be given on how to set up a little Telnet server. This is a program you can log into over the internet and can accept commands and send you data. You can use this to make a remote controlled system that can run over the Ethernet , intranet or even the internet.

Some more case studies

Doodle : A Graphics program

This program demonstrates the use of graphics manipulation routines. Doodle is a simple program that allows you to draw into a PictureBox. You can save the data to a standard BMP file. You can select to draw geometric shapes such as draw lines ,circles ,ellipses and rectangles , or you can draw freehand. It allows you to select colors for your drawing and specify filled or open shapes

Miniterm :A simple terminal

This program will allow us to communicate with the outside world in a simple fashion. A number of practical things are given that will show you , besides serial communication , how to implement some data manipulations.

AlphaServer : A Telnet Server application.

This program can be started on any machine that has a valid IP address , and is connected into a TCP/IP network (LAN / WAN / Intranet / Internet). You can log on to it and interrogate it. This could be a program that collects data on some remote site , and can be accessed and controlled via remote.

LoanCalc : Using Excel from your program

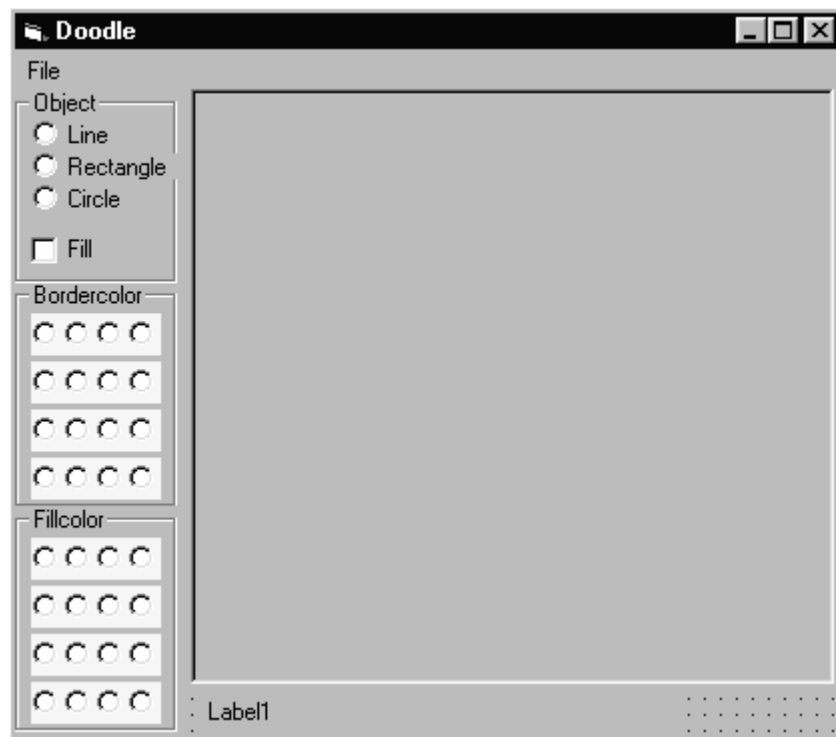
This example shows yet another way to control other programs. It will derive an object from the 'Excel' program and use it to perform some calculations. Actually it uses Excel as a 'Server' program to perform its task.

Case Study 3 : Doodle A graphics program*Doodle.vbp*

As with any program we start by creating a new project and adding the standard menu. File – Save / load /quit. The necessary code to quit is written as well.

I inserted a picture box called 'workspace' that will be used as the target of the drawing operations. I created a couple of frames as well. The first frames holds 3 option buttons. They will allow me to select the kind of shape I want to draw. In this frame is also a checkbox that allows m to select a filled shape.

The other two frames contain each an array of 16 option buttons. These will be used to select the drawing colors



At the bottom I added a label as well. This will be used to display the cursor coordinates during the drawing operation. As you can see the option buttons that

allow the color selection have a strange appearance. I Set the caption to empty and changed the bgcolor to light yellow. The actual changing of the bgcolor has no particular meaning. It is just an indication for me that something will be done with this color from within the code.

The primary code attached to this form looks like this :

```
Dim Sbordercolor
Dim sFillColor

Private Sub bordercolor_Click(Index As Integer)
    Sbordercolor = Index
End Sub

Private Sub colorfill_Click(Index As Integer)
    sFillColor = Index
End Sub

Private Sub Form_Load( )
    Dim X
    For X = 0 To 15
        Bordercolor(X).BackColor = QBcolor(X)
        colorfill(X).BackColor = QBcolor(X)
    Next X
End Sub

Private Sub quitprogram_Click( )
    End
End Sub
```

In order to have easy access to the selected Bordercolor and fill color I created two variables : Sbordercolor and sFillColor. One of the option button arrays is called Bordercolor , the other is called colorfill. Whenever I click on one of these buttons I store the index of the clicked option button into the appropriate variable.

The form load routine is going to assign the colors to the radio buttons. I perform a sweep from 0 to 15 and ask the corresponding QBcolor . That QBcolor is then assigned to the Backcolor property of the radio button.

Finally the Quitprogram_click simply terminates execution

In order to display the coordinates I simply attach a small blurb of code to the MouseMove event of the workspace.

```
Private Sub workspace_MouseMove(Button As Integer, _  
    Shift As Integer, X As Single, Y As Single)  
  
    Label1.Caption = "X:" + Str$(X) + " Y:" + Str$(Y)  
End Sub
```

This will update the contents of the *label1.caption* property whenever a **MouseMove** is detected on the Workspace PictureBox.

With this out of the way we can concentrate on the real drawing routines. Depending on the selected shape in the first frame we need to decide what to do. We can only take action if the user clicks with his mouse to point to the coordinates of the shape he wants to draw.

Logically you would use the *workspace.click* event. But alas ... this does not return the coordinates where the user has clicked. So we need to attach code to the MouseDown event. The problem is that the user needs to click twice , the first click designates top left , and second click the bottom right corner. So I created 3 static variables to hold this information.

Remember static variables ? No ? Okay : static variables are not destroyed upon exiting the subroutine. So the next time I enter they are still existing , and the data in them is still valid. Read the section about variables again.

One of them is going to be used as a Boolean. The first time the user clicks the content will be 0. The if-then else clause will then execute the part of the code where X and Y get stored in StartX and StartY. It also turns the Firstclick to 1.

The second time the user clicks the program flow will run over the Else clause.

```
Private Sub workspace_MouseDown(Button As  
Integer, _  
    Shift As Integer, X As Single, Y As  
Single)  
    Static firstclick  
    Static startx, starty  
  
    If Button = 1 Then  
        If firstclick = 0 Then  
            ' this is the firstclick  
            ' store the coordinates  
            startx = X  
            starty = Y  
        Else  
            ' second click : execute  
        End If  
        ' toggle firstclick  
        If firstclick = 0 Then  
            firstclick = 1  
        Else  
            firstclick = 0  
        End If  
    End If  
  
End Sub
```

All we need to fill in now is the code to do the drawing. This code goes into the Else clause ('second click : execute).

First we need to decide what the user selected as shape style.

```

If drawtype(0).Value = True Then
    workspace.Line (startx, starty)-(X, Y) _
                    , QBColor(sbordercolor)
End If

```

If he selected line we are simply going to draw a line from the start coordinates to the current coordinates with the selected border color.

In a similar way the box can be created. Except that here the user could have selected to fill the box. Drawing a filled box with a different border color is not directly possible. So what I am going to do is draw two boxes. One full box with the FillColor. On top of that I am going to draw a second box with the border color.

```

If drawtype(1).Value = True Then ' rectangle
    ' if required draw a filled box
    If fillit.Value = 1 Then
        workspace.Line (startx, starty)-(X, Y), _
                        QBColor(sfillcolor), BF
    End If
    ' draw the outline
    workspace.Line (startx, starty)-(X, Y), _
                    QBColor(sbordercolor), B
End If

```

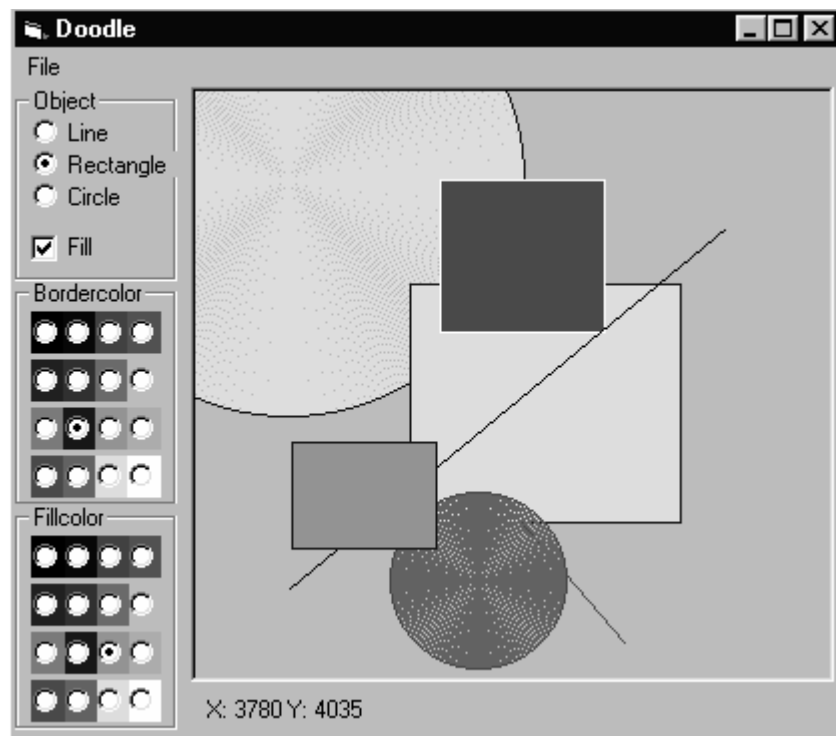
Very simple as you can see. The same is done for the circle command. Except that here the second X coordinate will determine the radius. The circle command does not allow you to create filled circles. So we have to be a bit creative here. Furthermore we can't use the x and y coordinates since we need to specify the radius.

The radius is determined by the difference between startx and x . This can be obtained by calculating the absolute value of Startx-x.

To create the filled image I draw a number of circles and change the radius from 1 to the maximum radius.


```
If drawtype(2).Value = True Then  
  If fillit.Value = 1 Then  
    For z = 1 To Abs(startx - X)  
      workspace.Circle (startx, starty), z,  
      —  
      QBColor(sfillcolor)  
    Next z  
  End If  
  workspace.Circle (startx, starty), Abs(startx  
  - X)_  
  , QBColor(sbordercolor)  
End If
```

The result of some doodling could look like this :



That's it. Now you can create simple drawings. Of course this is not like a real drawing program but it gives you insight in simple graphics operations.

You could add now the Save command using the Savepicture method , and load one using the Loadpicture command. But that is up to you.

Case Study 4 : The data terminal*MiniTerm.vbp*

The basics of this program are the same as any program. A form , A menu with a Quit entry and the usual Startup and exit code

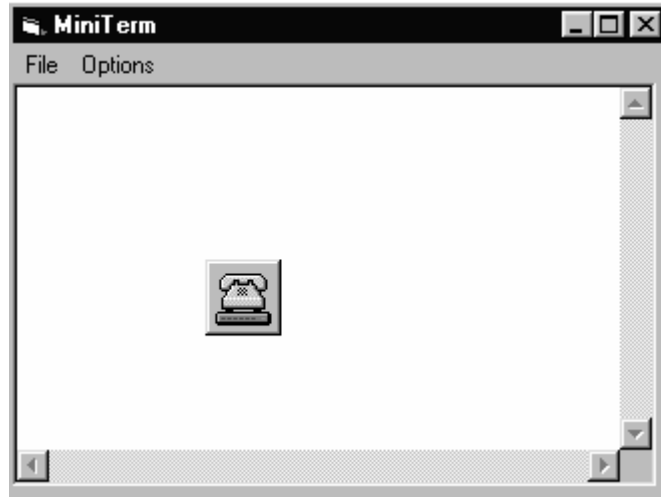


```
Private Sub Form_Load( )  
    Me.Show  
    DoEvents  
End Sub  
  
Private Sub quitprogram_Click( )  
    End  
End Sub
```

Since a terminal works accepts data from keyboard and remote site we need a control where we can display this data. Since , during the conversation with the other side , a lot of characters could have been sent , it might be good if there was a scroll back buffer of some sort.

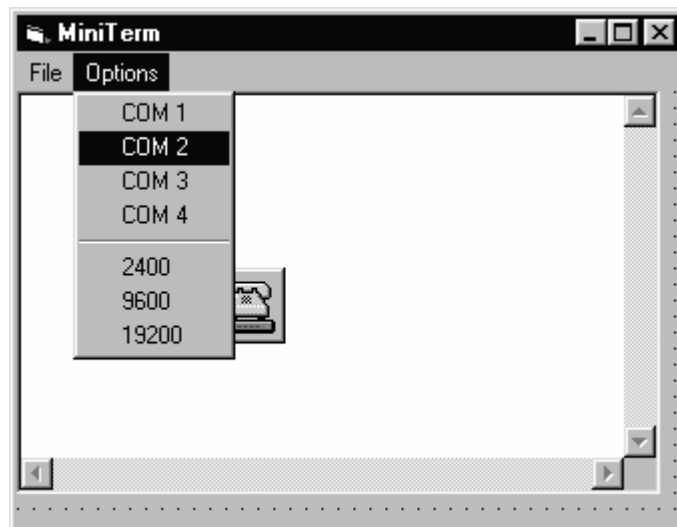
The easiest way to accomplish this is using a textbox with Multiline and scrollbars (both) turned on.

Next thing we are going to need is the MSComm object. Just enable this on the control toolbar and insert it onto the form. Never mind the location on the screen. Once the program starts running it will disappear anyhow.



Since we don't always know who is on the other side , it would be nice if we could select the port we want to talk to and also the baud rate setting.

The pitfall here is that , when we attempt to open a non-available port , or try to set it to impossible parameters we get a runtime error. So we need to tackle these. I know I have not explained you yet how to do that. For now , just ignore this. It is explained in Part III of this book.



To select comports I added an options menu where you can click the port and speed. If the port is not accessible , or the speed is invalid then the values will simply be grayed out. When the program starts I simply activate each of the options to allow this to happen.

```
Private Sub comport1_Click()  
    ' switch off the selected ports  
    comport1.Checked = True  
    comport2.Checked = True  
    comport3.Checked = True  
    comport4.Checked = True  
  
    ' make sure the port is closed  
    If rs232.PortOpen = True Then rs232.PortOpen  
= False  
    rs232.CommPort = 1  
    On Error GoTo openfailed  
    rs232.PortOpen = True  
    comport1.Checked = True  
    Exit Sub
```

```
openfailed:
    comport1.Enabled = False ' disable menu
End Sub
```

The above code will treat an occurring error by graying out the appropriate menu entry. During the startup of the program we simply call each of these routines. This will make sure the user gets to see only the available ports. The code for the four other comports is exactly the same except that we set the *RS232.comport* clause to the appropriate port , and make sure the correct menu entry gets checked.

```
Private Sub Form_Load( )
    Me.Show
    DoEvents
    comport1_Click
    comport2_Click
    comport3_Click
    comport4_Click
    comport1_Click
End Sub
```

Testing serial ports takes some time. Since the *Form_Load* gets executed before the form actually is displayed this might give the impression that the program is slow. So if we apply a little trick we give the user the feeling the program is fast. By explicitly executing the *Me.Show* statement we force the display of the form. Of course we need to give Windows the time to do this, Thus we execute the *DoEvents* command.

The code to control the baud rate is simply going to close the current selected port , change the baud rate and reopen it.

```
Private Sub baud2400_Click( )  
    If rs232.PortOpen = True Then rs232.PortOpen  
    = False  
    rs232.Settings = "2400,N,8,1"  
    rs232.PortOpen = True  
End Sub
```

What we have built so far is the entire user interface and the options to allow the user to customize the program configuration. Now we have to focus on the real communication.

The MSComm object generates an OnComm event whenever something happens on the serial port. If we then check if it was an incoming character we can simply read it from the buffer and send it to the textbox. This would allow us to receive data.

```
Private Sub rs232_OnComm( )  
    Select Case rs232.CommEvent  
        ' Handle each event or error by placing  
        Case comEvReceive ' Received  
        RThreshold # of  
                                ' chars.  
                                Text1.Text = Text1.Text +  
rs232.Input  
                                rs232.Input = ""  
    End Select  
End Sub
```

In order for this to work you must not forget to set the Rthreshold to 1. This makes sure the OnComm event gets fired for every incoming character.

To transmit data we need to check when the user types something. Now we can't use the *Textbox*.**change** event since it can be changed by the data

receiving routine as well. Another problem would be that we need to extract the last character of the entire text since that would be the last character typed. This would involve moving a lot of data every time and make the program slow. But we have a Keypress event. Furthermore the keypress event returns us the ASCII code of the pressed key. So that is exactly what we need

```
Private Sub Text1_KeyPress(KeyAscii As Integer)
    'ship the character to the comport
    ' here you could first do translation if
    required
    rs232.Output = Chr$(KeyAscii)
End Sub
```

Of course the presented program is a simple case. You could improve it largely by allowing it to a CR/LF translation. Some machines (like UNIX) only use LF to indicate a new line. PC's require CR/LF. So you could add options to translate this. This could be done by checking the ASCII code of the incoming character, or keypress ,and changing the code to the appropriate code(s).

The following routine would translate an incoming LF to CR/LF

```
Private Sub rs232_OnComm( )
    Select Case rs232.CommEvent
        ' Handle each event or error by placing
        Case comEvReceive ' Received
            RThreshold # of
                                ' chars.
                                x$ = rs232.Input
                                rs232.Input = ""
                                If x$ = Chr$(10) Then
                                    Text1.Text = Text1.Text +
vbCrLf
                                Else
                                    Text1.Text = Text1.Text +
rs232.Input
                                End If
            End Select
End Sub
```

This one will send a CR/LF whenever you hit ENTER.

```
Private Sub Text1_KeyPress(KeyAscii As Integer)  
    'ship the character to the comport  
    If KeyAscii = 13 Then ' CR to CRLF translate  
        rs232.Output vbCrLf  
    Else  
        rs232.Output = Chr$(KeyAscii)  
    End If  
End Sub
```

So we managed to write a little terminal program in a matter of minutes. You can of course extend this much further , but that is not the job of my book. I have given you the basis of how to do serial communications and how to respond to the events associated with it.

Case Study 5 : AlphaServe : A Telnet server**Aserve.vbp**

The basics of this program are the same as any program. A form , A menu with a Quit entry and a textbox that will act as a console.

In order to have access to TCP/IP we need to instantiate the MsWinsock control. Since we are building a server we will need a public and a private object. Why will come clear later.

In the startup code of the form we need to specify that the Public object (in this case the Public Socket) has to listening to port 23. Port 23 is the port used for Telnet connections. The definitions of these ports are defined in the TCP/IP protocol.

```
Private Sub Form_Load  
    PublicSocket.Localport=23  
    PublicSocket.Listen  
End sub
```

The above code will make that happen.

Next we have to attach some code to the Connectrequest event of our Public Socket

```
Private Sub Publicsocket_ConnectionRequest(byval  
    _  
        requestid as long)  
If privatesocket.state = sckClosed then  
    Privatesocket.Localport=0  
    Privatesocket.Accept requestid  
    Privatesocket.Senddata "Hello there , _  
        you just logged on to me via  
TCP/IP !"  
End if  
End Sub
```

Whenever a connectionrequest is occurring we will check if the Privatesocket is free to receive connection. If so we will pass the connection request to Privatesocket and establish the link by sending some data.

Remember in Winsock that you are either a public listener or a connected Talk/Listener. A public listener will accept any data coming from anyone. It can also transmit data to anyone out there. But if two users are connected , both will receive the data.

The private listener is set up for point to point. Only valid connected users will get to see this.

The Publicsocket is listening to anyone transmitting on port 23. The Privatesocket sets up a private link to your machine.

That's it. It's that simple.

Case Study 6 : LoanCalc : Using Excel in your applications**LoanCalc.vbp**

This program demonstrates how to access functions in other programs. The basics of this program are the same as any program. Some new concepts will be introduced here.

The basis of the program is a couple of textboxes and a button to start the calculation

The three textboxes are designated txtAmount , txtYears and txtInterest respectively. The button is simply called cmdCalc.

```
Private Sub cmdCalc_Click()  
    Amount = Val(txtAmount.Text)  
    Years = Val(txtYears.Text)  
    Interest = Val(txtInterest.Text)  
    Payment = CalcPay(Amount, Years, Interest)  
    Label3.Caption = Payment  
End Sub  
  
Public Function CalcPay(Amount, Years, Interest)  
    Dim excel As Object  
    Const ExcelObject = "Excel.Application"
```

```
Set excel = CreateObject(ExcelObject) ' grab  
excel  
  
CalcPay = excel.Pmt((Interest / 100) / 12,  
Years * 12, -1 * Amount)  
excel.Quit  
  
Set excel = Nothing ' release excel  
End Function
```

The cmsCalc routine is pretty straightforward and self-explanatory. Let's focus on the actual CalcPay routine. First of all we need to declare a variable to access the external object 'excel'. For easiness sake let's call it simple Excel.

The second line will assign The Excel.application to this variable. Actually what we are doing here is obtaining a so-called 'handle' to excel. Once we have a handle to an object we can manipulate it.

That is exactly what happens in line three. We access the PMT function of the excel object. , pass it variables just like we would to any other object , and retrieve the result.

If you no longer need the object you can simply destroy it. This normally happens automatically. For instance if you close a form , the object contained in the form are automatically destroyed.

Since we created the object during runtime we are responsible need to destroy the object again. The pitfall in this case is that excel is an external program. While it was launched automatically upon creating the handle , it is not terminated automatically. Therefore we first ask Excel to terminate itself and then release the handle to excel by setting the handle of the 'excel' variable to Nothing.

Unfortunately not every program allows you to derive objects from it. In general all Office product allow you to do this.

Conclusion

This concludes Part II of this book. In the previous parts we have seen what visual basic is and what we can do with it. Ranging from simple applications to full-blown programs that can be communicating to the outer world.

Visual basic brings a vast amount of routines and procedures with it. These can exist either inside the compiled code or externally in libraries. Microsoft ships a lot of extra libraries with Visual basic. You can also find on the market a lot of 3rd party libraries and objects to use with VB. But we have been neglecting the biggest library of them all : The windows core itself.

In the next part we are going to dig deeper into this amazing system and explore it. I'm also going to show you how to build your own libraries and controls. This will even extend the capabilities even further.

Have Fun

Visual Basic

For Electronics Engineering Applications

Part III

Master Programming

Visual Basic

For Electronics Engineering Applications

Part III :

Master Programming with Visual Basic

Introduction

So far we still have only scratched the surface of what you can do with the Windows operating system. While VB unleashes most of it, you will sometimes find yourself in a situation where you ask yourself : how do they do that ?. you might have been trying to create a program that acts in a similar way as a program you have seen. Yet it seems close to impossible to do it. Well maybe you're not looking in the right place. While VB brings a wealth of procedures, functions, routines and objects, there is a tremendous amount of stuff dormant in any system, waiting for someone to open the can on it.

Windows itself is to be considered one huge collection of functions and routines that you too, as a VB programmer, can exploit to your benefit. This part of the manual will plunge into this matter and explain you how to take a lead start on creating very impressive applications.

Chapter 15:

Digging into Windows.

This chapter will take you deeper into the operating system. You will learn how to exploit the embedded functions of the operating system for the benefit of your programs

15.1 DLL's

What are DLL's ? Simply said : a Library of routines. The name itself : 'Dynamic Linkable Library' might be scary at first. But really it's nothing more than a library you can attach to your code and use the routines in it , just as they were your own written routines.

A DLL however is not like an ordinary program library. In a normal programming style a library is glued or 'linked' into a program during compile time. Suppose you create 50 applications using the same library and you load them all into memory at once. You would have 50 copies of your library loaded in your computers memory . what a waste of space. This is where the concept of DLL kicks in. The library is not embedded into an application. It's merely distributed with it. Whenever an application that needs it , gets started the operating system loads the library into memory. Any application that needs it can use this library. Upon termination of the last application that uses one particular library it is automatically unload.

Well, this is all nice but what can you do with them. Well. Most Windows programs are built out of 2 parts. A user interface and a DLL that contains the real workings of the program. Let's take excel as an example. The user interface is nothing more than a data manipulator. The real calculation routines are stored in a DLL library. This library gets referenced whenever the GUI part of excel calls for it.

Since you can access DLL's from VB you can use the Excel routines inside your program directly. Why write a complex math routine or graphic display routine if someone else already has done this.

Actually the example of excel is poorly chosen. There is no need to access the DLL since you can access the whole of Excel as an object.

A better example is the following.

There are some things you cannot do in VB . And I really mean Not. In a DOS based environment you could read and write memory locations at will using Peek and Poke. The same was true for IO operations .

Since Windows manages all of these things now , it doesn't like you fiddling around with them. That is the main reason why Microsoft has left these operations out. As a matter of fact , they are gone in most programming languages. But in assembler you are still king of the system. There you can do whatever you want . So if you could create a routine which accesses hardware , compile it into a DLL and then use it in your program Now if only someone would write this.... . More on this later.

VB provides a mechanism to access any DLL library. All you need to know is the name of the library , the function name and the arguments it takes. When a function in a DLL is a reserved keyword for basic you can use the ALIAS statement to specify a different name for it.

You have to think of the operating system as a layered structure. Deep buried inside it are the most low level functions. While moving to the outer layers the operations become internally more complex but to the programmer easier to use. VB , and any other programming language for that matter ,exposes only the top layer of this structure directly. If you want to access the layers below you need to do API access. Every part of the operating system resides somewhere as a function you can access. Sometimes these DLL's are disguised as drivers or whatever. But you can still reach them.

15.2 Accessing DLL routines

This is done exactly like you would create any other subroutine or procedure. Except that now you precede the Sub or Function with the Declare keyword. This denotes that what you are going to do now is specify an already existing routine.

Syntax:

```
[Public | Private] Declare Sub name Lib _
    "libname" _
    [Alias "aliasname"] [(arglist)]

[Public | Private] Declare Function name Lib _
    "libname" [Alias "aliasname"] [(arglist)]
[As type]
```

Name	Any valid procedure name. Note that DLL entry points are case sensitive.
Lib	Indicates that a DLL or code resource contains the procedure being declared. The Lib clause is required for all declarations.
Libname	Name of the DLL or code resource that contains the declared procedure.
Alias	Indicates that the procedure being called has another name in the DLL. This is useful when the external procedure name is the same as a keyword. You can also use Alias when a DLL procedure has the same name as a public variable, constant, or any other procedure in the same scope. Alias is also useful if any characters in the DLL procedure name aren't allowed by the DLL naming convention.
Aliasname	Optional. Name of the procedure in the DLL or code resource. If the first character is not a number sign (#), aliasname is the name of the procedure's entry point in

	the DLL. If (#) is the first character, all characters that follow must indicate the ordinal number of the procedure's entry point.
arglist	Optional. List of variables representing arguments that are passed to the procedure when it is called.
type	Optional. Data type of the value returned by a Function procedure; may be Byte, Boolean, Integer, Long, Currency, Single, Double, Decimal (not currently supported), Date, String (variable length only), or Variant, a user-defined type, or an object type.

For Function procedures, the data type of the procedure determines the data type it returns. You can use an *As* clause following *arglist* to specify the return type of the function. Within *arglist*, you can use an *As* clause to specify the data type of any of the arguments passed to the procedure. In addition to specifying any of the standard data types, you can specify *As Any* in *arglist* to inhibit type checking and allow any data type to be passed to the procedure.

Empty parentheses indicate that the Sub or Function procedure has no arguments and that Visual Basic should ensure that none are passed. In the following example, *First* takes no arguments. If you use arguments in a call to *First*, an error occurs:

```
Declare Sub First Lib "MyLib" ( )
```

If you include an argument list, the number and type of arguments are checked each time the procedure is called. In the following example, *First* takes one Long argument:

```
Declare Sub First Lib "MyLib" (X As Long)
```

Note

You can't have fixed-length strings in the argument list of a Declare statement; only variable-length strings can be passed to procedures. Fixed-length strings can appear as procedure arguments, but they are converted to variable-length strings before being passed.

Note

The `vbNullString` constant is used when calling external procedures, where the external procedure requires a string whose value is zero. This is not the same thing as a zero-length string (`""`).

15.3 On Passing parameters to procedures and functions

In the previous section we have briefly seen the `ByVal` and `Byref` keywords. So let's take a closer look at them and what can be done with them.

First we need to understand the concept of a 'Pointer'. This is something often used the 'C' language. Most beginning programmers find this a very difficult concept. Well it is not. The problem is that most books explain it very poorly.

Then what is pointer? Well the name really says it all. It is something that points to something else. Suppose you I am an office clerk and you are my chief. You come in and ask for a business report A534. As a good clerk I jump up from my chair go to the filing cabinet, pop out the report and hand it to you. Of course you are very pleased with my swift response and my neatly organized filing cabinets. The next day I am not at my desk (I'm ill and at home). You come and ask for the same report. But where is it ??? . Now if only I had given you a reference to where it was ... That is exactly a pointer.

Now this comparison is not exactly one to one. In real programming life a pointer is the following. You can either pass a variable `ByVal` (By value : content) or `ByRef` (By reference : pointer). Passing something `ByVal` means that you will pass the contents of the variable. If I pass you the variable `X` `ByVal`, and `X = 5` then I am passing you 5.

On the other hand if I pass it ByRef (By reference) I am passing you the locations of where X is stored. Then you can retrieve the contents of X.

Now what good is this. Well very simple. If I pass you what is stored in the variable you can use this data. End of line. If I pass you where it is stored , you can retrieve it but you can also MODIFY it ! This can be very dangerous. Typically when something goes wrong in a program , and it contains a pointer then that is the first place to look at. Another big problem is that most programmers , which think they fully understand pointers , use pointers whenever they like.

A pointer is something you only use when you want to give the ‘demanding’ side the privilege to modify the contents of the location (variable) it points to.

Now there is a vary tricky part here. Unless you explicitly state that some variable to be passed by Value it is passed by reference. The reason is that it saves both time and memory to do this. So be careful never to change the contents of passed variables inside a procedure unless you want to do this !.

Where can you use this ? Simply said : Function and procedure calls that return more then one result.

Suppose you have a Procedure called SwapVariables. This procedure will swap the contents of two variables. You can’t do it with a function , because it can only return you one of the two. Furthermore you would need extra storage space to temporarily hold the data. This consumes , memory and CPU time.

If you give that function access to the passed variables you can do it.

```
Sub SwapVariables (ByRef X, ByRef Y)
    Dim Tmp
    Tmp = Y
    Y = X
    X = Tmp
End sub
Sub Form1_Load( )
    X = 5
    Y = 3
    Debug.print X,Y
    SwapVariables x,y
    Debug.print X,Y
End sub
```

15.4 API programming

What is this API ?.

API is short for Application Programming Interface. It is an interface between your applications and the computer's operating system. In a complex system like Windows you can't just fiddle around with whatever you want. Low-level stuff like disk access , screen update etc is handled by the operating system. It's a very bad idea to mess with these when you are not absolutely sure about what you are doing. Windows exposes all of the underlying code through this API. Even the most primitive operations can be done using this interface. Not only windows , but also every windows program , can have an API. The same goes for windows components like network interfaces , Database engines etc.

The Windows API is a very rich set of commands (over 5000 routines) that can enable you to perform task which are not normally possible. As explained in the section about DLL's , with a normal programming language you scratch only the surface of Windows.

Explaining all of the functions in the Windows API would take a course of a couple of months and a book a couple of thousand pages long. Someone has done this however. The best source is the technical documentation Microsoft

provides through its Developers Network. This is to be considered the 'bible'. However, due to the very technical nature of this information it is not always clear to understand. There is a very good book by Dan Appleman that describes the Windows API functions and how you can use them with Visual Basic. VB itself has already an include file that predefines most of the API calls you can perform inside windows.

15.4.1 A simple API example

Suppose you want to make the computer emit a simple Beep. Well duh, you can't. Fortunately the API of windows has this function because it needs it to allow command.com emit beeps. When you launch a DOS window then the beeps generated there are translated into a call to a kernel function of windows.

All you have to do is simply expose this function to VB.

```
Declare Function Beep& Lib "kernel32" (ByVal  
    defer_  
        As Long, ByVal duration As Long)
```

And there you have it : your first DLL access ! Now you can make windows emit simple beeps and sounds. Check out the Beepdemo.vbp file to learn more.

Note :

The Win95 platform no longer supports the parameters Frequency and Duration. They are simply denied.

So what resides where in this huge windows library ? Actually there is more than one library inside windows. The operating system itself is a layered construction of processes that communicate amongst them. It is neatly organized in chunks that have a certain, well-specified task.

Windows kernel modules

Kernel32.DLL	Low level operating functions, Memory management, task management, resource handling and related operations
User32.DLL	Windows management, Messages, Menu's, cursors, carets, timers, communication etc
GDI32.DLL	Graphics device interface : device output , all graphics manipulations, display content, metafiles, coordinate handling and font management
COMDLG32.DLL	Additional stuff for common dialogs,
LZ32.DLL	file compression

Extension libraries

COMCTL32.DLL	New windows control like tree list and richtext edit
MAPI32.DLL	Mail interface. Allows you to read ,create and send messages via e-mail systems
NETapi32.DLL	Control and accessing of networks
ODBC32.DLL	Open Database Connectivity Allows you to interface with a number of common database formats.
WINMM.DLL	Multimedia stuff, sound ,video etc

Most of the functions in the additional libraries are accessible via the built-in objects and controls of visual basic. The same goes for the things contained inside Windows. After all a simple button is contained in USER32.DLL.

To show you the power of these API calls a little example. Suppose you want to restrict the users cursor to certain boundaries. Any idea how to do that ? . No ? Of course not. This is against standard windows programming . But still you can do this , after all windows calls these every time you resize the desktop of your computer (video card resolution)

There is a API call to Clipcursor.

```
DIM myrect as RECT , mypoint as POINTAPI
DIM DL&
Mypoint.x = 0
Mypoint.y = 0
DL& = clienttoscreen&(pctcursor.hwnd,mypoint)
Meyrect.top = mypoint.y
Myrect.left = mypoint.x
Myrect.right = myrect.left +
pctcursor.scalewidth
Myrect.bottom = myrect.top +
pctcursor.scaleheight
DL& = clipcursor&(myrect)
```

Now all of the above might seem mumbo-jumbo to you and to be honest it is to most of us. API programming requires you to know a big deal on how windows works internally. You have to know how to obtain handles, create rectangles (not simple rectangles , a rectangle is the basic internal object that windows uses to do graphics manipulations.

A more practical use of the windows API is to get environment information. Suppose you want to check who is using the computer. You can do this by asking for the name of the user who logged into the system.

```
Declare Function GetUserName Lib "advapi32.dll"  
Alias _  
    "GetUserNameA" (ByVal lpBuffer As String, _  
        nSize As Long) As Long  
  
Public Sub Main()  
    Dim Buffer As String  
    Dim BufferSize As Long  
    Dim ReturnVal As Boolean  
    'Create Buffer  
    Buffer = Space(255)  
    BufferSize = Len(Buffer)  
    ReturnVal = GetUserName(Buffer, BufferSize)  
    MsgBox "User name is " & UCase(Trim(Buffer))  
End Sub
```

If you want to know more about API accessing you should read the book by Dan Appleman : VB5 programmers guide to the WIN95 API. This is the bible on Windows functions. 1541 pages no-nonsense reference material. Good luck !

Chapter 16 :

ActiveX Control Creation

This is a new ,catchy name to disguise OLE. Actually ActiveX is OLE. Now that does not exactly answer the question. Then what is OLE (or ActiveX for that matter).

It is a mechanism that allows you to 'embed' other programs inside you program. OLE literally means Object Linking and Embedding.

A practical example of this is the following : When you create a document in Microsoft Word you can embed tables inside this document. Word however does not know how to handle a table. After all it was designed to be a word processor , not a spreadsheet. But there is a program that masters tables : Excel. Well that's exactly what word is going to do. It will store the physical data , which it does not know how to handle inside the document. Upon opening the document it will read as much as it can and then ship the chunks of mystery data off to somebody else : excel. Excel will then visualize this part of the data on screen.

This is a very practical system. After all we are not here to reinvent the wheel all the time. Now this thing has drawbacks too. You need the other application to be able to interpret the data. IF you would make a little VB program and embedded excel into it (which is perfectly possible) you'd end up with a tremendously big executable. So there is another possibility

Programs are written in a layered way. You have the so-called core libraries. These are the real routines that perform the operations with the data. There is an API interface built into this. A user interface then reacts to the users command and calls the appropriate functions. .

When doing OLE , you are actually going to bypass the User interface and talk to the core of the system itself. This is of course a very cumbersome task. It means browsing through a list with thousands of calls. So this is not done very often.

The first method (embedding the entire system , including API and all) is called Document embedding. (or ActiveX document embedding).. The latter method is the real Object embedding.

Now what are the ActiveX objects then. Well they are simply small programs that are really embedded on Document style. However they are so small that you can consider them as objects. They have properties, methods , and events. Actually Excel could be thought of as an ActiveX object too, but since that can be used as a standalone program it is not considered an object.

And there you have it. An ActiveX object is a program that has no real use when used stand-alone. Consider a round button. What good would it do to write a program that creates a round button. You could click it ... and then what ?. But if you embed it into your program you can design a nice user interface to the worlds most astonishing program (written in VB of course ☺) .

You can , using Visual basic create your own ActiveX objects. You can compile them into OCX files and distribute them . Even C++ programmer can use them !. As a matter of fact any programming language for windows can call them and use them.

16.1 Creating an ActiveX Object.

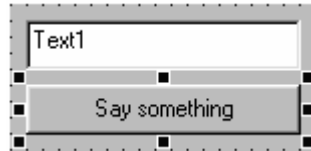


This is really quite simple. The basic process is similar to creating a regular project. Start a new project and select ActiveX Control. You will get an empty screen. Notice that there is no title bar or control box present.

This is typical for a control object. After all , you are going to place this on another form , so there is no need for title bars or control boxes.

Now you can insert standard windows objects. Take care to use only the objects that are available inside windows. Avoid to use additional objects and or pluggable components. It will work , but the problem is transporting your object to other computers. You will need to transport the embedded objects as well. Now this works fine as long as you can create a setup program for it. But since an ActiveX object gets compiled to an Object this information is lost in the process.

So lets add a little button (button1) and a textbox (text1).



We will attach code that displays a string into the textbox. In the Control_Load form we assign a certain text to this string.

```
Dim atext$

Private Sub button1_Click( )
    Text1.Text = atext$
End Sub

Private Sub UserControl_Initialize( )
    atext$ = "Hello world"
End Sub
```

Note:

Similar to a projects startup code (Form_Load) there is a Control_load that marks the entry into the control. This gets executed whenever a control is loaded. If you load multiple copies this code will execute each time. But don't worry each copy (or instance) has a life of it's own.

The next thing we have to do now is add a project to the existing one in order to be able to test our control. I have not told you this before but you can compile multiple projects at once. This is called creating a group

So first rename the current project to MyHello.VBP and the control file to MyHelloctl.

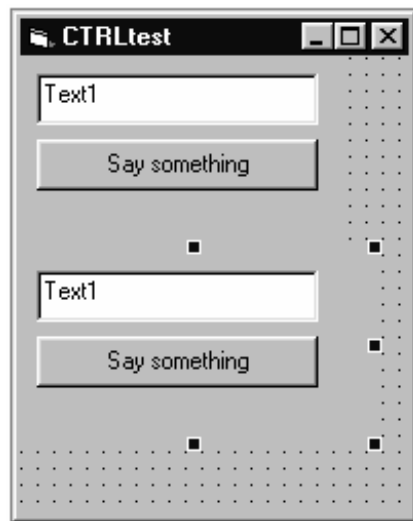
Now let's add a project <File><Add project>. Save this project as ctrltest.vbp and ctrltest.frm. No finally save the project group as Myhello.grp.

First we need to make sure that the Form window for the control is closed (minimized is NOT good). The reason for this is that a control leads a life of it's own. Once you insert a control it automatically starts running. As long as you have the user interface designer is open for this control it cannot be initialized properly.

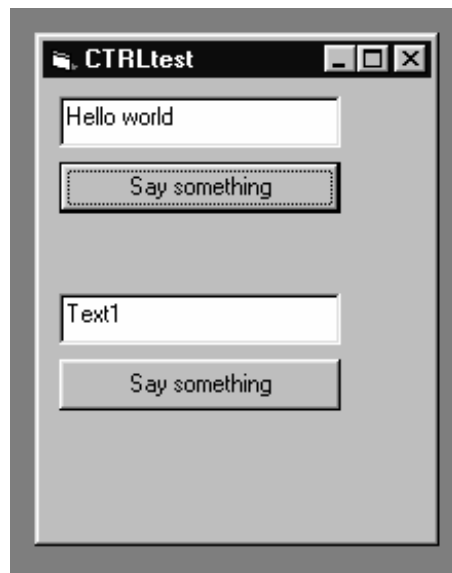
You will notice that on the control browser there is no a new symbol. (the last one inserted here displayed with a gray box around it). This is actually our control. So you don't really need to compile it first to make it run.



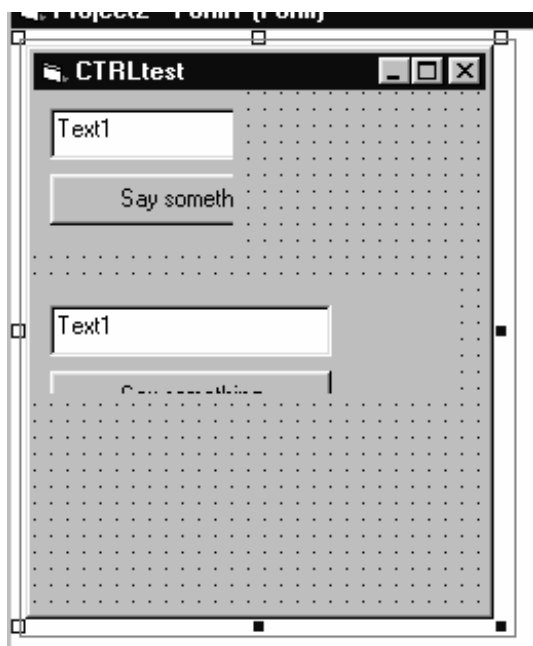
If we now insert this control twice we should get a screen like the following.



Now if you hit the Run button (F5) you can try clicking on the two buttons and you will see that you have two really independent controls. It's that easy.



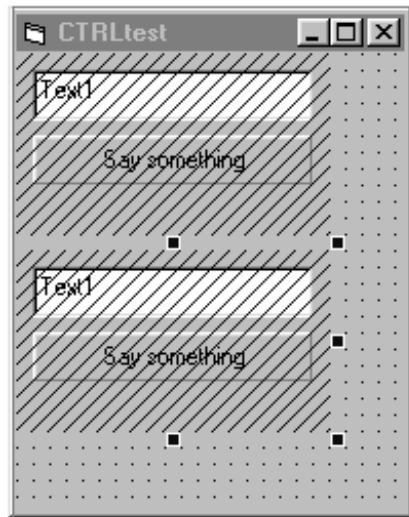
You might have noticed that you can size the control when putting it on the form. But , alas , it does not really size. You merely 'cut' it .



You can control this by attaching code to the `form_resize` event of your control. Let's force it to a fixed size. Simply attach this code to the control.

```
Private Sub UserControl_Resize()  
    UserControl.Width = 2295  
    UserControl.Height = 1125  
End Sub
```

To do this: reopen the form of the control. You will now notice that the controls already on the screen will be come grayed out with diagonal lines.



This is your indication that the user interface designer for that control is open and that what you see on the screen is no longer representative since the control is under modification

After insertion you close the control design form again and you will immediately see the control on the screen being updated.

If you try to select and size them you will see that they will jump to a fixed size and that you cannot stretch them anymore. This is because the `Usercontrol_resize` event 'glues' their size stuck.

The next page shows the full code of the very first control we made.


```
Dim atext$

Private Sub button1_Click()
    Text1.Text = atext$
End Sub

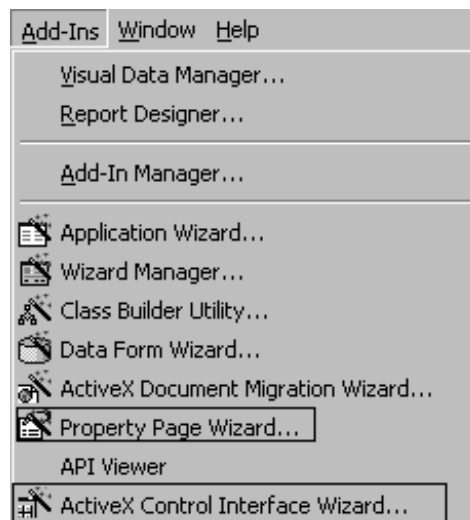
Private Sub UserControl_Initialize()
    atext$ = "Hello world"
End Sub

Private Sub UserControl_Resize()
    UserControl.Width = 2295
    UserControl.Height = 1125
End Sub
```

16.2 Adding property's and events

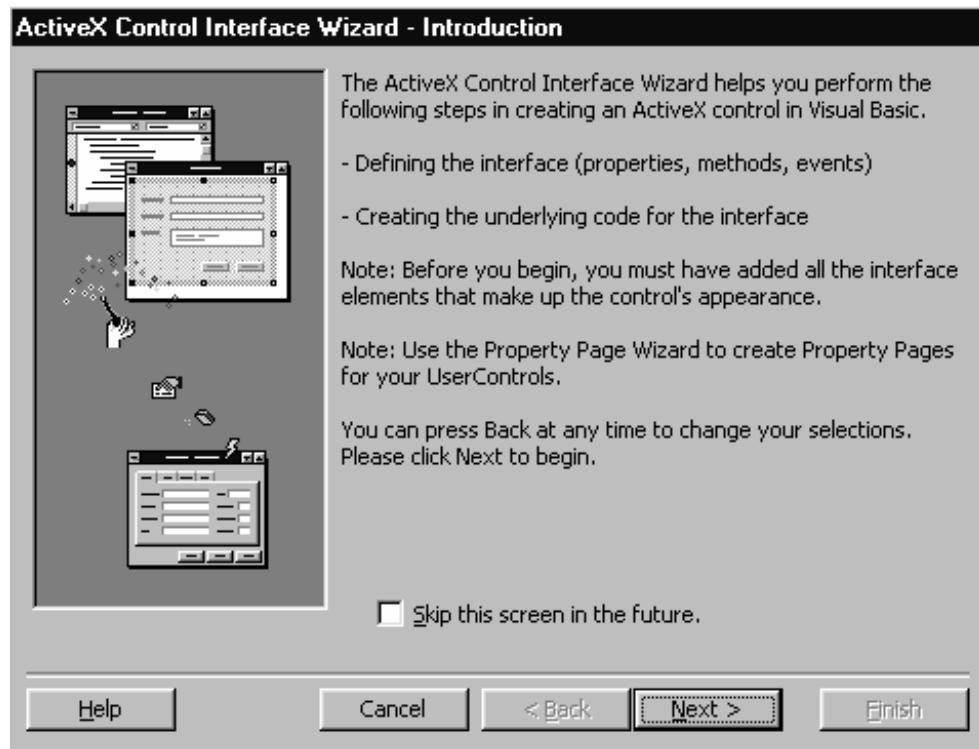
But what good is a control if you cannot communicate with it. As you have seen , standard controls like buttons have properties and events. All we have made so far is a dumb box we can insert and play with , but which is not interacting with the programs we are writing.

So we need to add some properties and events. Visual basic has a plug in that will assist you in doing this. You have to insert this in the Add-Ins menu.



Start the Add-In Manager and select the Property Page and ActiveX Control Interface Wizard. Once you have done this you can launch the ActiveX control Interface wizard.

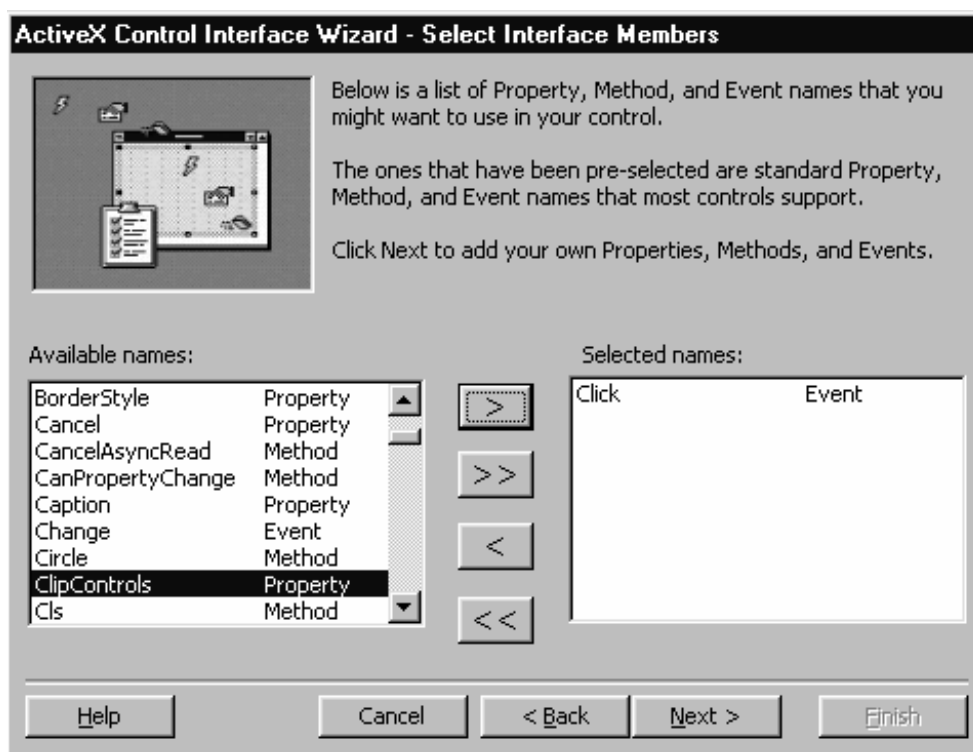
You will get a nice screen that you should read the first time in order to get a better understanding of what we are going to do now.



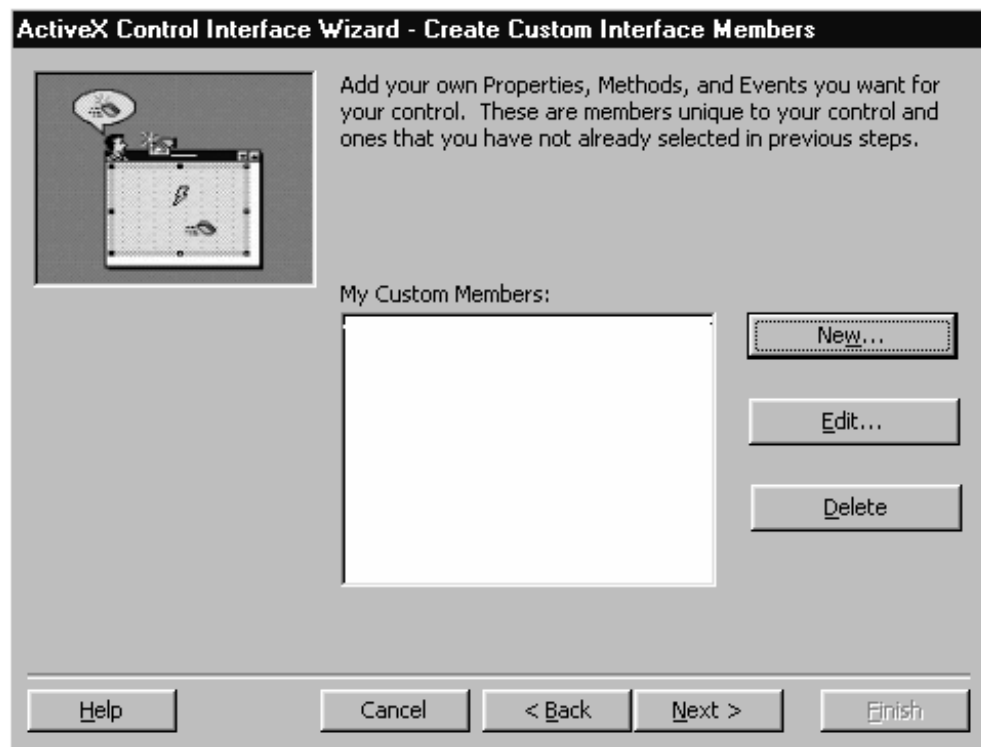
For now you can just click on Next. The next screen shows you the available STANDARD properties , methods and events you can attach to your control.



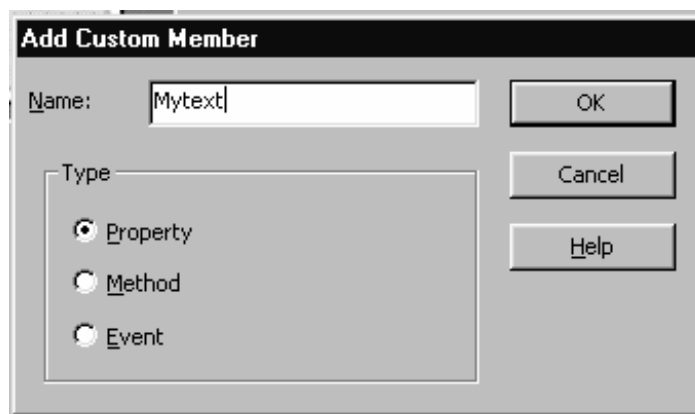
On the left side you see what is available and on the right side you see what is implemented for now. Since we are building a very simple control we don't need these. So let's make this list empty by clicking on the double arrow button which points left (<<). This makes the list empty. We are only interested in a Click event. So search for the Click event in the left listbox, select it and click on the single arrow right (>) button.



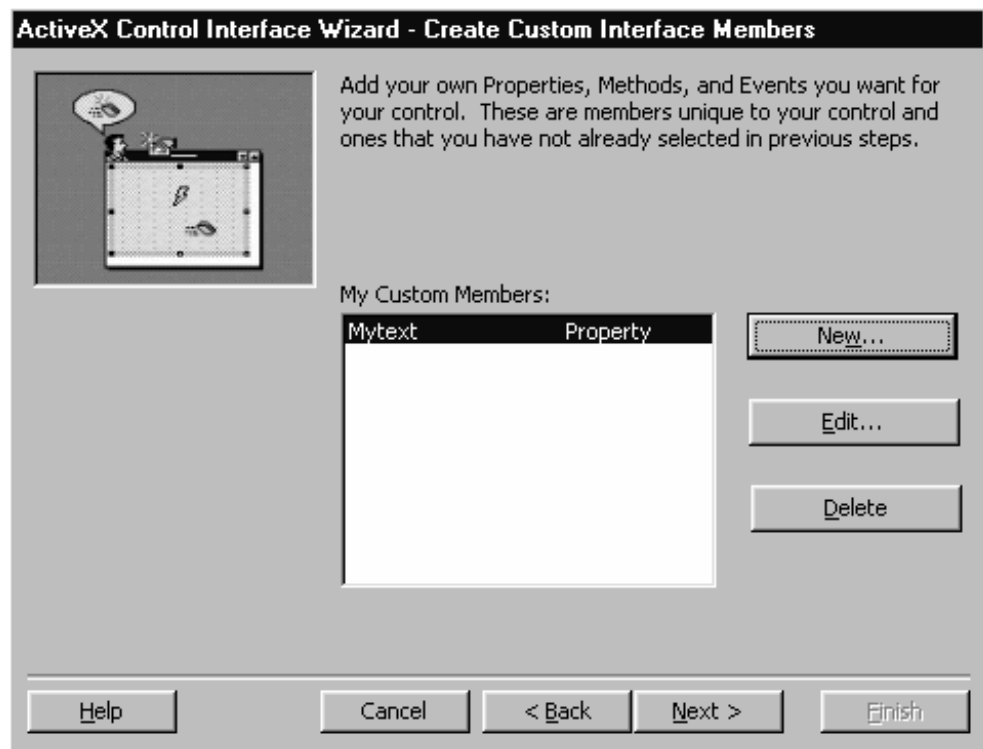
When that is done we can move on to the next screen: Inserting our own custom properties, methods and events. So go ahead , click next.



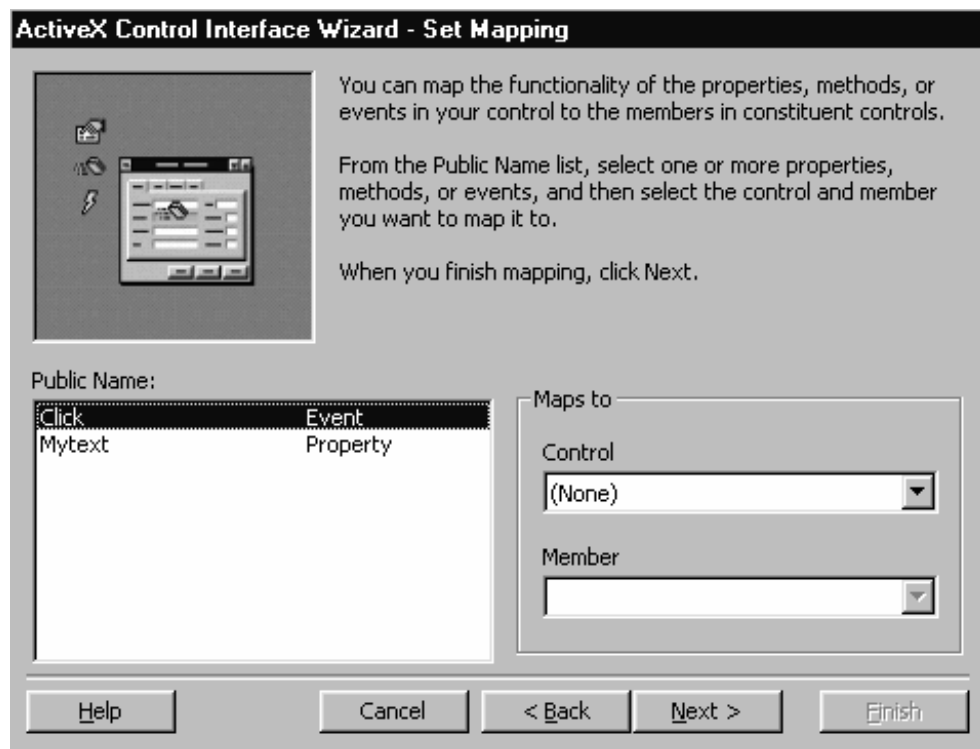
This screen shows you the display to enter your own items. So go ahead and click New.



Simply type the name you want to give it and the class it belongs to. In our case this is a property called MyText. Click OK when done.



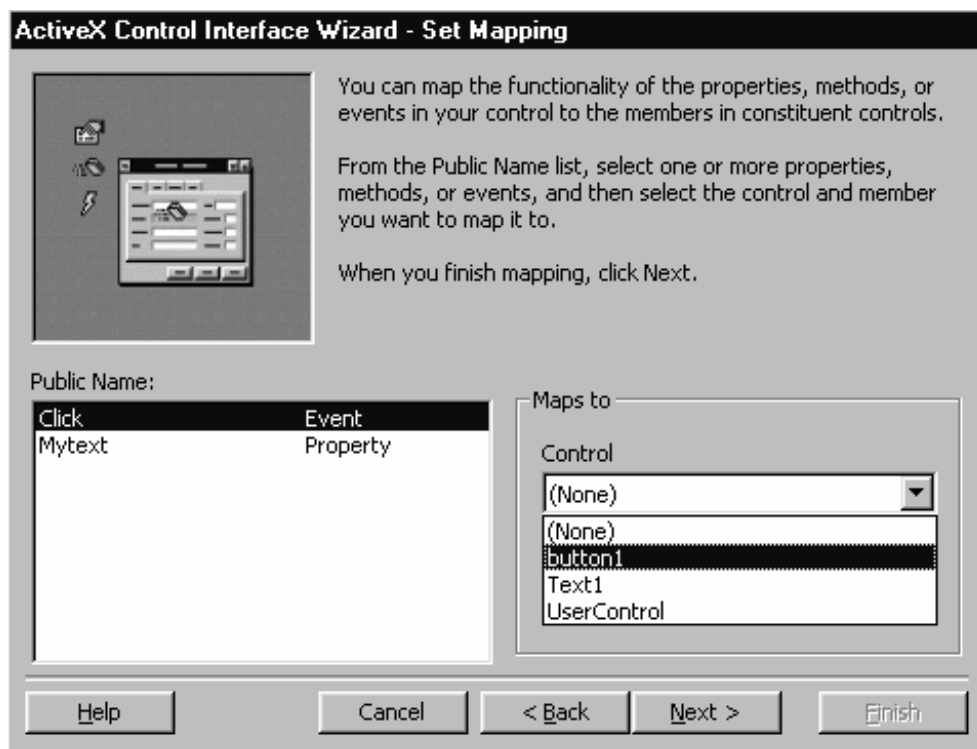
since this is all for now we click next to continue. If you would like to add more things you could just go on here.



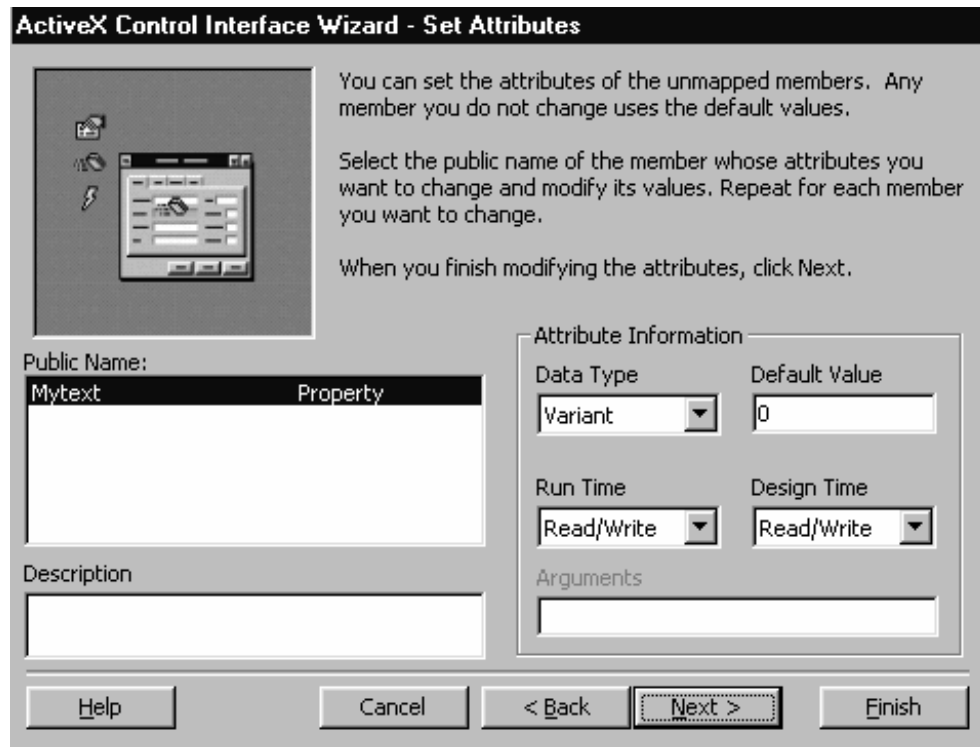
The next window is the so-called Set mapping. This allows us to map the things we selected directly to existing controls in the object.

In our case is want to map the Click event of my control to the Click event of the button1 object in my control. This means that , whenever I click the button the click event will be fired for my control. The user of my control can then take action. If I would have defined a custom event I could map it as well. Mapping is done by selecting the item of desire and specifying the object in the 'Maps To' frame. When you select a control here you will be able to select the event , property or method in the Member list.

For now I only want to map this click event.



So I select Button1 as the target control. And the Click event as the target member. Finally I click on next.



The last screen we need to fill out is the attribute definition. This allows us to define the type of attributes of the properties defined. In our case the Mytext property has to be set to variant. In the attribute information you can also specify if the attributes need to be available at runtime , or design time or both. You can also select if they have to be read, write or read/write. And you even give them a default value.

The default value is the value that will be shown on the properties browser when we are manipulating our control. For now we will not use that feature.

So now we can click on Finish. Finally the wizard will update the code you have written so far with the additions you just made

16.3 What the wizard came up with ...

After completing the above process the wizard will generate all necessary code to implement the specified events, properties and methods. While you could do all of this manually, it is handier and faster to let the wizard handle all of this. But what did the wizard generate ?

```

Dim atext$
Const m_def_Mytext = 0
Dim m_Mytext As Variant
Event Click() 'MappingInfo=button1,button1,-
1,Click

Private Sub button1_Click()
    RaiseEvent Click
    Text1.Text = atext$
End Sub
Private Sub UserControl_Initialize()
    atext$ = "Hello world"
End Sub
Private Sub UserControl_Resize()
    UserControl.Width = 2295
    UserControl.Height = 1125
End Sub
Public Property Get Mytext() As Variant
    Mytext = m_Mytext
End Property
Public Property Let Mytext(ByVal New_Mytext As Variant)
    m_Mytext = New_Mytext
    PropertyChanged "Mytext"
End Property

'Initialize Properties for User Control
Private Sub UserControl_InitProperties()
    m_Mytext = m_def_Mytext
End Sub
'Load property values from storage

```

```
Private Sub UserControl_ReadProperties(PropBag
As PropertyBag)

    m_Mytext = PropBag.ReadProperty("Mytext",
m_def_Mytext)
End Sub
'Write property values to storage
Private Sub UserControl_WriteProperties _
    (PropBag As PropertyBag)

    Call PropBag.WriteProperty("Mytext",
m_Mytext, _
    m_def_Mytext)
End Sub
```

Now the above piece of code will probably look very confusing. You will see references to property bags , Get , Let etc What is all of this stuff you might ask ?

The property-bag is a storage space that an object uses to store the name references to its internal workings. The real names of the properties it allows you to manipulate are stored there. Get and Let are functions that are called whenever a property changes. If , during design time , you change one of the objects properties , the Let function will be called and take appropriate action. The same will happen if during the run of your program you change this property. If you read the property the GET function will kick in. The availability of these function depends directly on the parameters you have given in the 'set Attributes' form of the ActiveX control interface wizard.

15.6 A closer look at the final code.

Let's take a look at the code generated by the wizard.

```
Dim atext$  
'Default Property Values:  
Const m_def_Mytext = 0  
'Property Variables:  
Dim m_Mytext As Variant  
'Event Declarations:  
Event Click( ) 'MappingInfo=button1,button1,-  
1,Click
```

The first line of code is still the same as we wrote. Then the wizard has inserted a constant to specify a default value for a parameter. This parameter is the MyText. When an object initializes it cannot do this with un-initialized variables. Then, in order to have clean code, the wizard has defined the m_Mytext variant. M_MyText is a temporary holding place for the data of the MyText property.

Last it has declared the click event. This event is followed by what appears to be a remark. THIS IS NOT A REMARK !. Don't remove this. The compiler reads this information and will see where to map the event. Actually this is the place, as you can see, where my Click event gets mapped to the Button1 Click event. The default value is -1; this means the button is initially not clicked.

```
Private Sub button1_Click( )  
    RaiseEvent Click  
    Text1.Text = atext$  
End Sub
```

The next procedure that has been modified is the Button1_click event handler. The wizard has inserted a line that will trigger the Click event we defined.

RaiseEvent triggers will trigger whatever code the user of our object has attached to our objects Click event.

Finally the wizard has inserted the necessary code that will allow the user of our events to change and retrieve properties.

```
Public Property Get Mytext( ) As Variant  
    Mytext = m_Mytext  
End Property
```

This procedure allows the user to retrieve the Mytext property. It copies the contents of the temporary information (m_Mytext) to the users calling code.

```
Public Property Let Mytext(ByVal New_Mytext As Variant)  
    m_Mytext = New_Mytext  
    PropertyChanged "Mytext"  
End Property
```

Same story goes here. Except now this is the code that can change the Mytext property. Whenever the user assigns a new value to the Mytext property this code gets executed. It sets the m_mytext variable to the new value;

The PropertyChanged event is going to update the propertybag. This makes sure that the Property browser window during design time gets updated as well.

```
'Initialize Properties for User Control  
Private Sub UserControl_InitProperties( )  
    m_Mytext = m_def_Mytext  
End Sub
```

The initproperties code gets called whenever a new instance of the object gets loaded. Just as the object_Initialize procedure initializes the user interface , this code will initialize all variables. The final pieces of code allow windows and visual basic to store and retrieve the settings to and from the propertybag. As

explained above this is the placeholder that explains windows the available properties , and their respective names , from an object.

```

'Load property values from storage
Private Sub UserControl_ReadProperties(PropBag _
    As PropertyBag)
    m_Mytext = PropBag.ReadProperty("Mytext", _
        m_def_Mytext)
End Sub

'Write property values to storage
Private Sub UserControl_WriteProperties(PropBag
    _
        As PropertyBag)

    Call PropBag.WriteProperty("Mytext", _
        m_Mytext, m_def_Mytext)
End Sub

```

And that's it. The only thing left is the piece of code that makes the object do what it was intended for : sending a message to the textbox

In order to make it work fully we should now send the m_Mytext contents to the textbox upon clicking.

```

Private Sub button1_Click( )
    RaiseEvent Click
    Text1.Text = m_Mytext
    ' used to be    Text1.Text = atext$
End Sub

```

And there you have it . A fully operational object you can use in your programs. If you close the editors for the object you will see that the objects on the other projects form remain grayed out; this indicates that the object has been modified so much that the IDE cannot recover. You should select and delete the objects and replace them with fresh copies.

If you now look in the object browser you will see that there is a property called MyClick. Simply type there "Yo dude".

And if you double click on our object you will get code for the Click event. There we attach a message box.

```
Private Sub Form_Load( )  
    Myhello1.Mytext = "Yo dude"  
End Sub  
  
Private Sub Myhello1_Click( )  
    MsgBox " You clicked me "  
End Sub
```

And then : le moment supreme. Make it all work by hitting F5 ...

Of course this is only a simple and not so useful example , but it gives you an overview how to create your own controls. Its just as easy as creating a normal program. We only have to launch the wizard to assist us in building the necessary links.

Finally you can ask Visual Basic to create the OCX for the object and then you can distribute it to anyone.

Chapter 17 :

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 to Bad programs. Then what are bad programs.

Well I can give you a lot of examples

- A program that crashes very often
- Eats memory and does practically nothing
- Is terribly slow
- Looks way ugly (depends strongly on personal taste)
- Behaves strangely some times
- Corrupts and wastes your data

I Think we can all agree on the above. Well except maybe the 'Looks Way ugly'. This depends on personal taste. And I'm 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. Have Clean source code ! Don't make constructions that you yourself hardly understand. Insert comments . It doesn't hurt and will not waste memory or speed once compiled. And most of all adhere to the KISS principle.

17.1 The KISS Way

No , I'm 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. Don't 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's write it out so it becomes more clear.

```
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's gotten a bit better. But still what does it do ? Let's analyze.

If X is smaller than 4 it gets incremented with one. If not it is decremented with one. Then x is compared against 4. If it is 4 we jump to the label Stopit. If not we do it all over again. IF you think a bit more about this code then you will see that the decrementing part never gets executed. If 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. So 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 never get called. It cannot eliminate the above case of dead code.

So 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 you will see that all we are doing here is incrementing X until it reaches 4. So why not use a While Wend ?

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

Now isn't that a whole lot more readable. ?

This demonstrates a number of basic KISS principles

- One line = One command
- Avoid Goto's and Gosubs (except of course for error handling)
- Don't write dead code

The advantage of KISS is not only easy readable code. It will run faster , compile to smaller EXE's and most important. If you or someone else has to review this program months or years from now , he will understand what it is doing ! So by thinking a little longer about implementing a piece of code you can save yourself a lot of time , both during execution of the program , and during your work to maintain and update the program.

Adhering KISS rules will also lead you to using variable names with a meaning. Instead of Simply Using X and Y you should use names that have a meaning. After all , it has no impact on the speed of the program.

Another big point in KISS is :Don't reinvent the wheel !. Use as much of the existing things as possible. Don't create every the same routines over and over again.

So how can this be done? Well : Comment your code and partition it. This means that you divide your program in small manageable chunks and transform them into a routine. Then save this routine in a module. Later on , maybe in a different project , you can simply re-use this routine over and over again. Also make sure that the routines you write are well documented.

If you want to create re-usable code the two things that you should adopt is indenting your code and using CamelWriting. ..??.. Stop laughing. I'm dead serious here.

Indenting is the process where you indent lines of code depending on where they belong

```
If x=5 Then  
  B=4  
Else  
  Open "myfile" for output as #1  
  Print #1,x  
  Close #1  
  X=0  
  If b= 10 then  
    Open "myfile" for input as #1  
    Line input #1,a$  
    B=val(a$)  
    Close #1  
  End if  
End if
```

Well ?? Where do the 'end ifs' belong ? If you would indent it you could immediately see

```
If x=5 Then
    B=4
Else
    Open "myfile" for output as #1
    Print #1,x
    Close #1
    X=0

    If b= 10 then
        Open "myfile" for input as #1
        Line input #1,a$
        B=val(a$)
        Close #1
    End if
End if
```

Now you have to agree that this looks a lot more readable. You immediately see the If 'B=10 -then - else' block and the main 'If -Then Else' block. And now let's take a look at CamelWriting. ... Quit Laughing ! When writing Good and clean code you create names for functions , procedure or variables that are descriptive of their nature. Often this name will consist of more then one word. To make these names more readable for humans you should adopt this CamelWriting thing.

The name *thisisavariabale* becomes more readable if you write it as *ThisIsAVariable*. This style of writing , where you capitalize every first letter of a word is known a CamelWriting.

17.2 Atomic Programming

This is a derivative of the KISS principle that dictates the following.

Break your program in the smallest possible routines. (Atoms)

Suppose you have a program that contains a number of routines that all need to access a certain file . Depending on the procedure some will need to open it in

read mode , some in write mode. Some routines even need to do both. So inside these routines you will frequently open and close this file; It might be a good idea to eliminate all these lines of code and replace them with 2 custom procedures. OpenFile and CloseFile

```
Sub streamout (text$)
    Open "mystream" for output as #1
    Print #1,text$
    Close #1
End sub
Sub Streamin (message$)
    Open "mystream" for output as #1
        Print #1,message$
    Close #1

    Open "mystream" for input as #1
        Line input #1,text$
    Close #1
End sub
```

Suppose this is a big program that uses this stream to communicate with another program. Practically you would need to implement a lot more error handling since the other guy might have it open .

These routines (and others) that access the stream could be floating around anywhere. If you rewrite this to the following :

```
Sub streamout (text$)
    OpenStream
    Print #1,text$
    CloseStream
End sub
```



```

Sub Streamin (message$)
    OpenStream
        Print #1,message$
    OpenStreamRead
        Line input #1,text$
    CloseStream
End sub
Sub OpenStream
    Close #1
    Open "mystream" for output as #1
End sub
Sub CloseStream( )
    Close #1
End sub
Sub OpenStreamReadmode( )
    Close #1
    Open "mystream" for input as #1
End sub

```

You end up with something far more readable and maintainable. IF the file changes or you want to implement error handling now ,you can simply modify the OpenStream , CloseStream and OpenStreamReadmode procedures.

You don't need to go digging in thousands of lines of code to patch all of the open and close commands.

17.3 Naming objects

Another rule of thumb is , just like with variables , to give object and controls meaningful names. Don't keep the default things like button1 and button2 or text3 and text5. Give them a real name. And even better : Precede their name with a 3-letter abbreviation that describes their nature.

For a button you could use btnHelloWorld , the matching textbox could be txtHelloWorld. You would then write a procedure like this

```
Private sub btnHelloWorld_Click()  
    TxtHelloWorld.text="Hello World"  
End Sub
```

Years from now you will read this code and will be able to find out what it does and what it refer to.

Note:

To find a variable declaration quickly simply right click it and select the Definition option. The code editor will jump to the definition of the variable or procedure. To return , right click again and select Last Position

17.3 Error handling.

Even when you have been programming perfectly clean code, and have a fully bug free program , something can crew up. Something stupid, like running out of disk space, can crash even the most perfect program. Fortunately there are ways to deal with what is called 'externally invoked errors'. System failure , invalid filenames and more can be intercepted and handled by writing a bit of code.

Visual Basic has a built in object that provides an easy interface to handling errors. All you have to do is enable it inside your code.

When developing an application and an error occurs , you get a message that something went wrong and the debugger asks you to either terminate the execution of jump you to the line where it occurred. Once the program is compiled , this is no longer the case. IF an error occurs then you simply are kicked out of execution . You will get a notification that an error of some type has been detected.

17.3.1 The On Error Goto clause

The enables an error-handling routine and specifies the location of the routine within a procedure; can also be used to disable an error-handling routine. Error handling is always done inside a routine. You have to enable an On error Goto clause in the beginning of the routine. The target of the Goto has to be inside the same routine.

```
On Error GoTo < line >  
On Error Resume Next  
On Error GoTo 0
```

The On Error statement syntax can have any of the following forms:

Statement	Description
On Error GoTo line	Enables the error-handling routine that starts at line specified in the required line argument. The line argument is any line label or line number. If a run-time error occurs, control branches to line, making the error handler active. The specified line must be in the same procedure as the On Error statement; otherwise, a compile-time error occurs.
On Error Resume Next	Specifies that when a run-time error occurs, control goes to the statement immediately following the statement where the error occurred where execution continues. Use this form rather than On Error GoTo when accessing objects.
On Error GoTo 0	Disables any enabled error handler in the current procedure.

If you don't use an On Error statement, any run-time error that occurs is fatal; that is, an error message is displayed and execution stops. An "enabled" error handler is one that is turned on by an On Error statement; an "active" error handler is an enabled handler that is in the process of handling an error. If an

error occurs while an error handler is active (between the occurrence of the error and a Resume, Exit Sub, Exit Function, or Exit Property statement), the current procedure's error handler can't handle the error. Control returns to the calling procedure. If the calling procedure has an enabled error handler, it is activated to handle the error. If the calling procedure's error handler is also active, control passes back through previous calling procedures until an enabled, but inactive, error handler is found. If no inactive, enabled error handler is found, the error is fatal at the point at which it actually occurred. Each time the error handler passes control back to a calling procedure, that procedure becomes the current procedure. Once an error handler in any procedure handles an error, execution resumes in the current procedure at the point designated by the Resume statement.

Note

An error-handling routine is not a Sub procedure or Function procedure. It is a section of code marked by a line label or line number.

Error-handling routines rely on the value in the Number property of the Err object to determine the cause of the error. The error-handling routine should test or save relevant property values in the Err object before any other error can occur or before a procedure that might cause an error is called. The property values in the Err object reflect only the most recent error. The error message associated with Err.Number is contained in Err.Description.

On Error Resume Next causes execution to continue with the statement immediately following the statement that caused the run-time error, or with the statement immediately following the most recent call out of the procedure containing the On Error Resume Next statement. This statement allows execution to continue despite a run-time error. You can place the error-handling routine where the error would occur, rather than transferring control to another location within the procedure. An On Error Resume Next statement becomes inactive when another procedure is called, so you should execute an On Error Resume Next statement in each called routine if you want inline error handling within that routine.

Note

The On Error Resume Next construct may be preferable to On Error Goto when handling errors generated during access to other objects. Checking Err after each interaction with an object removes ambiguity about which object was accessed by the code. You can be sure which object placed the error code in Err.Number, as well as which object originally generated the error (the object specified in Err.Source).

On Error GoTo 0 disables error handling in the current procedure. It doesn't specify line 0 as the start of the error-handling code, even if the procedure contains a line numbered 0. Without an On Error GoTo 0 statement, an error handler is automatically disabled when a procedure is exited.

To prevent error-handling code from running when no error has occurred, place an Exit Sub, Exit Function, or Exit Property statement immediately before the error-handling routine, as in the following fragment:

```
Sub InitializeMatrix(Var1, Var2, Var3, Var4)
    On Error GoTo ErrorHandler
    . . .
    Exit Sub
ErrorHandler:
    . . .
    Resume Next
End Sub
```

Here, the error-handling code follows the Exit Sub statement and precedes the End Sub statement to separate it from the procedure flow. Error-handling code can be placed anywhere in a procedure.

Un-trapped errors in objects are returned to the controlling application when the object is running as an executable file. Within the development environment, un-trapped errors are only returned to the controlling application if the proper options are set. See your host application's documentation for a description of which options should be set during debugging, how to set them, and whether the host can create classes.

If you create an object that accesses other objects, you should try to handle errors passed back from them un-handled. If you cannot handle such errors, map the error code in `Err.Number` to one of your own errors, and then pass them back to the caller of your object. You should specify your error by adding your error code to the `vbObjectError` constant. For example, if your error code is 1052, assign it as follows:

```
Err.Number = vbObjectError + 1052
```

Note

System errors during calls to dynamic-link libraries (DLL) do not raise exceptions and cannot be trapped with Visual Basic error trapping. When calling DLL functions, you should check each return value for success or failure (according to the API specifications), and in the event of a failure, check the value in the `Err` object's `LastDLLError` property.

Example of an error handler:

```
Sub OnErrorStatementDemo( )  
    On Error GoTo ErrorHandler      ' Enable  
error-handling  
    Open "TESTFILE" For Output As #1    ' Open  
file  
    Kill "TESTFILE"      ' Attempt to delete open  
file.  
    On Error Goto 0      ' Turn off error  
trapping.  
    On Error Resume Next      ' Defer error  
trapping.  
    ObjectRef = GetObject("MyWord.Basic")  
    ' Try to start nonexistent  
    If Err.Number = 440 Or Err.Number = 432  
Then  
        ' Tell user what happened.  
        ' And clear the Err object.  
        Msg = "There was an error attempting  
to _  
                open the Automation object!"  
        MsgBox Msg, , "Deferred Error Test"
```

```
Err.Clear      ' Clear Err object
fields
End If
Exit Sub      ' Exit to avoid handler.
ErrorHandler:  ' Error-handling routine.
Select Case Err.Number ' Evaluate error
number.
Case 55      ' "File already open"
error.
Close #1     ' Close open file.
Case Else
' Handle other situations
here...
End Select
Resume      ' Resume execution at same
line
' that caused the error.
End Sub
```

17.3.2 The Err object

This object contains information about run-time errors. It tells you which error happened and can also give you a description of it. You can use it to simulate errors as well. This is very useful to test your error handlers when writing software.

The generator of an error—Visual Basic, an object, or the Visual Basic programmer, sets the properties of the Err object. The default property of the Err object is Number. Because the default property can be represented by the object name Err, earlier code written using the Err function or Err statement doesn't have to be modified.

When a run-time error occurs, the properties of the Err object are filled with information that uniquely identifies the error and information that can be used to handle it. To generate a run-time error in your code, use the Raise method.

The Err object's properties are reset to zero or zero-length strings ("") after any form of the Resume or On Error statement and after an Exit Sub, Exit Function,

or Exit Property statement within an error-handling routine. The Clear method can be used to explicitly reset Err.

Use the Raise method, rather than the Error statement, to generate run-time errors for a class module. Using the Raise method in other code depends on the richness of the information you want to return. In code that uses Error statements instead of the Raise method to generate errors, the properties of the Err object are assigned the following default values when Error is executed:

Property	Value
Number	Value specified as argument to Error statement. Can be any valid error number.
Source	Name of the current Visual Basic project.
Description	A string corresponding to the return of the Error function for the specified Number, if this string exists. If the string doesn't exist, Description contains "Application-defined or object-defined error".
HelpFile	The fully qualified drive, path, and file name of the Visual Basic Help file.
HelpContext	The Visual Basic Help file context ID for the error corresponding to the Number property.
LastDLLError	On 32-bit Microsoft Windows operating systems only, contains the system error code for the last call to a dynamic-link library (DLL). The LastDLLError property is read-only.

You don't have to change existing code that uses the Err object and the Error statement. However, using both the Err object and the Error statement can result in unintended consequences. For example, even if you fill in properties for the Err object, they are reset to the default values indicated in the preceding table as soon as the Error statement is executed. Although you can still use the Error statement to generate Visual Basic run-time errors, it is retained principally for compatibility with existing code. Use the Err object, the Raise method, and the Clear method for system errors and in new code, especially for class modules.

The Err object is an intrinsic object with global scope. There is no need to create an instance of it in your code.

Example :

This example uses the properties of the Err object in constructing an error-message dialog box. Note that if you use the Clear method first, when you generate a Visual Basic error with the Raise method, Visual Basic's default values become the properties of the Err object.

```
Dim Msg
' If an error occurs, construct an error message
On Error Resume Next      ' Defer error handling.
Err.Clear
Err.Raise 6 ' Generate an "Overflow" error.
' Check for error, then show message.
If Err.Number <> 0 Then
    Msg = "Error # " & Str(Err.Number) & _
        " was generated by " _
        & Err.Source & Chr(13) &
        Err.Description
    MsgBox Msg, , "Error", Err.Helpfile,
    Err.HelpContext
End If
```

17.3.3 Resuming execution after handling the error

The Resume clause can resume execution after an error-handling routine is finished.

```
Resume [0]
Resume Next
Resume <line>
```

The Resume statement syntax can have any of the following forms:

Statement	Description
Resume	If the error occurred in the same procedure as the error handler, execution resumes with the statement that caused the error. If the error occurred in a called procedure, execution resumes at the statement that last called out of the procedure containing the error-handling routine.
Resume Next	If the error occurred in the same procedure as the error handler, execution resumes with the statement immediately following the statement that caused the error. If the error occurred in a called procedure, execution resumes with the statement immediately following the statement that last called out of the procedure containing the error-handling routine (or On Error Resume Next statement).
Resume line	Execution resumes at line specified in the required line argument. The line argument is a line label or line number and must be in the same procedure as the error handler.

If you use a Resume statement anywhere except in an error-handling routine, an error occurs. The following example uses the Resume statement to end error handling in a procedure, and then resume execution with the statement that caused the error. Error number 55 is generated to illustrate using the Resume statement.

```
Sub ResumeStatementDemo( )  
    On Error GoTo ErrorHandler      ' Enable  
error-handling  
    Open "TESTFILE" For Output As #1    ' Open  
file  
    Kill "TESTFILE"      ' Attempt to delete open  
file.  
    Exit Sub          ' Exit Sub to avoid error  
handler.  
ErrorHandler:          ' Error-handling routine.  
    Select Case Err.Number  ' Evaluate error  
number.  
        Case 55          ' "File already open"  
error.  
            Close #1      ' Close open file.  
        Case Else  
            ' Handle other situations  
here....  
    End Select  
    Resume              ' Resume execution at same  
line  
                        ' that caused the error.  
End Sub
```

17.3.4 Trappable errors

Trappable errors can occur while an application is running. Some trappable errors can also occur during development or compile time. You can test and respond to trappable errors using the **On Error** statement and the **Err** object. Unused error numbers in the range 1 – 1000 are reserved for future use by Visual Basic.

The following sections give you an overview of errors that can occur. For ease of use they are categorized by cause. Note that I am not going to discuss all errors. Most of the possible error codes you will probably never get. I'm only giving the ones that pop up once in a while.

17.3.5 Syntax Errors (errors against the Basic syntax)

These typical occur when starting the program inside the IDE for the first time.

Code	Message	Explanation
3	Return without GoSub	Pretty clear
5	Invalid procedure call	You attempted to call a procedure but forgot to pass some parameters
13	Type mismatch	You tried to assign data to a variable of the wrong type like a string to an integer.
20	Resume without error	You errorhandler contains an error !
92	For loop not initialized	You have a Next without a for
35	Sub, Function, or Property not defined	You are accessing something which does not exist

17.3.7 Runtime errors

These occur during the run of the program. They are mostly because of flawed programming logic , or memory problems.

Code	Message	Explanation
6	Overflow	The result of a calculation is too big to store in the allocated variable, or is simply too big to be calculated at all.
7	Out of memory	Now how did that happen ?
11	Division by zero	A typical calculation error. There is no mathematical solution for dividing by zero.
14	Out of string space	You tried to cram more data into a too small string.
28	Out of stack space	Programming recursive stuff ?

17.3.8 Flawed Programming logic errors.

Code	Message	Explanation
9	Subscript out of range	This occurs when trying to access an inexistent array element. If you defined an array of 10 elements and try to read or write element 11 you will get this
10	This array is fixed or temporarily locked	You tried to Redim a static array
16	Expression too complex	Try breaking it down in simpler parts

17.3.9 File handling errors

These errors occur when handling files.

Code	Message	Explanation
52	Bad file name or number	There is a problem creating the handle
53	File not found	The file does not exist
54	Bad file mode	You tried to read from a file opened for output or vice versa
55	File already open	You tried to open an already open file
57	Device I/O error	Ouch! Serious one. The device where the file resides is not ready. Typically for floppy drives.
61	Disk full!	Disk Space. The final frontier ... or Get a bigger disk
62	Input past end of file	You tried to read beyond the end of the file
63	Bad record number	Can occur when reading records from files
67	Too many files	You attempted to store too many files in a directory
68	Device unavailable	The target where the file exists went offline . typical

		for removable media
70	Permission denied	You cant do that. The file is in use by someone else.
71	Disk not ready	A timeout on disk operations. Can happen on floppy drives or virtual file drivers
74	Can't rename with different drive	You can rename a file across drives
75	Path/File access error	The path or file does not exist
76	Path not found	The path is invalid

The above-mentioned errors are the most common. There are a lot more possible errors, but you should consult the on-line help for Visual Basic when those occur.

Note

Any error occurring during the run of the executable, that is not handled properly, is FATAL. This means bye-bye program. So start writing error handlers.

Chapter 18 :

The Windows registry

You can consider this as 'advanced digging into windows' The registry is probably the most obscure part of windows. Most people still regard to this as the mythical place where Windows stores the data needed for its internal operation. Well it's more than that. You can use it too. Under the older Windows version you would store configurations for your program in separate INI files. Now you can use the Registry. However you cannot manipulate the registry directly. The reason for this is that the registry is managed by windows and you cannot just start changing this file. Also it is a very complex structure where a lot of information is stored. A simple screw-up could result in a total system crash.

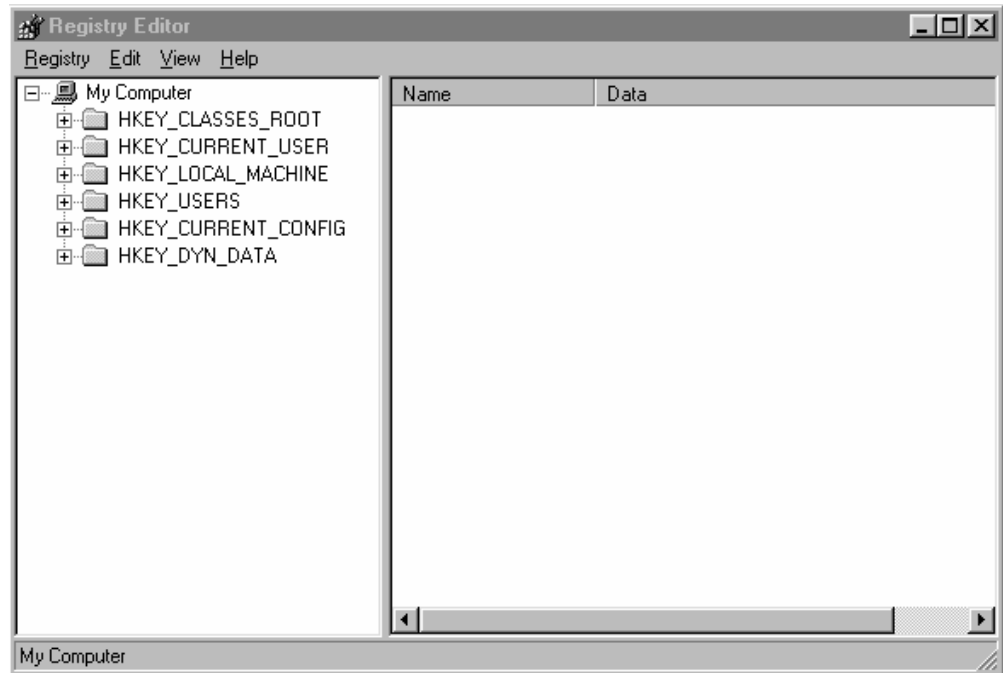
Fortunately VB has a command set that allows us to store and retrieve data using the registry.

18.1 Digging into the registry

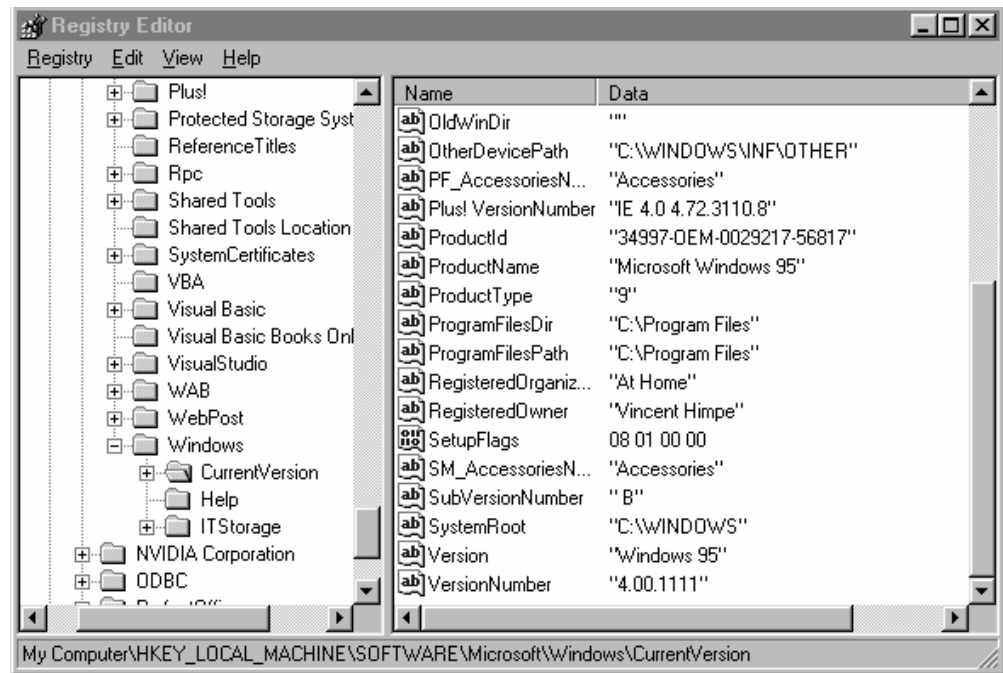
You can have a look at the registry by launching the Regedit program. This is a hidden program inside Windows that allows you to view and manipulate the registry. In order start it you have to follow the following procedure :

- Click on the Start button
- Select Run
- Type Regedit and click OK

This starts up the Registry editor. You will get the basic screen from Regedit



If you want to find something you need to know the keyname for the entry. The registered user of the operating system can be found using the RegisteredOwner key. So if you click on <Edit> <Find> and type RegisteredOwner it will jump you to the location where this information is stored.



As you can see, numerous other information can be found here. The question is what can we do with it? Well not much really. The keys already in the registry belong to other programs. You can check for certain keys to verify if certain programs are installed. You could for instance create a program that requires you to have Excel installed on the computer. During installation you could check for the registry key for Excel. If it was not found in the registry then you could prompt the user that your program explicitly needs excel in order to run.

18.2 Data Mining in the registry

Now that we have a basic understanding of the registry it is time to start sniffing around. So far we have used the Regedit program, but we can do this from Visual Basic as well.

18.2.1 GetSetting

This returns a key setting value from an application's entry in the Windows registry.

GetSetting(*appname*, *section*, *key*[, *default*])

Part	Description
Appname	String expression containing the name of the application or project whose key setting is requested.
Section	String expression containing the name of the section where the key setting is found.
Key	String expression containing the name of the key setting to return.
Default	Optional. Expression containing the value to return if no value is set in the key setting. If omitted, default is assumed to be a zero-length string ("").

If any of the items named in the GetSetting arguments do not exist, GetSetting returns the value of default.

18.2.2 SaveSetting

Saves or creates an application entry in the Windows registry.

SaveSetting *appname*, *section*, *key*, *setting*

Part	Description
Appname	String expression containing the name of the application or project to which the setting applies.
Section	String expression containing the name of the section where the key setting is being saved.
Key	String expression containing the name of the key setting being saved.
Setting	Expression containing the value that key is being set to.

An error occurs if the key setting can't be saved for any reason.

18.2.3 DeleteSetting

Deletes a section or key setting from an application's entry in the Windows registry.

DeleteSetting *appname*, *section*[, *key*]

Part	Description
Appname	String expression containing the name of the application or project to which the section or key setting applies.
Section	String expression containing the name of the section where the key setting is being deleted. If only appname and section are provided, the specified section is deleted along with all related key settings.
Key	Optional. String expression containing the name of the key setting being deleted.

If all arguments are provided, the specified key setting is deleted. However, the DeleteSetting statement does nothing if the specified section or key setting does not exist. Use this command with extreme caution. Don't start deleting at random or you could be faced with the blue screen of death (General protection failure) and an inoperative computer pretty soon.

18.3 Make use of the registry

You can store your own program settings inside the registry. This can be useful to store user settings , last accessed file lists etc. Another useful thing is the window size and position last used. When the program is restarted later it will always appear at the last coordinates. Since you can specify a default value it will work even the first time the program is started.

```
Private Sub Form_Load( )  
    Me.Left = GetSetting(App.Title,  
    "Settings", _
```

```

                                "MainLeft", 1000)
    Me.Top      = GetSetting(App.Title,
"Settings", _
                                "MainTop", 1000)
    Me.Width    = GetSetting(App.Title,
"Settings", _
                                "MainWidth", 6500)
    Me.Height   = GetSetting(App.Title,
"Settings", _
                                "MainHeight", 6500)
End Sub

```

The above code will store the relevant information into the windows registry.

You will not that it makes use of the App object. The App object is an object that returns relevant information about the program. You can retrieve the application name , path to the program , check if another copy of it is running etc. For more information about it you should check the Visual basic Help File about the App object.

The following code takes care of saving the information upon exiting the program. If the program is currently minimized it does not store the information.

```

Private Sub Form_Unload(Cancel As Integer)
    Dim i As Integer
    If Me.WindowState <> vbMinimized Then
        SaveSetting App.Title, "Settings",
"MainLeft", _
                Me.Left
        SaveSetting App.Title, "Settings",
"MainTop", _
                Me.Top
        SaveSetting App.Title, "Settings",
"MainWidth", _
                Me.Width
        SaveSetting App.Title, "Settings",
"MainHeight", _
                Me.Height
    End If
End Sub

```



One last warning might be in place here. Never ever fiddle with the Registry when you don't know exactly what you are doing.

When developing program that access the registry it is a good idea to make sure you have a safe copy of the registry. To create this simply fire up the registry editor and Select <File><Export>. Typically I call this regback.txt and put it in the root directory of my boot disk. When something goes wrong and the registry gets corrupted you can reinstall this safe copy. To do this you have to run the Regedit program under DOS. There you can specify it to import this data and recover the registry.

Note

Anything installed or modified to this registry after you took the backup will be lost

Chapter 19 :

Scripting interpreters

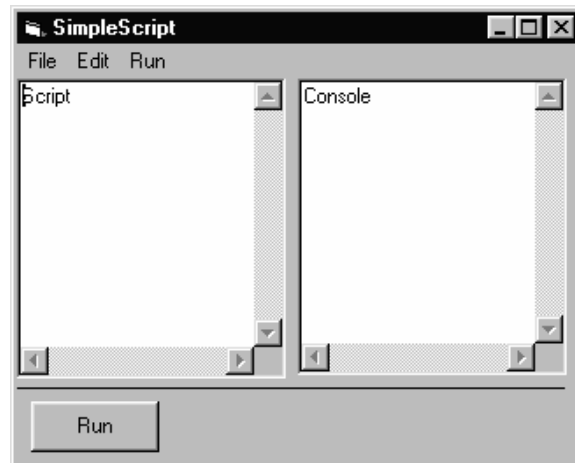
Often you will write programs that control a system and you want to give the user some means of further automation. You can do this by writing a script engine or you can even give your user access to the VbScript engine . VbScript is a real programming language not unlike Visual basic for applications (VBA for short) that can be found in numerous Microsoft programs.

There is a difference between a scripting and a programming language. A script runs top to bottom and has no constructs like loops , subroutines etc.

19.1 Building A simple script interpreter

When you don't need a programming language but only want to give the user the possibility to record a sequence of actions ,and recall them to run the sequence again you can work with a script interpreter.

Typically the first thing you need is a place where the user can write and edit a script. You should also provide routines to read and save scripts. You can easily build this based on our little text editor from Part I.



I just added a menu and a button that allows to start the execution of the code. Also a second textbox was added. This one will act as the output window for the script engine. Generally when people are developing scripts, things will go wrong. So it is a good idea to store the script to a temporary file during execution. The file gets deleted upon termination. If your program crashes you can simply restore this file yourself, or you could make the scripting engine smart enough to do an automatic recovery. So let's attach some code that will create this temporary file and recover for us.

```
Private Sub Form_Load( )
On Error GoTo NoRecovery
    Script.Text = " "
    Open "tmpscript.scr" For Input As #1
    Script.Text = "' Recovered script:" + vbCrLf
    While Not EOF(1)
        Line Input #1, a$
        Script.Text = Script.Text + a$ + vbCrLf
    Wend
NoRecovery:
    Close
End Sub
```

Upon execution of the program it attempts to open the tmpscript.scr file if this fails we know that the previous run did not crash. If there is such a file it will be loaded into the script window.

19.1.1 Running the script

```
Private Sub Fire_Click()  
    tmpfile = FreeFile  
  
    ' store script to temporary file  
    Open "tmpscript.scr" For Output As #tmpfile  
    Print #tmpfile, Script.Text  
    Close #tmpfile  
  
    ' load and start interpreting  
    Open "tmpscript.scr" For Input As #tmpfile  
    While Not EOF(tmpfile)  
        ' read script line by line and  
        interpret  
        Line Input #tmpfile, commandline$  
        ' clean up the input  
        commandline$ = Trim$(commandline$) + "  
"  
        ' check for comment  
        If Left$(commandline$, 1) = "'" Then  
            Else  
                ' execute command  
            End If  
        Wend  
    Close #tmpfile  
    Kill "tmpscript.scr"  
End Sub
```

The above code is attached to the Fire button. The command Run from the menu simply calls the Fire_Click method to invoke the execution. Upon activating the engine the script gets written to the temporary file. Then the engine reopens the file and , as long as the end has not been reached, reads it line by line.

This is the basis for the interpreter. The next thing we need to do is clean up the read line so it does not contain any unwanted stuff. After all the user of the engine might decide to modify it manually with other programs. A good point is to remove all leading and trailing spaces , and that's exactly what the next piece

of code does. The script engine will analyze the code and try to extract blocks of text. But suppose we the user has entered an empty line ! This could lead the script engine to crash. So we will add two dummy spaces at the end of each command. It will come clear later why exactly 2 spaces. The next thing the code does is checking if the line begins with an apostrophe ('). If it is the case it will not be passed to the script engine since we defined this character as the comment character.

Upon completion the temporary file is deleted using the KILL command. If the script engine crashes this file will not be deleted and the recovery routine in the Form_Load will pick it up. Now that all this preliminary work has been done we can concentrate on the real engine : The Parser.

19.1.2 The script Parser

This is the real engine that will determine what commands and arguments , if any , are present in the line and will invoke the appropriate code. First we need a way to extract the command. This engine is based on following criteria

- one command per line
- unlimited amount of arguments per line
- commands and arguments separated by a fixed character (a space.)
- comment is preceded by an apostrophe (')

We now we need to extract the command from the command line . Since our script language dictates that all commands and optional parameter should be separated by a space this is quite easy

```
Ecmd = InStr(commandline$, " ")  
cmd$ = Left$(commandline$, Ecmd)  
  
' Now determine the string with arguments  
argument$ = Right$(commandline$,  
Len(commandline) - Ecmd)  
  
' clean up the junk and convert cmd$ to  
uppercase  
cmd$ = trim$(UCase$(cmd$))  
' now we are ready to parse the commands
```

Depending on what functions you want to make available you can modify the script engine. At least you should give the user the possibility to see the script running and that is exactly why I put the second textbox on the screen.

It would be nice if the user could manipulate this console. It would be very useful if the user could, at least, add text to the console and clear the console

```
Select Case cmd$  
  Case "QUIT", "END", "BYE"  
    End  
  Case "CLS"  
    Console.Text = ""  
  Case "PRINT"  
    Console.Text = Console.Text + argument$ +  
vbCrLf  
End Select
```

A MessageBox would come in handy too. It allows the user to stop the script temporarily

```
Case "MESSAGE"  
    MsgBox argument$, , "Message :"
```

Using the same programming logic you can add instructions yourself. Since we are using a general Select Case system as a parser it is easy to allow multiple possibilities for one command. As you can see I already created a command that allows you to terminate the program

Now that these basic things are out of the way we can start to implement the real instruction set. Instruction typically requires data input of some sort. So we will need a routine that can extract the relevant parameters from the argument\$.

19.1.3 Parameter extraction.

Depending on the command the user might have passed one or more parameters. In order to cope with this you would have to write a routine for each possible case. But there is an easier way. If you make a routine that can extract one parameter at a time from the parameter string we could call it the number of times we are expecting parameters. Furthermore the routine could warn the user if he has forgotten one or more of them.

```
Function GetArgument$(ByRef argument$)
    tmp$ = Trim$(argument$)
    If Len(tmp$) = 0 Then
        MsgBox "Error in Script : Missing
parameter "
    Else
        tmp$ = tmp$ + " "
        x = InStr(tmp$, " ")
        ' extract argument and return value
        GetArgument$ = Trim$(Left$(argument$, x))
        ' delete argument from argument string
        argument$ = Right$(argument$,
Len(argument$) - x)
    End If
End Function
```

Let's make two script commands ADD2 and ADD3. ADD2 will add two arguments together. ADD3 will add three arguments. This will demonstrate the use of the GetArgument\$ function

```
Case "ADD2", "ADD"
    a = Val(GetArgument$(argument$))
    b = Val(GetArgument$(argument$))
    c = b + a
    entry "Result = " + Str$(c)
Case "ADD3"
    a = Val(GetArgument$(argument$))
    b = Val(GetArgument$(argument$))
    c = Val(GetArgument$(argument$))
    d = c + b + a
```

```
entry "Result = " + Str$(d)
```

The Entry routine is a simple routine that can write a string to the console.

```
Sub entry(txt$)
    Console.Text = Console.Text + txt$ + vbCrLf
End Sub
```

The above code demonstrates the basic creation of commands and the handling of the arguments. However, this is by far not the end of what is possible. If you want a routine which can take an undetermined number of arguments you could do the following :

```
Case "ADDX"
    While Len(Trim$(argument$)) > 0
        x = x + Val(GetArgument$(argument$))
    Wend
    entry "Result =" + Str$(x)
```

To test all of the above you can try out this little script :

```
CLS
PRINT This is my first script
ADD2 5 6
ADD2 7 8 9
ADD3 7 8 9
PRINT the following command will produce a
SCRIPT error
ADD3 5 6
' The following demonstrates the ADDX command
ADDX 1 2 3 4 5 6 7 8 9
Message The system will now crash to demonstrate
recovery
crash
END
```



As you can see scripting is a very powerful tool to embed in your applications. It allows you to automate frequently used tasks. When I build big test systems I create a program that allows the user to manipulate every machine and system in it. The sequences that need to be executed to perform the actual measurement are not hard coded but embedded in scripts. This allows the user of the system to modify at will, without me needing to revise the program over and over again. More, it hides all the down-to-earth stuff from the people operating the system. They don't need to know how to set up and acquire data from a certain instrument. No, they simply write in the script `READVOLTAGE`, and the system will control the appropriate instrument, retrieve the result and dump it to the console.

As stated before this home-built script engine is NOT a programming language. If you need features like looping, jumps etc., you will need a real engine.

19.2 MSScript : A real script interpreter.

Visual Basic allows you to embed the MSscript engine inside your programs. This is a very powerful tool not unlike Visual Basic For Applications (VBA). There is one small problem with this interpreter: It's not installed on every machine. Furthermore there is no help for it.

It is very well possible that you can't try out the following. You can install the MSScript engine from the Microsoft Windows SDK toolkit. These tools are also available in the Professional editions of Visual Studio.

19.2.1 Scripting language

The language used by MSscript is VbScript. However , not all functionality of Visual Basic is embedded in VbScript.

➤ **Variables :**

One big difference is that here there is only the Variant data type. So there is no need at all for the DIM command except for creating arrays.

➤ **Objects :**

Only objects exposed to VbScript explicitly are accessible. Besides these VbScript only knows the ERR and dictionary objects. Dictionary is an object that stores key and data values.

When you switch UseSafeSubset to False then you get access to additional objects that allow you to do file manipulations. However , I strongly suggest that you DON'T do that. Handle all file manipulations in the program where VbScript has been embedded.

19.2.2 The MSscript properties

Msscript has a number of properties that allow you to specify its behavior. The most important are listed below.

AllowUI	When set to True Msscript can display objects like Messageboxes etc.
CodeObject	Returns you the exposed objects
Error	Returns the information about the scripting error
Language	You can set this to either VBscript or Jscript (Java)
Modules	Contains a collection of Modules
Procedures	Contains a collection of Procedures
Timeout	Allows you to specify the maximum time the script will run before it aborts
UseSafeSubset	Prevents access to security critical objects like files and disks.

19.2.3 Script Control Methods

The following are the Methods embedded in the script control

AddCode	Allows you to send code to the script control
AddObject	Allows you to expose an object to the Script Control
Eval	Allows you to evaluate an Expression

ExecuteStatement	Execute a single statement
Reset	reset the script engine
Run	execute a subroutine

The Eval method is very interesting. It allows to evaluate mathematical expressions. If you execute the following code you will get the result for the calculation

```
X = ScriptControl.Eval " Sin (1+(3/4))
```

The script control will return the Sinus of 1 and $\frac{3}{4}$. You can use this to evaluate user entered mathematical expressions in your program.

19.2.4 Adding code to the script engine

This is only a matter of calling the AddCode method. This method is automatically checking the syntax of the transmitted source code. If there is an error you will be notified. So make sure you write an error handler. The error handler should check the Script's error object and not the one from the main program

```
Private Sub Sendcode_Click()  
    On Error GoTo scripterror  
    ScriptControl1.AddCode Text1.Text  
    Exit Sub  
scripterror:  
    MsgBox "Error line " +  
    Str$(scriptcontrol1.Error.Line)_  
    + VbCrLf + ":" + scriptcontrol1.Error.Text _  
    + VbCrLf + "scriptcontrol1.Error.Description  
    Scriptcontrol1.Error.Clear  
End Sub
```

The above code will display the line number , the contents of the line , and the description of the error.

19.2.5 Exposing Objects

You can give VbScript access to any object inside the program where you use the VbScript. All you have to do is expose this object to VbScript. Suppose you have a Label called Display and you want to be able to control this from the Script

```
ScriptControl1.AddObject "display", Display
```

The above code will do the trick

In the script you can then simply write

```
Display.Caption="Hello"
```

You can also expose functions and procedures inside your program for activation by the script. However this is not straightforward. You need to create a Class module and embed the functions in there. Inside the program you can then create a new object from this class and add this to the objects exposed to

the Script engine . This will be explained in detail in the example on VbScript in appendix III.

Chapter 20 :

Classes

Since we are working in an OOP environment we have to know a bit about classes. What exactly is this concept of a Class.

20.1 The Class concept

Remember the Controls we put on a form. ? Yes ? Good ! Well these are actually instantiated classes. Just like an Object or control has properties , methods and events , A class can have all of these.

So you could consider a class as an object or vice versa . The nice thing about classes is that you can treat them as objects. You can define a new variable based on a class. Confused ? Perfectly normal.

If you put a control on the screen, let's say a label. You give this the name 'Display'. Well from now on you can access the properties for this object by the name 'display'. You could think of this as a variable. The property 'Caption' is embedded into the Object Label. This means that the variable 'Property' belongs to the Class 'Label'. The same goes for the Move method. You can apply the move method to the label. Well , the 'Move' method belongs to the class 'Label'.

Then what is the difference between Classes and Objects (controls). Simple :Nothing. Except maybe that Classes have no visual substance (user interface on screen) . Take for instance the Printer object. This is the perfect example of a Class. It is not visible , has no substance yet you can activate methods , set properties and the printer object can raise events.

Classes are a construction to make programming more structured and augment the manageability of large and complex programs.

20.2 Creating a Class

You start the process of creating a class by selecting Add class module from the Project menu. This will create a new class in your project.

Once you have this you can start creating properties , methods and events. Creating Methods is nothing else then writing Public subroutines and Functions. Properties have to be defined using the Property command

Example :

```
' userfunction class

Private c_msg$ ' internal storage for msg$

Public Sub Dosomething( )
    MsgBox "I did it" + c_msg$
End Sub

Public Property Let message(msg As String)
    c_message$ = msg
End Property
```

The above piece of code shows you how to create a method (DoSomething) and a Property (Message). Note that for the moment you can only assign something to the property. In order to retrieve the data from the property we need to write a 'Property Get' handler.

```
Public Property Get message( ) as variant  
    Message = c_msg$  
End Property
```

You also need to allocate storage space to hold the data assigned to a property internally. As you should remember, once a subroutine exits the internal data is destroyed. Therefore you need to declare a variable at module, or in this case Class, level. You can also declare private subroutines inside your class. These can be accessed from inside the class but are invisible to the user of the class.

Adding an Event is as easy anything else. Just write the appropriate code for it.

```
' Declare the event  
Public Event YO(ByVal text as string)  
  
' activate it ( this should be in a procedure or  
function )  
RaiseEvent YO( " Yo Dude" )
```

That's it. You just created a class.

20.3 Instantiating objects from a class.

This is equally simple. There are two ways to do this, however only one will unleash the full potential of the class modules.

You can either use DIM to declare a new object that will be derived from a class. By the way that is the correct terminology to say that you want to access a class


```
Dim User as New Userclass  
Dim SecondUser as New UserClass  
Dim Userlist(50) as New userclass
```

The second method is by declaring a new variable of type Object and then assigning it to a class.

```
Dim User as object  
Set User = New UserClass
```

The difference here is that the object is not actually created with the Dim statement. Only when it gets assigned to the Userclass then it gets created. This conserves memory.

Note

You can instantiate as many objects from a class as you want.

20.4 A practical example

```
` Class Yelling  
  
Private c_msg$ ` internal storage for msg$  
  
Public Sub YO()  
    MsgBox c_msg$ ,, "Yo dude"  
    RaiseEvent Yell(" Yelling completed")
```

```
End Sub  
  
Public Property Let message(msg As String)  
    c_msg$ = msg  
End Property  
  
Public Property Get message()  
    message = c_msg$  
End Property  
  
Public Event Yell(ByVal text as string)
```

This is the end of the class

```
Private WithEvents shout As Yelling  
  
Sub Form_Load ()  
    Set shout = New Yelling  
  
    ' assign text to property Message  
    Shout.Message = "Hello World !"  
    ' Activate YO method  
    Shout.YO  
End sub  
  
Sub shout_Yell(byval txt as string)  
    Debug.print txt  
End sub
```

And this is the end of the main code. You will notice that here I have used yet another syntax to instantiate an object derived from the class. The reason is that I want to attach to the events generated by the class. If I don't explicitly specify, using WithEvents, that I want to have access to the events, then I don't get them.

Furthermore if I create my object this way, Visual Basic will know all the properties and methods of the class. This means that from then on VB will assist

my during my code writing just as it would with any other of it's objects. It will display the nice listbox with all objects and properties the moment it type the dot.

Yet More Case studies

This section will show some examples of programs that apply the techniques described in this part.

Killing windows via an API call

This little program shows you how to embed an API call in your program

The LED activeX control

I always wanted a simple indicator on my screen. Well here it is.

The PassBox activeX control

A simple control that allows you to enter a password , and that displays start instead of the characters you type.

MiniBasic : A program editor for MSscript

This is the basis for a script interpreter based on VbScript. You can expose objects AND your own procedures to the VbScript engine

Additional Notes on the use of classes

This is not a case study but a proposal for practical use of classes.

Case Study 7 : Killing Windows via an API call

Suppose you need to build a 'closed' system. A kind of setup where the user can only fire up the computer , do his thing , and then only can shutdown the system. It might be neat if you could do this directly from the application.

Since you can do this from the Start button of windows , it means it must be accessible somewhere. And indeed , this is an API call just as any other.

Shutdown.bas:

```
Public const ForceExit = 4
Public const Logoff = 0
Public const reboot = 2
Public const Shutdown = 1
Public declare function exitwindowsex lib
"user32" _
(byval uFlags as long , Byval dwReserved as long
) as long
```

Main form :

The form is very simple : only one command button.

>image

```
Private sub Command1.click
    X = exitwindowsex (logoff,0)
End sub
```

If you hit the button it will log you off of your current session. If you want to kill the operating system simply replace logoff with Forceexit.

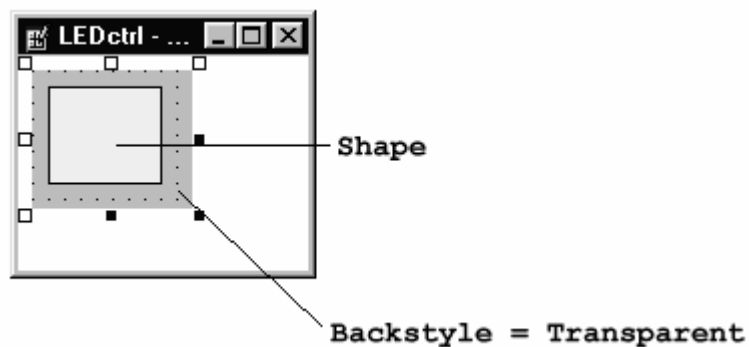
Ehm ... Did I tell you to save before trying this ? No ! .. Oh I'm Sorry , you were supposed to save it .☺.

Case Study 8 : The LED ActiveX control

This example will show you how to build a simple ActiveX control. You start as usual with a normal Project. Now you add a new project of the type ActiveX Control.

To make our life easy I will use a shape control from the toolbar as indicator for our led. I set the property backstyle of the control form to Transparent. This will assure that only the shape is visible.

The shape is simply called shape1. The control should look like this :



Since I want to allow the user to size the led , I have to attach some code to the resize event.

```
Private Sub UserControl_Resize()  
    Shape1.Left = 0  
    Shape1.Top = 0  
    Shape1.Width = UserControl.Width  
    Shape1.Height = UserControl.Height  
End Sub
```

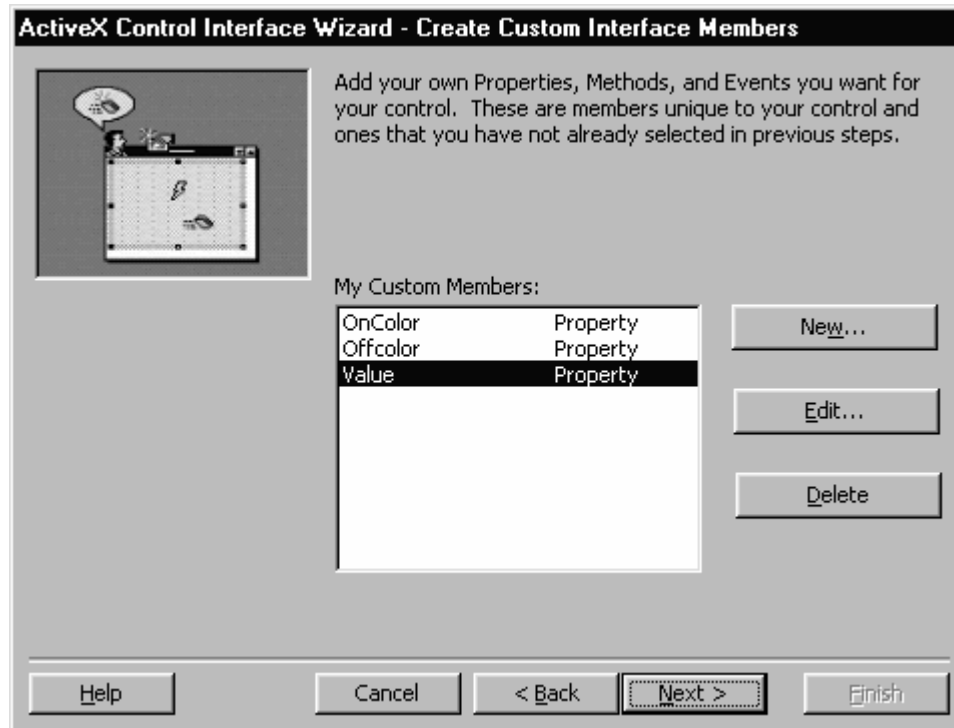

Whenever the size of the led changes ,the shape will size with it to fill the entire boundary.

Since I want to start attaching some properties to the Led it's time to fire up the ActiveX Control Interface Wizard.

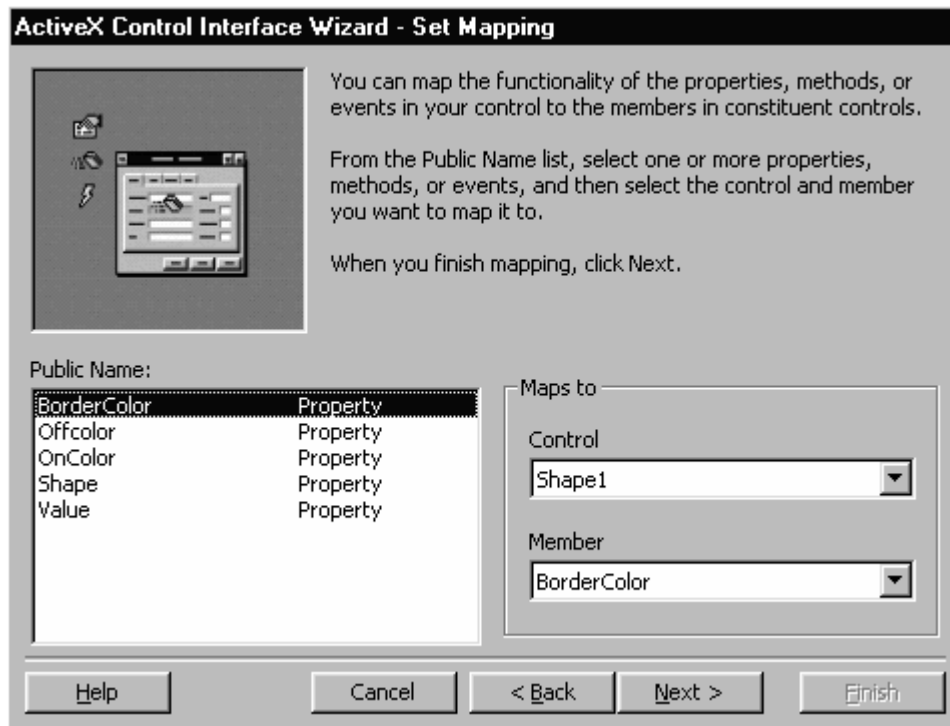
In this wizard is took the properties Shape , and Bordercolor from the standard properties selector.



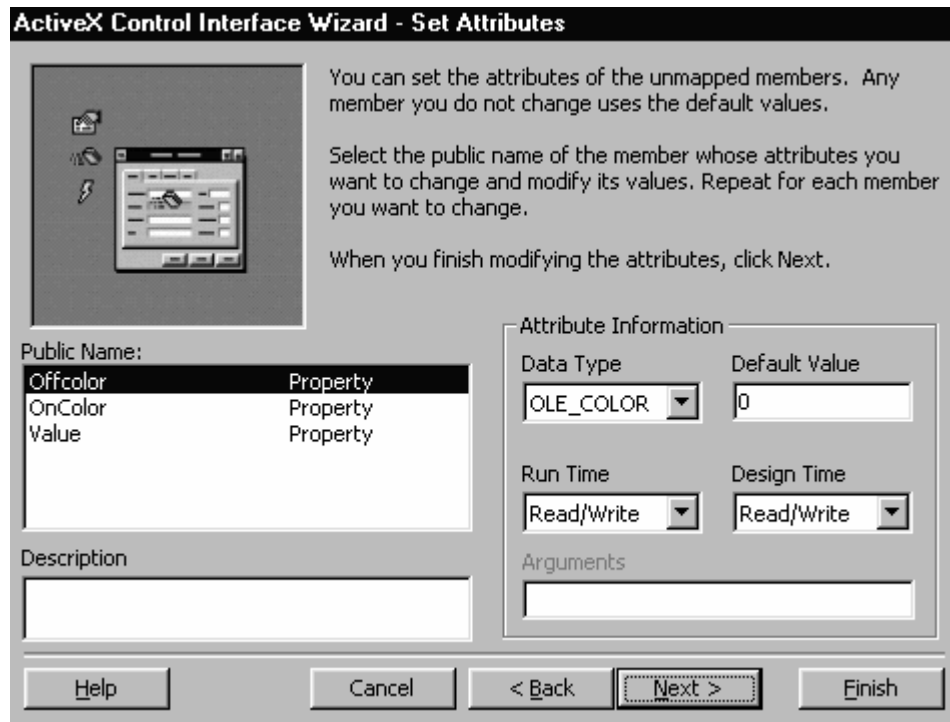
The next step that has to be done is creating additional properties. I want to specify the OnColor and Offcolor and the value of the led. A value of 1 means that the led will be On , and similar , 0 will be off.



The next thing I have to do is specify which of my defined properties are mapped to which properties of the existing objects in my control.



As I can directly map **Bordercolor** and **Shape** to the Shape1 object , I do so. This saves me from having to write all the code for this. For the other properties I don't specify anything. This mapping is done in the next screen.



I define the Offcolor and Oncolor properties as type OLE_color . This means that in my property window I will automatically get a color selection tool to specify the appropriate colors. The value gets specified as type variant.

Now that all of this user interface stuff is out of the way I can click on finish and the wizard will write all the necessary code for me.

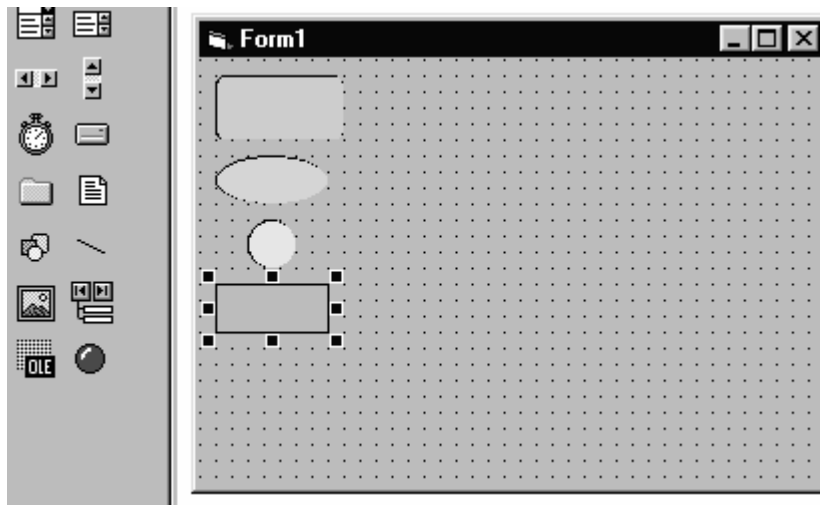
The only thing I have to write is the piece of code that changes the color according to the setting of the Value property. So I dig up the correct routine and write the necessary code.

```
Public Property Let Value(ByVal New_Value As
Variant)
    m_Value = New_Value
    If m_Value = 1 Then
        Shape1.FillColor = m_OnColor
    Else
        Shape1.FillColor = m_Offcolor
```

```
End If  
PropertyChanged "Value"  
End Property
```

The only lines I have to write is everything between If and End IF. That's it . control Ready ! This is the basic control that will perfectly perform what I intended it to . But we can do better. I will assign a bitmap to the control . this means that this bitmap will appear in the object browser of the Visual Basic. To do this I select a simple bitmap and assign it to the ToolboxBitmap property of my control form.

I can now put some of my objects on the main form of the second project in the group. The next picture shows a screenshot of the control browser with a demo from that contains 4 copies of my Led control



And now you think this is finished. No way. If you take a look at the Shape property you see that it only contains a numerical field. I would like to see a little pulldown menu there with the possible shapes on it. Enumerating a variable can do this.

Enumerating is something I have not explained yet. IT is the process in where you define a variable and assign it a list with possible values and a textual description. You could for instance make a variable DayOfTheWeek. Then you

would enumerate it as '1 Saturday, 2 Sunday, 3 Monday ... etc . Furthermore you can set the variable to the day of the week by specifying simply the name.

If could do the following.

```
DayOfTheWeek = Tuesday.
```

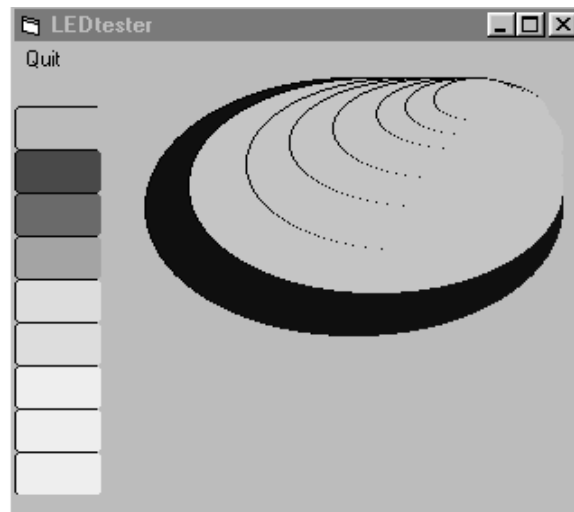
If I print DayOfTheWeek it would return me 5 !.

Declaring the variable with the Enum keyword does enumerating.

```
Public Enum shapetypes  
    rectangle = 0  
    Square = 1  
    Oval = 2  
    Round = 3  
    Roundedrectangle = 4  
    RoundedSquare = 5  
End Enum
```

When this is done all I have to do is change the declaration of the Shape property. Now it is defined as Integer. When I change it to shapetypes it will be enumerated according to that list. And presto. A fully working , professional looking control.

So now I can make a little program with it. You can find this in the LED directory of the CD-ROM. All the program does is generate a number of random values and display them in bar graphs made from my LED's.



The program uses two control arrays. On an interval basis specified by a timer it generates two random numbers and creates two bar graphs. The left one is an ordinary bar graph. The right one is a Funky 3D elliptic style stacked bar graph.

Once you are satisfied with the look and feel of the control you can ask Visual basic to compile it into an appropriate OCX file . From that moment on you can use your control in any program as a separate object. You can even give , or better : sell , it to third parties.

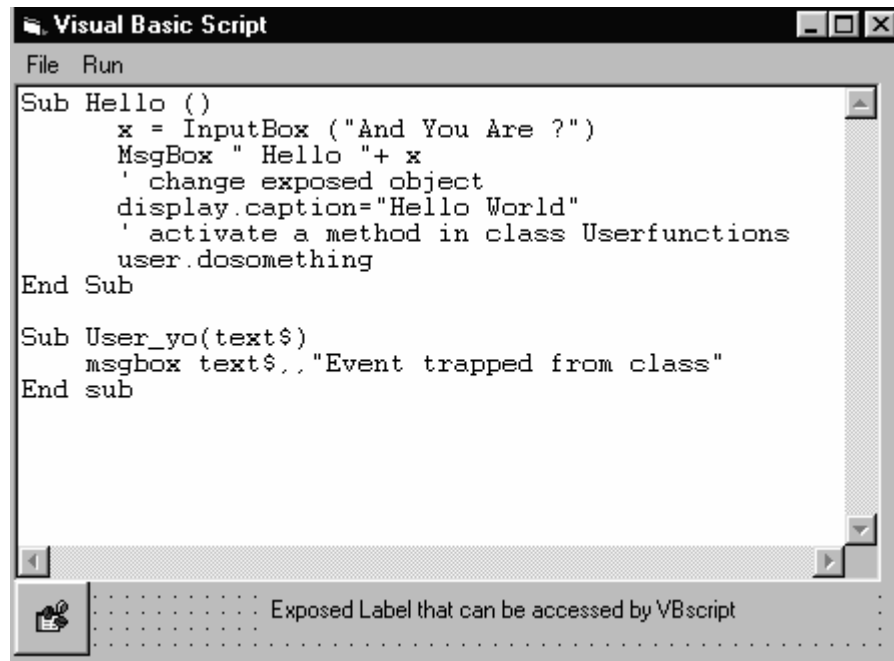
Case Study 9 : MiniBasic : A program environment for MSscript

This is by far not a full-fledged programming environment such as Visual basic. But nonetheless it gives an idea on how to embed scripting into your programs.

Let's start with the usual stuff. A form with a textbox called Script , a menu with a File-Quit and a Run Menu. On the run menu I want to put the Run , Eval and Main entries.

Run will execute the program , Eval will prompt me for an expression and evaluate it , and Main will allow me to specify the startup routine for the Script code.

Furthermore I need to insert the MSscript control as well. Since it is not loaded into the control browser by default I have to enable it using the right click and selecting Customize. There I will see a reference the MSscript , simply check it and I'm up and running.



I also inserted a simple label in order to be able to demonstrate the use of object exposure to the script. As you can see I set the Multiline and scrollbar features of the textbox on and already entered a piece of code into the textbox.

The next thing I need to do is write some code for the user interface.

```
Private Sub Form_Load()  
    ScriptControl1.AddObject "display", Display  
    Dim userclass As Object  
    Set userclass = New Userfunctions  
    ScriptControl1.AddObject "User", userclass  
End Sub
```

The Form_load procedure exposes the label 'Display' to the script engine. This means that from now on the script engine can access all the objects properties , methods and events.

Since I want to give the Script engine access to some routines I have defined in my program I create a new object of the Class Userclass and then add it to the script engine just as I did with the label.

```
Private Sub runprogram_Click()  
    On Error GoTo scripterror  
    ScriptControl1.AddCode Text1.text  
    If Scriptmain <> "" Then  
        ScriptControl1.Run Scriptmain  
    Else  
        MsgBox "Specify a Main routine first " +  
vbCrLf + "Doubleclick the routine and select  
<Run> Set Main"  
    End If  
    Exit Sub  
scripterror:  
    MsgBox "Error line " +  
Str$(ScriptControl1.Error.Line) _  
    + vbCrLf + ":" + ScriptControl1.Error.text _  
    + vbCrLf + ScriptControl1.Error.Description,  
vbExclamation, "Script Error !"  
    ScriptControl1.Error.Clear  
End Sub
```

The RunProgram procedure will copy the contents of the textbox to the script engine. If during the syntax check an error occurs , the error handler will retrieve the line number , the string with the error in it and the description.

It will also check if I have specified the startup routine. If not the program will not be executed.

```
Private Sub quitfile_Click()  
    End  
End Sub  
  
Private Sub setmain_Click()  
    setmain.Caption = "MAIN :" + Text1.SelText  
    Scriptmain = Trim$(Text1.SelText)  
End Sub
```

In order to be able to expose user modules a created a class called Userfunctions. From this class I derived the Userclass object in the Form_Load. This object is then exposed to the Vbscript engine as the User object by issuing the AddObject method. As you can see below.

```
Private Sub Form_Load()  
    ScriptControl1.AddObject "display", Display  
    Dim userclass As Object  
    Set userclass = New Userfunctions  
    ScriptControl1.AddObject "User", userclass  
End Sub
```

In this class I defined some modules that are of general use

```
' userfunction class
' put here calls to expose to VBscript

Dim c_message$
Dim tmp

Public Sub Dosomething()
    MsgBox "I did it" + c_message$, , "Activated
Class"
End Sub

Public Property Let message(msg As String)
    c_message$ = msg
End Property
```

The above class contains a property message and a method Dosomething. Since they are exposed to VbScript as the User object you can access these items as User.DoSomething and User.Message . This code is of course expandable as far as you want. Furthermore you could create as many extra classes as you like. Each of these could be added to the VbScript engine as a new object.

Case Study 10 : Additional notes on the use of Classes.

Classes can be very useful to define high level blocks of code. The end user need not know what exactly is going on but can work with an abstract object. It is sufficient that he knows the properties , methods and events associated with that class.

Suppose you have for instance you have two machines that have a similar function but a different interface. You could define a class for each , let's call them Machine1 and Machine2. Both classes contain the same names for the methods, events and properties. However , the internal code is completely different. You could make a program that controls the machines without having to bother with the actual code for the machines. You just use an object created from their class. When you want to use the other machine all you have to do is derive the object from the other class.

Example

```
' Class for a machine from Vtronix

Public Event Overrange( )

Private C_x
Public Property Get Measure( ) As String
    Measure = "V = " + Str$(C_x) + " :Vtronix"
End Property

Public Property Let Range(x As String)
    C_x = x
    If (x > 100) Then RaiseEvent Overrange
End Property
```

The above class would be the definition of a imaginative machine from the company Vtronix. This machine can be set to a certain range using the Range property. And can return Results using the Measure property. In case of an over-range the machine will raise the Overrange event.

The next class defines a similar machine from the company Hvsystems

```

' Class for a machine from HVsystems

Public Event Overrange( )

Private C_x
Public Property Get Measure( ) As String
    Measure = "V = " + Str$(C_x) + " :HVsystems"
End Property

Public Property Let Range(x As String)
    C_x = x
    If (x > 10) Then RaiseEvent Overrange
End Property

```

All we need to do in the program is instantiate the instruments from the proper class.

```

Private WithEvents voltmeter1 As Vtronix
Private WithEvents voltmeter2 As HVsystems

Private Sub Command1_Click( )
    voltmeter1.Range = 100
    voltmeter2.Range = 10
End Sub

Private Sub Command2_Click( )
    voltmeter1.Range = 10
    voltmeter2.Range = 100
End Sub

Private Sub Command3_Click( )
    Label1.Caption = voltmeter1.Measure
    Label2.Caption = voltmeter2.Measure
End Sub

Private Sub Form_Load( )
    Set voltmeter1 = New Vtronix
    Set voltmeter2 = New HVsystems
End Sub

```

```
Private Sub Quit_Click()  
    End  
End Sub  
Private Sub voltmeter1_overrange()  
    MsgBox " Voltmeter 1 in overrange "  
End Sub  
Private Sub voltmeter2_overrange()  
    MsgBox " Voltmeter 2 in overrange "  
End Sub
```

The main form is a simple form with three buttons and 2 labels. Two of the buttons program ranges to the instruments . The third retrieves measurements from the machines.

While this example is pure hypothetical , it shows clearly the use of classes to create objects. This is exactly how the printer object works. Depending on the printer you select the object Printer is derived from another class. If you use the method Print it will print your piece of text to the printer. It does not matter if this is a Laser, an inkjet or a Matrix printer. The code embedded in the class knows how to handle this low level stuff. The user only needs to know that he can print using the Print method of the Printer object.

This kind of class usage is implemented in the GPIB system described later on in this book.

ClassWork is a library of transportable instrument classes. If you need access to a machine you simply create an instance of this class by creating a new variable of the class.

```
Dim Voltmeter as new HP34401  
Voltmeter.address=4  
Voltmeter.Range VoltsDc  
Debug.print Voltmeter.Measure
```

These simple command will derive an instance 'voltmeter' from the HP34401 class, assign a GPIB address and select a range. The last command retrieves a measurement. If tomorrow that machines is not available you can simple derive the object from a different class.

Dim Voltmeter as new FLUKE45

More about this later on.

Visual Basic

For Electronics Engineering Applications

Part IV

Visual Basic in the Lab

Visual Basic

For Electronics Engineering Applications

Part IV :
Visual Basic for the Engineering Lab

Introduction

Well hello , apparently you have made it so far . Or did you skip a lot of stuff ?. No problem. This book was written so you could skip stuff that is of no interest to you at this particular time and place.

Now that you have learned a big deal about the language , how to write and compile programs , talk to other programs and wrap them up for distribution ,create objects , classes and controls and many, many, many more things , it is time to have a look at what it Visual Basic can do in a technical environment.

When I am talking about a lab , and take my word for it : I know the Lab Environment , this can mean anything . A chemistry lab , an electronics lab , a physics lab , even an optics or medical lab. Forgive me if during this chapter , I would appear biased towards the electronics lab. After all I've been working (

and still am) as chief ops for a Research lab of a leading Semiconductor company.

Typical lab work includes controlling a test setup , driving instruments and collecting and processing data. Processing data is something we can do offline with existing tools. Applying Visual Basic for Lab work mainly concentrates on the application of Visual Basic programs to help use control a test setup and acquire data for us. It can automate cyclic tasks , and collect data for us. This data can either be written disk or exposed using other means (a Telnet server for instance).

In order to build such setups we will need , besides the computer and visual basic , a plethora of equipment. This can go from simple switches to complex measurement equipment. Some of this equipment will be connected to the computer , some might be plugged in to the computer. So , to bring the task of automating a test setup or 'bench' to a good end we need to know a bit more about the possibilities of our computer. We need to have a basic understanding of the machine , the standard communication ports at our disposal , and the practical things we can do with them.

Chapter 20 :

The Computer

As this will form the core of your automated test system , we need a basic understanding of this system

20.1 The PC : A Historical Overview

The PC was first conceived as ‘a smart input terminal’ . It was never intended to be used as a standalone machine. When Don Estridge and his team of 13 started this project the goal was to make a small , smart terminal that could run some front-end software. The idea was to unload the big Mainframe computers from the task of serving consoles.

The project was to be an open-architecture low budget kind of thing. External companies did most of the work. They started building the PC using a S100 bus computer board from Intel , a Monitor program previously written for a 6802 CPU from Motorola and a CP/M version for 8086 .

Starting from these parts and some experimenting a final schematic was drawn and the monitor program was extended to become , as we know it today , the BIOS. At that point it became clear that CP/M was not the way to go. IBM had seen a demonstration of an operating system developed by Seattle Computer products. Called QDOS. One of the design team members talked about this to the young Bill Gates. This guy joined the team as an external solution provider. It was agreed that Microsoft would port QDOS to the hardware platform of the PC and then modify an earlier written Basic interpreter (for Tandy corporation) to run on this platform.

When the PC was announced in 1981 PC DOS was its primary operating system. Outside market researchers pointed out that this project was doomed ! . Who would buy a computer that was not attached to a mainframe !.

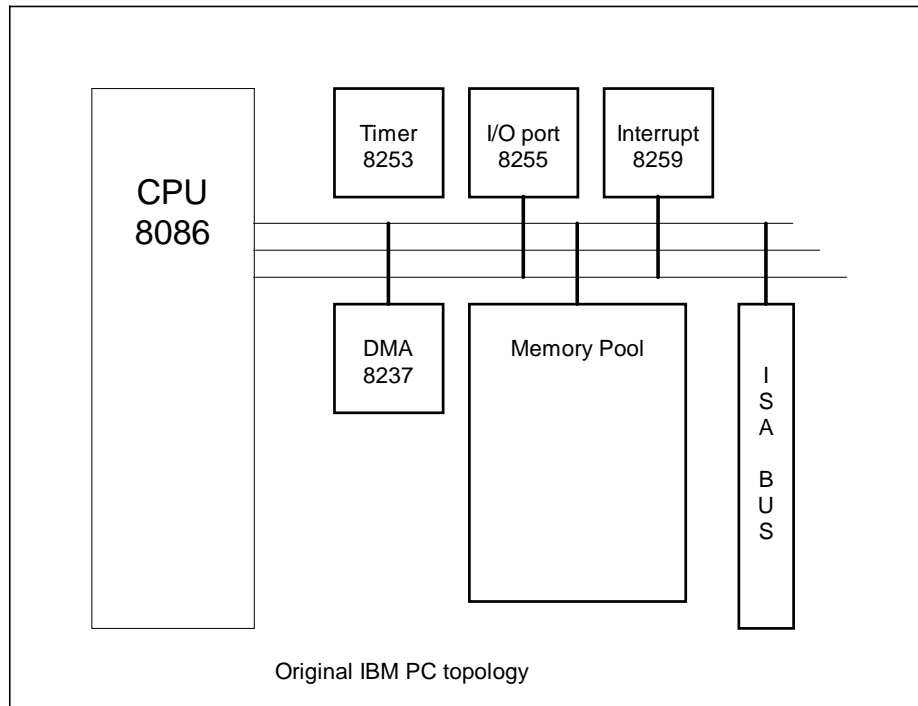
Well it looks like they were wrong . Now , almost 20 years later the situation is the opposite : The mainframe has been moved to the museum a long time ago .Every office holds more computing power then the average mainframe of 1981.

Besides the software the hardware has improved substantially. Where the first machine ran PC DOS 1.0 , today's machines run Windows / Windows NT / UNIX and clones and every other possible operating systems. Literally millions of applications have been developed , both commercial and shareware / freeware.

This machine has set off what has been called 'the computer revolution'

20.2 The PC : A Hardware Description

The original IBM-PC hardware merely consisted of some standard chips that Intel was selling at that time . The block schematic showed the following components :



Besides the CPU , memory and some chips to get the thing running there was nothing else in the machine. Every PC , whether it is an original IBM-5100 from 1979 or the latest state of the art souped-up Dual Pentium-IV Xeon 2.63 GHz with 10 Gbyte of Rambus Memory, still adheres more or less to this topology.

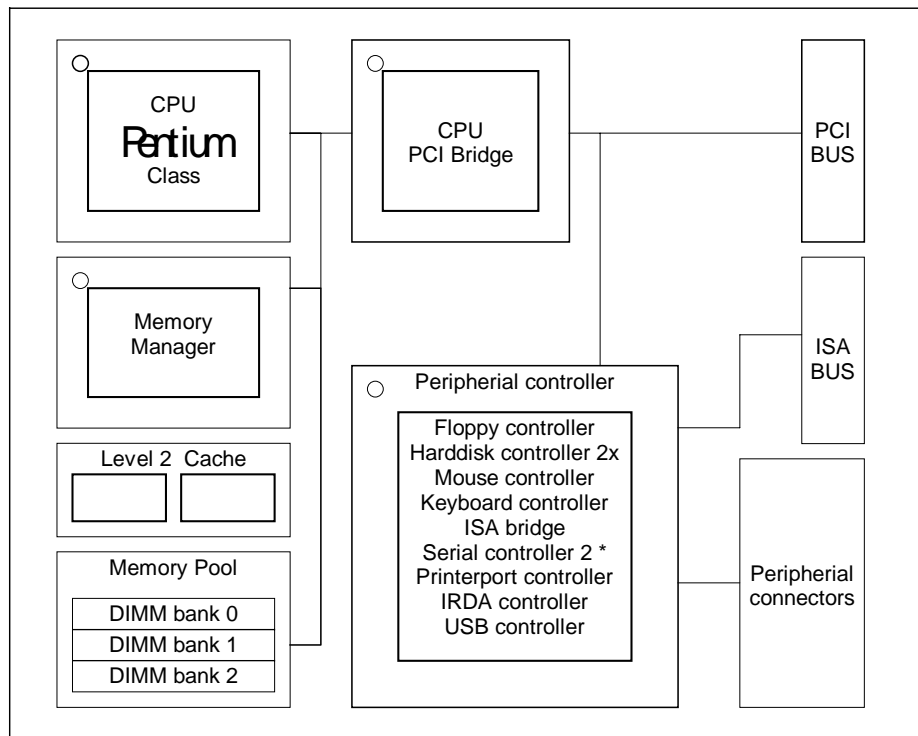
You will still find a 8253 Triple timer , one or two 8259 interrupt controllers and a DMA controller of the 8237 type inside your computer. Maybe they are no longer visible as such but they are in there somewhere. The only things that have really changed in the PC are the speed ,and the width of the data and address bus.

The speed of all components and ,markedly the speed of the CPU , have gone up tremendously. Where the original machine ran at a blazing 4.77 MHz , today's machines easily break the 2 GHz barrier. That is more then a 400 fold performance !.

This has lead to the fact that new techniques had to be developed to cope with the new speed-demons.

Things such as second level and third level cache have been designed. , new buses emerged (VESA Local bus , PCI) , and new communication standards have been set forth (IRDA , PCI ,Fire wire , IEEE1394). But apart from this , the PC looks still the same.

A modern PC block diagram looks somewhat like this. The astonishing fact is that you can translate this directly to a component schematic:



The entire PC has been scaled down to a mere 4 IC's. You still have the CPU which is now of the Pentium - Class . (Pentium / Pentium Pro / Pentium II Pentium III all with or without MMX) . Besides this you need 3 more components to build a computer : The memory controller. This component is a single chip that handles all accesses from the CPU to the memory and AGP bus. It takes care of refresh cycles , cache update , and so on. This chip is often referred to as the 'North Bridge'

A second chip (also known as the South Bridge) builds an interface between the CPU and the PCI local bus. The signals coming from the PCI bridge are fed to

the backplane connectors. For your convenience most manufacturers also put a so-called 'multi-io' chip on the main board. This one contains all the peripherals that, in the original PC, were previously on Plug-in boards.

In Pentium II or III style machines the memory controller including the cache is contained inside the processor package. Just add memory and you are ready to run.

20.3 The PC's Input and Output Components

So far we have seen the parts that build a PC as we know it. All of these are located inside the box of the computer. In order to be able to connect it to the outside world we need a means to interface the computer. And that is exactly what this chapter is all about.

20.3.1 The Parallel port

This is the standard parallel port that you use to connect a printer. This port is often referred to as a Centronics interface or Printer port. The PC has the capability to handle 3 parallel ports. While this port was mainly designed to attach printers and plotters, you can use it for a lot of different tasks. Over time most people have discovered the usage of this port as a general IO channel. They use it to control attached boards and equipment. A number of professional machines use this interface to talk to the PC and the programs running on the computer. Items such as Device programmers (EPROM , Flash etc), Emulators for microprocessors to complete measuring systems are available off the shelf. The use of this port as IO channel will be handled later on .

20.3.2 The Serial port

The Serial port is the second port that is standard port available on any computer. There are two different styles. You can have either the full-fledged 25-pin connector or the shrunken 9-pin connector. The port is controlled by a UART of the 8250 / 8251 or 165x0 type. The latter has more advanced features like transmit and receive FIFO's, however from a programmer's point of view these controllers all look the same.

Since the port holds a lot of registers and is quite complicated to control low level , I'm not going to detail on that here and now. I will explain the things you need to know and how you can make an interface that works reliable all the time.

While the UART can be set to all sorts of different baud rates , parities , stop bits , modes and so on , the only one that is really important is 9600,n,8,1 mode. This is a typical communication mode that is most widely used to talk to devices of all sorts.

In the PART II of this book I have already detailed about the MsComm object that allows you to perform communication on this port. Later on in this part I will discuss the hardware side of these channels. A standard PC can drive up to 4 serial ports . There are special plug in boards that give you access to more channels. However these boards are costly and sometimes poorly supported. Anyhow , the chance that you will need more then 2 ports is small.

The physical communication can take different forms. Most used is the so-called RS232 standard. For large systems over long distance they use RS485. More about these standards comes later.

20.3.3 The USB port.

This is a new rising star on the PC interface domain. The reason for the development of this bus was the quest to diminish the vast amount of cables connected to a typical PC. Today's computers often have the following cabling, A keyboard, a mouse, a printer, a modem, a network cable and a video cable. If you have a scanner there is an additional cable required. An extra printer? : extra cable! In the end you end up with a terrible mess. USB tries to deliver the answer to this problem. By defining a universal bus that boasts fast communications in a network style you minimize cabling. You can lead one cable from the PC to the printer. From the printer to the monitor. And then from the monitor a cable to the mouse and keyboard. From the monitor you also go to the scanner. The only point is that you need small 'hubs' to connect all these cables. Fortunately these can easily be built into already existing hardware. Monitors featuring USB often have 4 or 5 USB entry connectors. So it is easy to connect keyboard, mouse and printer to the monitor and then go from monitor to the PC. USB also provides transparency to the system. The driver inside the operating system handles all low-level tasks such as assigning the addresses and configuring the devices.

20.3.4 Fire Wire Channel

Besides the USB bus there is also FireWire. While USB is a 'lightweight' bus, FireWire has a bigger capacity. The data throughput is higher and the number of nodes larger. Of course this brings additional overhead. At the time of writing the debate is going on between USB and FireWire. The first machines start appearing that have this interface and only time will tell which one will eventually win.

20.3.5 Local Area Network (LAN) and Wide Area Network (Internet)

Most machines now either have access via a LAN board of some sort (Ethernet, Thin-net, Thick-Net, ATM, FDDI, Token Ring etc ..) to the premises network. This network links computers and peripherals together. Some

machines , which are controlled by a computer , allow you to control them via these channels.

From a LAN to a WAN is only a small step. Controlling applications via LAN or internet is possible using the Winsock control. Part II of this manual gives an explanation of the possibilities of these channels , and presents some samples.

20.3.6 Field buses (CAN VAN etc)

Besides the above-mentioned communication systems , a number of dedicated automation buses have been set up. They are commonly used on factory floors and start finding their way to Lab environments. A number of instrument vendors already have equipment that can patch into these buses. The drawback of these buses , compared with the buses presented in previous points , is that you will need to buy an adapter card to plug into your pc. No computer vender can deliver you a machine with this kind of channel built in.

The advantage is that these are very rugged buses that can withstand very harsh environments. They have no problem with noise and feature a fast data throughput and quasi real-time event handling.

20.3.7 The GPIB Bus

Many consider this the 'golden-oldie' of all instrumentation buses. While this bus also requires a special adapter , this is possibly the most commonly used bus in instrumentation systems. While very old (end 60's) it is still regarded to as one of the most powerful buses around. The vast throughput (10MByte / second) and the well-documented bus (IEEE-488, IEEE488.2 and IEC625 standards) have lead to the huge success of this bus. Almost any data collecting equipment can be delivered off-factory with this interface on board. It allows you to connect up to 32 instruments directly onto one bus.

20.3.7 VXI / PXI / SCXI / Compact PCI etc ..

These are not really connection channels to the pc , but channels amongst equipment. Typically systems that support these buses have embedded

computers. This means the PC is actually built into the same basic block that contains all the equipment. This is an emerging standard that is still growing. The boards that support these buses start to appear on the market. However, this is a different approach. Normally you connect your PC to a measurement setup. In these systems, you embed the computer in the system. Your PC actually becomes a fixed part of the setup. Where you normally plug acquisition board into the computer, here you plug the computer in the acquisition system.

When you want to control a system with you have a number of possibilities. You can either interface it via a standard channel like the serial port or you can go the industrial way and use buses links like SCSI, USB, and GPIB etc. However, when building electronics you don't always have the possibility to use these buses due to the hard and software overhead.

For the quick and dirty job you only have 2 real options. Go serial (RS232) or bit-bang your stuff on a printer port. Each of these has a number of advantages and shortcomings.

If you can invest some more time you can use a universal IO board plugged into your computer. Designing such a board is not that difficult, or you could simply buy one off the shelves.

20.3.8 SCSI

SCSI (pronounced "skuzzy") is an acronym for the Small Computer System Interface. It grew out of a proprietary interface protocol, SASI, which was developed by Shugart Associates to connect computers to hard disk drives. Because of these origins, SCSI bus operations are oriented for efficient use by mass storage devices like hard disks, CD-ROMs, rewritable optical disk drives, and tape drives; in practice, nearly all computer systems that use SCSI use it to connect such devices. SCSI started to come into widespread use around 1984 and became the standard way to connect workstations such as the Sun-3 to disk and tape drives. The definition of the SCSI bus and how it should operate is defined by ANSI (American National Standards Institute). The original SCSI standard was ratified in 1986; the current version of the standard is SCSI-2, which was finally ratified in 1993. There is also a new version of the standard, SCSI-3, under active development.

Since SCSI is used primarily as a connection to disk drives, you might wonder why we're discussing it in a chapter on data communications. The answer is that SCSI is a full-fledged peripheral bus that provides communications between a

host computer and disk or tape drives in different enclosures (or within a single enclosure) by means of a SCSI-bus cable. Thus, although the protocols and the devices being interconnected are quite different, SCSI is very similar in function to GPIB and other data communications protocols.

SCSI is the standard bus for connecting disks and tape drives in workstations such as those from Sun and DEC and is the standard built-in expansion port on Apple Macintosh machines as well. It's taken longer for SCSI to catch on in the PC world, mostly due to its higher costs and software compatibility problems. However, the proliferation of CD-ROM writers, Zip drives and Tape streamers and the early availability of SCSI hard drives with capacities above 512MB have brought SCSI into fairly widespread use on PCs also.

There are actually several allowed variations of SCSI that came into being with the SCSI-2 standard; this can create considerable confusion when you're first trying to understand how SCSI systems work. First, SCSI can use either single-ended signaling (each bus signal is carried on a single wire) or, less commonly, differential signaling (each signal is transmitted as a voltage difference between two wires). Second, the SCSI-2 standard allows "Fast SCSI" to be implemented. With Fast SCSI, data transfers take place at a maximum burst rate of 5-10 million transfers per second for blocks of data. In the original SCSI-i standard, the maximum data transfer rate was 5 million transfers per second. Sometimes SCSI-2 is taken to be synonymous with Fast SCSI, but this isn't necessarily the case-Fast SCSI is an option. In any event, Fast SCSI only ensures that the burst data rate is above 5 MHz. A Fast SCSI device could have a data transfer rate of only 5.1 MHz.

In SCSI-1, only 1 byte at a time can be transferred over eight data lines, DBO-DB7. However, the SCSI-2 standard also allows "Wide SCSI" to be used, with 2 bytes transferred at a time over 16 data lines. Thus, by using both Fast and Wide SCSI, up to 20MB/sec data transfer rates can be obtained. There is also an option in Wide SCSI for the use of 32 data lines, but the auxiliary cable required for this is so awkward that virtually no one uses it.

Like GPIB, the SCSI bus transmits either command or data bytes on eight wires running in parallel (or 16 wires for data if Wide SCSI is used). It's considerably faster than GPIB, however, with data rates being anywhere from 1-20MB/sec. Also like GPIB, there can be more than one host controller on a SCSI bus, although this is very rare in practice. A typical SCSI bus system for a PC has a single host, typically a SCSI card or a built-in SCSI controller in the PC. The host is connected to one or more SCSI drives, each containing its own embedded SCSI controller.

Unlike simpler disk-connection standards such as IDE, SCSI devices are very intelligent and can carry out many operations on their own.

20.4 The internal buses

Besides the already mentioned buses above, the PC has internal buses as well. These can be used to connect hardware directly. These buses are already used to interconnect the IO ports such as printer ports, video cards etc.. to the CPU and memory system. Over time a number of buses have emerged and some have disappeared again.

20.4.1 ISA Bus

The original PC as designed by IBM had an IO bus to plug in all sorts of components. At the same time the PC was introduced, IBM released a Technical manual describing the whole schematic of the computer and BIOS listings. From this a number of third party developers started constructing their own add-on and plug in boards. Unfortunately IBM never specified the so called AT bus. To remedy this problem Intel and a number of other important players started to specify timing and loading parameters. After some years the interface bus gained the label ISA (Industry Standard Architecture) bus. While the bus was originally intended to put adapters such as video cards, printer ports, network etc in the computer it also started to be used as IO bus for other boards. A number of companies provide digital I/O, AD-DA boards, relay cards and more.

Pretty soon the bus got clogged up due to the fact that there are only a limited number of IO ranges, interrupts and DMA channels available. This and the cumbersome configuration of all the jumpers found on these early boards proved to be a big bottleneck.

Moreover, while the original PC had an 8-bit data-bus, the introduction of the 286 called for a 16-bit bus. At that time the designers decided to add more interrupts and DMA channels. But that space got mapped pretty soon too.

20.4.2 EISA Bus

An attempt was made to extend the ISA bus even further while at the same time keeping it backward compatible with the ISA standard. A consortium lead by Intel and Compaq began developing what would become known as EISA (Extended Industry Standard Architecture). The main mechanical difference was a clever design for the connector slot. To support the faster and wider data and address buses many new control signals were added. This caused such a hardware overhead ,which in turn is costly to develop , that the bus never caught on.

20.4.3 MICROCHANNEL Bus

Since the ISA bus started leading a life of it's own , and IBM lost a big market share to third party developers , they decided to regain their position by introducing a new IO concept. This bus featured very modern technologies such as bus mastering , interrupt sharing etc. Unfortunately this bus had the same problems as EISA. The hardware overhead made the overall implementation too expensive. That and the fact that it was not hardware compatible with older cards lead to the quite death of this bus.

20.4.4 VESA Bus

With the introduction of new 'Video' hungry operating systems the bandwidth to the video adapter became an important bottleneck. A consortium of Video Board manufacturers called VESA (Video Electronics Standards Association) was trying to emerge a new standard. To keep cost to a minimum they decided to simply connect directly to the CPU's local bus. The bus became widely used in the last days of the 486 processors. It managed to 'offload' the ISA bus for a number of bandwidth hungry tasks such as Video and hard-disk access.

20.4.5 PCI

In 1992 Intel began developing a new bus standard to interconnect peripheral components. The original idea was to create a standard for high-speed interconnections on a motherboard. Since recent attempts to 'upgrade' the original ISA bus had failed, it was time to create a new totally new bus architecture. The PCI bus took some of the ideas from the VESA bus and connects directly to the processors local bus. It also took some of the advantages of the Micro Channel bus, namely the bus-mastering technology and interrupt sharing. Intel started integrating this bus architecture in its motherboard chipsets and pretty soon card manufacturers started developing boards for PCI. This computer has now evolved to the point where ISA will no longer be implemented.

20.4.6 AGP port

Did evolution bring forth yet another bus? Not exactly. Modern computers boast such powerful and bandwidth hungry display cards that pretty soon PCI proved no longer efficient. The PCI standard has a strict timing scheme that allows no one to stretch the limits of the bus. The AGP (Advanced Graphics Port) is not a real bus. It only supports one device and has no purpose other than feeding data to graphics boards. You can think of it as follows 'AGP is to PCI what VESA was to ISA': a dedicated high-speed channel for Video applications.

20.4.7 PCMCIA (PC Card)

Several years ago , as notebooks became popular a new problem had to be faced. How do we get add-on boards in such a tiny little box ? Two consortia , one in Japan (JEIDA) and one in the USA (PCMCIA) started developing a new bus. In 1989 the PCMCIA consortium accepted the JEIDA developed connector standard and today these two consortia work together to promote this form factor. The official name for the bus has since then become 'PC-Card'. A special controller chip that bridges the PC-Card bus to the ISA or PCI bus handles the interface.

20.4.8 I2C Bus

This bus was designed by Philips semiconductors in the early 80's as an easy way to interconnect integrated circuits. The Inter-Integrated Circuit (IIC or I2C) bus was designed to get rid of address and data-buses. It is a sort of serial communication bus that only requires two signals plus a ground. It features multi-master operations and collision detection in hardware. While it was originally intended to find its way in consumer electronics such as TV's , Video's and audio equipment , it now is found on computer motherboards as well. This bus is used for system monitoring and handles tasks as battery control , temperature control , voltage and hardware monitoring etc . Typically a small processor (the same CPU that handles keyboard and mouse) controls this bus and provides an interface to this I2C bus. In portable computers this bus is also used for the touch-pad or stick that replaces the mouse.

Chapter 21:

Controlling Standard PC ports

When you want to control a system with you have a number of possibilities. You can either interface it via a standard channel like the serial port or you can go the industrial way and use buses links like SCSI ,USB , and GPIB etc. However , when building electronics you don't always have the possibility to use these buses due to the hard and software overhead.

For the quick and dirty job you only have 2 real options. Go serial (RS232) or bit-bang your stuff on a printer port. Each of these has a number of advantages and shortcomings.

If you can invest some more time you can use a universal IO board plugged into your computer. Designing such a board is not that difficult , or you could simply buy one off the shelves. This chapter will detail on these boards and explain their functionality and how to access them.

21.1 Finding the IO ports

Controlling PC I/O channels is very nice but how do you know what is available in your particular computer and where are they mapped. There are a few rules of thumb you can apply.

- Serial ports can be handled through the MSComm object. For these ports you don't need to know where exactly they reside in your system.

- Parallel ports can only reside on 3 possible addresses : 378 , 278 and 3BC. However most computers only have one and that is typically 378.
- IO boards can only be mapped into specific regions of the IO address space of the PC.

You might say : This is all nice but how can I know for sure ? Well there is really only one answer to this question : The computer knows ! Only question is : how do I get the computer to telling me ?

21.1.1 The BIOS system area

During startup of the computer (even before the operating system boots) the BIOS program scans all hardware. The BIOS is a library with routines to perform I/O on your computer. It has simple routines to write text to the display , initialize the disk array etc. In other words : it makes your computer work. This BIOS performs a number of scans to detect the hardware present in the computer. It stores this information not only for the user but also for itself in the so-called BIOS data area. This is the portion of main memory on page 0 at offset 400.

This block of data exists even under windows and windows NT. The reason is simple :you can't move it !. you can read and write it but you can't move it. Most programs need it , your operating system needs it and even the computer hardware itself needs it. In other words: You are always able to extract data from it.

When the BIOS has completed its task and handed over control to the bootstrap loader it left some data in memory. This data is sometimes called the System area or the 'System Metrics' Data. It is a block of 256 bytes that gives information about the machine.

A typical page dump looks like this

```
-d 0040:0000 ff
0040:0000  F8 03 F8 02 E8 03 E8 02-78 03 78 02 00 00 00 00
.....x.x.....
0040:0010  23 C8 00 80 02 00 30 A0-00 00 36 00 36 00 30 52
#.....0...6.6.0R
```

```

0040:0020  30 52 20 39 32 50 35 4C-35 4C 08 0E 08 0E 08 0E  0R
92P5L5L.....
0040:0030  66 21 66 21 0D 1C 30 52-3A 34 30 52 30 52 01 00
f!f!..0R:40R0R..
0040:0040  04 00 20 00 00 00 00 00-00 03 50 00 40 20 00 00  ..
.....P.@ ..
0040:0050  00 17 00 00 00 00 00 00-00 00 00 00 00 00 00 00
.....
0040:0060  07 04 00 D4 03 29 30 76-07 87 1C FF 84 6F 13 00
.....)0v.....0..
0040:0070  00 00 00 00 00 02 08 00-14 14 14 3C 01 01 01 01
.....<.....
0040:0080  1E 00 3E 00 31 08 00 60-09 11 0B 80 58 00 00 07
...>.1...`....X...
0040:0090  87 07 00 00 00 00 10 12-A0 00 40 00 88 FD FF FF
.....@.....
0040:00A0  00 00 00 00 00 00 00 00-2E 39 00 C0 00 00 00 00
.....9.....
0040:00B0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
.....
0040:00C0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
.....
0040:00D0  00 00 00 00 00 00 00 00-00 00 00 01 00 00 00 30
.....0
0040:00E0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
.....
0040:00F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
.....
-

```

Most of this stuff is irrelevant for what we do. However some very interesting things can be found. Let's take a look at the very first line

```

0040:0000  F8 03 F8 02 E8 03 E8 02-78 03 78 02 00 00 00 00
.....x.x.....

```

This is the line we are interested in. It shows the available ports and the addresses they are located on. The first four words specify the serial ports and the next 3 words specify the parallel ports. The last word is undetermined. Some machines use it for a fourth printer port but in general it is not in use.

Note :

A port address is specified by 2 bytes in little endian coding. This means that, in order to obtain the correct address, you have to swap the high and low byte. Example F8 03 becomes 0x03f8

From left to right we find this 0x03f8 0x02f8 0x03e8 0x 02e8 for the serial ports and 0x0378 and 0x0278 for the parallel ports. This means on this particular machine there are 4 serial ports and 2 parallel ports. If a port locator contains 0x0000 it means that this port does not exist. The BIOS functions (and

most DOS programs and) use these tables to look up the address for their transactions. This means that , if you want to put a LPT port at a certain address , you can plug in the board and specify the address here yourself. Any printer routine will then be redirected to this new address.

21.1.2 Using *DEBUG* to snoop around

Check the BIOS data area of your computer. Easier said than done ...you think. There is a neat little tool on any PC that allows you to do exactly this job. This tool is called *DEBUG* and dates back from DOS 1.2. While this is a very powerful tool it is also a very dangerous tool. If you have no clue what a certain *DEBUG* command does then you shouldn't try it! In the best case you run the risk of crashing the computer , and in the worst case you screw up your entire disk while doing it.

Hopefully I haven't scared you too much. What I am going to describe now is harmless for your computer. The above doomsday scenario was just intended to warn you not to experiment with commands that are not explained here.

Debug can mostly be found in the DOS or WINDOWS\COMMAND directory. If you are running on Windows NT this might not be installed.

Let's start it up : Open a Dos box and type *Debug*.

```
C:\>debug
-
```

You will get the debug prompt "-". You are now in total control of the machine. Feels good doesn't it ? To display the commands at your disposal simply type ? and press return

```
C:\>debug
-?
assemble      A [address]
compare        C range address
dump           D [range]
enter          E address [list]
fill           F range list
go             G [=address] [addresses]
hex            H value1 value2
```

```

input      I port
load       L [address] [drive] [firstsector] [number]
move       M range address
name       N [pathname] [arglist]
output     O port byte
proceed    P [=address] [number]
quit       Q
register    R [register]
search     S range list
trace      T [=address] [value]
unassemble U [range]
write      W [address] [drive] [firstsector] [number]
allocate expanded memory      XA [#pages]
deallocate expanded memory    XD [handle]
map expanded memory pages     XM [Lpage] [Ppage]
[handle]
display expanded memory status XS
-

```

As you can see , a wealth of instructions allows you to manipulate virtually everything on your computer.

21.1.3 The Dump command

Probably one of the most important command is the D(ump) command. It allows you to physically examine the contents of the memory

```

-d 0040:0
0040:0000  F8 03 F8 02 E8 03 E8 02-78 03 78 02 00 00 00 00
.....x.x.....
0040:0010  23 C8 00 80 02 00 30 A0-00 00 1E 00 1E 00 75 16
#. ....0.....u.
0040:0020  67 22 0D 1C 3F 32 0D 1C-0D 1C 0D 1C 64 20 20 39
g"...?2.....d 9
0040:0030  30 52 30 52 34 4B 30 52-3A 34 30 52 0D 1C 01 00
0R0R4K0R:40R....
0040:0040  D6 00 20 00 00 00 00 00-00 03 50 00 40 20 00 00  ..
.....P.@ ..
0040:0050  00 31 00 00 00 00 00 00-00 00 00 00 00 00 00 00
..1.....
0040:0060  07 04 00 D4 03 29 30 76-07 87 1C 04 E3 8C 10 00
.....)0v.....
0040:0070  00 00 00 00 00 02 08 00-14 14 14 3C 01 01 01 01
.....<....
-

```


If you simply type 'd 0040:0' and hit return , this command will show you the part of memory mentioned in the previous topic. Once you found what you need its time to leave DEBUG . This is done by simply hitting Q and pressing Enter.

21.2 Hardware Access

What could possibly be told about this ? Well more than you would expect. All the ports from the PC are accessible via a set of instructions residing in the Microprocessor. The PC's processor has a separate IO space that is accessed using IN and OUT instructions. Of course we are not going to write assembler but Visual Basic. So you would expect that there are equivalent Visual Basic instructions to access these IO ports. Well hardware access is one of the things that Microsoft deliberately left out of Visual Basic. Windows is not the platform to tinker with hardware . Furthermore , playing with it requires a great deal of knowledge about the system , and is only useful if you want to write system drivers. Since drivers can (for the moment) only be made in 2 languages (Assembler and C), these operations have been left out.

But sometimes someone might just need to do this kind of operations, so a clever guy ☺ created a DLL in assembler that allows just this kind of stuff. Now there is still a difference between programming such things in C and doing it in assembler. When using the 'c' language you have to use the built in routines of the language. These are written to be fully Windows compliant. This means that they behave very nicely and ask permission to the system before they access a port. Sometimes you will miss data , or simply be denied access. The DLL mentioned here (Win95io for 32 bit or WinIO for 16 bit) is written in pure assembler. And I for one am not polite with the system. Under normal circumstances you cannot do the thing the DLL executes. Windows would respond immediately with the known blue screen of death : ' this program has executed an illegal command and will be terminated ' and terminate the program. But if you temporarily kick windows out of the scene then you can do whatever you want. There is nobody looking over your shoulder to see what you are doing. And that is exactly what Win95io is doing. It temporarily disables ALL interrupt processing from Windows. Since Windows is an entirely interrupt driven system for its internal system management this means that the entire operating system comes to a halt. Once Windows has stopped the DLL performs the IO required and then reinstates the interrupt controller. At that point the operating system revives and continues where it left off. Since it was suspended while the IO took place it knows nothing about the actual operation and as such it does not generate this exception.

Windows NT is a different matter. There this trick does not work since the instructions that control the interrupt processing of the CPU run in a protected ring. If you try to execute them the processor himself will generate the exception. You only get access if you are a device driver to this kind of instructions. So for the moment Win95io does NOT work on Windows NT. However it works on windows 95/98.

Appendix A contains the Users Manual for this Win95IO DLL. You can use this DLL free as long as you don't sell software that makes use of it.

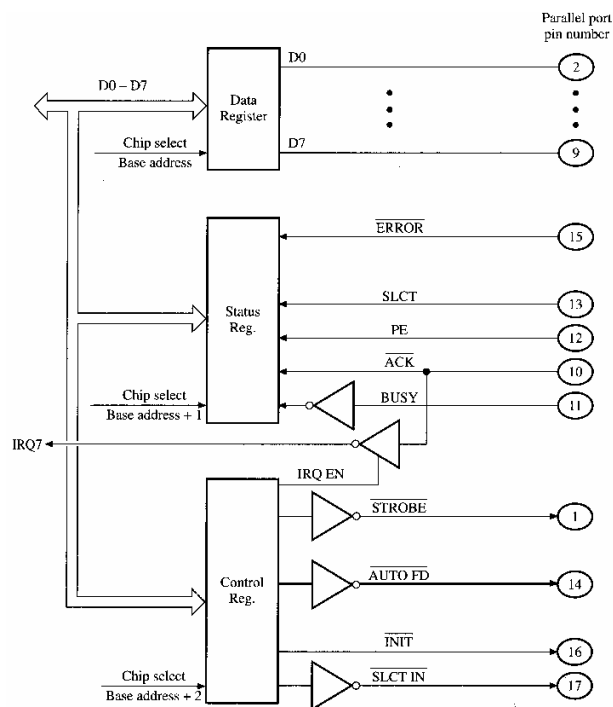
Chapter 22

The Printer port In Detail

This port (sometimes also called the Centronics port) dates from long before there was a PC. As the name indicates, it is a communication port mainly intended to control an attached printer. The Centronics designation comes from a company with the same name that has designed this kind of interface. Of course we are not interested in its capabilities to drive a printer but more as a universal IO channel existing on any PC.

22.1 Functional diagram

The schematic below shows the implementation of a standard printer port as can be found in any PC.



22.2 Register level description

The table below shows an address map of any printer port. Of course there are today also ports known as EPP, ECP or bi-directional. Since these modes neither are nor uniform in use nor available on every machine they will be omitted here.

Base	D7	D6	D5	D4	D3	D2	D1	D0	Output Register
	9	8	7	6	5	4	3	2	
Base + 1	BS	AQ	OP	SL	ER	-	-	-	Status Register
	11	10	12	13	15	-	-	-	
Base + 2	-	-	-	IE	SI	IP	AF	ST	Control Register
	-	-	-	-	17	16	14	1	

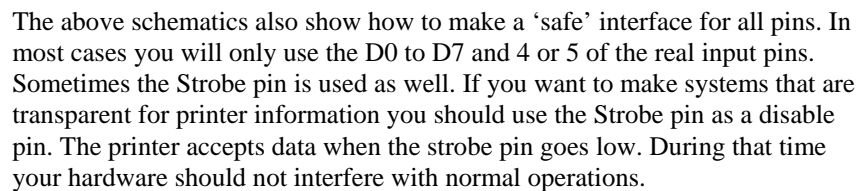
A typical PC has at least one port (mostly on 0x378 , sometimes on 0x3bc depending on where it resides). As explained in the previous chapter you can have up to 3 printer-ports in a PC. The numbers shown above correspond to the pin number of the 25-pole D-connector used for this kind of port.

The following table explains the meaning of the abbreviations used.

BS	Busy	Input
/A/Q	Acknowledge	Input
OP	Out of Paper	Input
SL	Select	Input
ER	Error	Input
IE	IRQ enable	Input
SI	Select input	Bidir
IP	Initialize Printer	Bidir
AF	Auto Feed	Bidir
ST	Strobe	Bidir

The Acknowledge pin is inverted inside the printer port hardware !

As you can see from the above , the printer port opens up a lot of possibilities to control boards. However , since a printer port is part of the computer , care should be taken not to destroy it accidentally . The following schematic shows a typical interface that will protect the port under most circumstances. It also acts as a buffer to clean up the signals generated by the port



22.3 Basic operations

Basically a printer port is used to control some external device such as an EPROM programmer, IO board or other piece of electronics. You will find that most often the port is used to create some pseudo serial interface. This 'emulation' is called bit banging and explained in the next topic.

To control the pins you need to make use of the INP and OUT commands of WIN95io. For people coming from a DOS basic : these operate in exactly the same way as under DOS basic. In the table below the mapping of the addresses is shown once again.

Base	D7	D6	D5	D4	D3	D2	D1	D0	Output Register
	9	8	7	6	5	4	3	2	
Base + 1	BS	AQ	OP	SL	ER	-	-	-	Status Register
	11	10	12	13	15	-	-	-	
Base + 2	-	-	-	IE	SI	IP	AF	ST	Control Register
	-	-	-	-	17	16	14	1	

Suppose you want to control the lines D0 to D7. The table shows that these are mapped onto the BASE address (which can be either 378, 278 or 3BC). So to send the byte '5A' simple do the following

```
Const LPTport = &h378
Out LPTport , &h5a
```

The status register shows the status of the 5 input pins. So to extract data you simply execute a INP statement

```
Const LPTport = &h3BC
X = INP ( LPTport +1)
```

Note in the above examples that I define a constant to access the printer port. If you do this as a global in a module it becomes easier to modify your program to run on different ports. You could also store the address for the printer port in a

variable. That way you can dynamically change the address for the used port. You could then allow the user to redirect the IO to a different port (via an option or setup menu for instance)

Now that you have retrieved this information you can use Boolean operations such as AND OR and NOT to isolate the bit or bits you want.

22.4 Bit-Banging interfaces

As already explained , in most case the printer port will be used to emulate some kind of serial device interface. Typical uses are SPI , I2C , Micro Wire , and JTAG etc. They can even be proprietary interface such as test patterns for an integrated circuit.

While there are several approaches to bit banging only few are really interesting and transparent .

22.4.1 Simple line control

When you simply need the printer port to act as a simple IO device the easiest approach is to make subroutines to control each line.

Suppose you have 8 relays that must be controlled independently. The problem is that you can only write to the output. You can't read from it (sometimes you can but this depends on the printer port installed in the computer. Don't count on this !). So whatever information was present is overwritten and lost. The solution is to use an internal variable that holds the data to be sent out. The contents of this variable are modified at will and then written to the output port.

```
Dim portdata as integer
Const LPTport = &h378

Sub RelayOn(relay)
    Portdata = portdata or (2 ^ relay)
    Out LPTport,portdata
End sub

Sub RelayOff(relay)
    Portdata = portdata and (255-(2 ^ relay))
```

```

        Out LPTport,portdata
    End Sub

```

The above routines simply calculate the binary value that corresponds with every output pin. And depending on the routine set it using an OR function or reset it using an AND operation with the inverted pattern. Let's take a closer look:

Operation		Output status	Comment
Portdata = 82		01010010	Relays 7,5 and 2 are currently on
RelayOn (5)	$2^5 = 32$	00100000	we want to set relay 6 on as well
OR on the above numbers		01110010	The relays that were on are still on
RelayOff(5)		00100000	
Inversion of this mask ($255-(2^r)$)		11011111	
ANDing the mask with the data		01010010	The relay is back off

This piece of source shows you simple IO operations. But there is more.

22.4.2 Serial protocol emulation

An other example might use different pins for different purposes. Let's look at a simple shift register. This typically consists of a data input and a clock. There might also be a reset pin.

Suppose that reset is attached to D0 , DATA is attached to pin D2 and CLOCK is attached to pin D5 of the printer port . the value of D0 = 1 , D2 = 4 and D5 = 32.

Note

These values are simple binary conversions . you get the value of a pin from the following formula: $\text{value} = 2^{\text{number of the bit}}$.

```

Dim LPTport,LPTdata
LPTport = &h3bc
LPTdata=0
Const RESETPIN = 1 ` these numbers are decimal
representations
Const DATApin = 4 ` of the value of each pin . they
are not the
Const CLOCKpin =32 ` pin numbers !!

```

The above is the initialization code. Now there are two ways to implement the actual code.

Example 1 : Monolithic Code

```

Sub transmit (pattern$)
  Out lptport,reset
  Out lptport,0
  For x = 1 to len pattern$
    If mid$(pattern$,x,1) = 1 then
      Out lptport,datapin
      Out lptport,datapin+clockpin
      Out lptport,datapin
      Out lptport,0
    Else
      Outlptport,0
      Out lptport,clockpin
      Out lptport,0
    End if
  Next x
End Sub

```

The above program is actually very simple for the person writing this , but to debug this it gets a bit more complex. And for someone who is casting his eyes for the first time on this code it might look crazy. Now in this example it is still fairly easy but in most cases the program is more complicated. You might need to control some additional pins as well. The additions might look like this

```

Out lptport,datapin+clockpin+chipselect+notreset

```

If you need to modify the protocol you will most likely have to rewrite all of this code. The solution is to partition your code. The above routine is called a Monolithic piece of code. This is completely against all the concepts in Object oriented programming and should be avoided in all cases.

Example 2: Partitioned code

```
Sub transmit (pattern$)
    ResetHi
    ResetLo
    For x = 1 to len (pattern$)
        If mid$(pattern$,x,1) = 1 then
            DataPinHI
        Else
            DataPinLo
        End if
    Next x
End Sub

Sub ResetHi()
    LPTdata = lptdata or resetpin
    Out LPTport,LPTdata
End Sub

Sub ResetLo()
    LPTdata = LPTdata and (255-ResetPin)
    Out LPTport,LPTdata
End Sub

Sub ClockHi()
    LPTdata = LPTdata or Clockpin
    Out LPTport,LPTdata
End Sub

Sub ClockLo()
    LPTdata = LPTdata AND (255-clockpin)
    Out LPTport,LPTdata
End sub

Sub DataHi()
    LPTdata = LPTdata OR datapin
    Out LPTport,LPTdata
End sub

Sub DataLo()
    LPTdata = LPTdata AND (255-datapin)
    Out LPTport,LPTdata
End sub
```

The above code is a lot longer to write but much easier to debug. You can nearly find the timing diagram implemented as code.

22.5 Printer port Control Using ClassWork

What is ClassWork you might ask ? Well this is going to be explained a bit further on. For now it is sufficient to know that this is a piece of software that exposes certain hardware (Printer-ports , GPIB instruments , USB etc) as objects. That means you can treat these devices just as if they were command buttons or textboxes.

To use the printer port class simply load the Printerport.cls file into your project.

```
Dim lptport as new Printerport.  
LPTport.address = &h378  
LPTport.D0 = True  
LPTport.D5 = False  
LPTport.dta = 123  
If LPTport.BS = true then msgbox "Pin 11 is logic  
High"
```

This Class allows you to control all pins of the printer port independently. Assigning a new value to the DTA property can also change the output register.

Method or property	Function	Implementation	Type
D0 .. D7	Output pins D0 to D7 individually controllable	Sub	Boolean
Dta	The output pins D0 to D7 as a byte	Sub	Integer
BS,AQ,OP,SL,ER	The input pins individually	Function	Boolean
Nibble	the high 4 bits of status register scaled down	Function	integer

Note

the inversion in the printer port for the AQ pin is taken into account. If the result of function is TRUE then the corresponding pin is at logic High !.

22.6 Special printer port modes

22.6.1 Bi-directional Parallel Ports

The PC's parallel port(s) would be much more useful if it could be used as a bi-directional port, that is, if it could transfer data in both directions. Unfortunately, the original IBM parallel port design was only meant to be unidirectional, just sending data out from the computer. This design was particularly unfortunate because it would have taken no additional hardware to make the parallel port bi-directional. In IBM's parallel port, the logic levels on the data lines can be read in (through a 74244 tri-state buffer that implements the input function of the data register); however, because the outputs of the 74374 used as the output of the data register are always turned on, no data from an external source can be put on the data lines. All you can do is read back the last byte written to the data

register. If the 74L5374's tri-state output enable pin had been connected to the extra bit on the interface's control register (a 6-bit 74LS174 latch), the data port could have been programmed for external input as well as output. Nonetheless, the standard parallel port design can be used for data input by making use of four of the five status register input lines to input data from an external device a nibble at a time. The fifth input line can be used to implement a handshake signal. This technique is used by a number of programs that allow you to transfer data between your desktop computer and your laptop machine by hooking their parallel ports together. Both the cabling and the handshake signals used are nonstandard and vary from one program to another. A good example of this type of program is the interlink / intersvr program included in MS-DOS or the Direct-Cable connection used by Windows 95/98.

When it introduced the PS/2, IBM made the parallel port bi-directional by allowing the data register outputs to be turned off using bit 5 of the control register (allowing data to be input from the parallel port if bit 5 = 1). To further complicate the situation, IBM then introduced a more sophisticated parallel port (called a Type 3 parallel port) on later PS/2s that allowed high speed bi-directional data transfers using DMA. Intel also introduced a laptop computer chip set containing what it called a Fast Mode parallel port that allowed higher speed data transfers. However, only a fraction of PC manufacturers adopted any of these improvements. At this writing, there are still many PCs being sold that use the original PC unidirectional parallel-port design.

22.6.2 The IEEE 1284 Standard

In an effort to obtain some standardization in the parallel-port variations, the IEEE has created a standard (IEEE P1284) to define five modes of parallel-port operation that allow parallel ports on PCs and peripheral devices (printers, scanners, modems, and so on) having differing capabilities to inter-operate with each other. These are the possible modes of operation for a 1284-compliant parallel port:

1. **Compatibility Mode.** This is the mode of operation used by the original PC parallel-port interface. The interface operates according to the Centronics printer interface specification with data being sent only from the PC to the external device. The signal definitions are described earlier in this section.
 2. **Nibble Mode.** This mode of operation was also described earlier. It uses the status line inputs of the original PC parallel interface to implement data
-

transfers from the external device to the PC. Data bytes are transmitted over four of the status lines as two sequential 4-bit nibbles.

3. Byte Mode. This is the mode introduced on the IBM PS/2, and was also described earlier. The improved parallel-port circuitry required to implement this mode allows data bytes to be transmitted over the data lines from an external device to the PC when the direction bit (bit 5) is set in the PC's control register.
4. ECP Mode. The Extended Capabilities Port Mode allows a PC and an external device to freely communicate back and forth with each other. Enhanced parallel port circuitry allows the original parallel-port control lines and handshake protocol to be redefined so that they implement an asynchronous 8-bit bi-directional data channel using the data lines.
5. EPP Mode. The Enhanced Parallel Port Mode requires fairly complex parallel port circuitry and uses the parallel port data lines as an 8-bit bi-directional bus carrying both data and addresses.

The standard is defined in such a way that PCs or peripherals that utilize the original PC parallel port interface will still work (although they can only use Compatibility Mode and Nibble Mode). This is done by requiring that a 1284-compliant parallel interface be in Compatibility Mode when power is turned on. The software controlling the host interface (the PC) must then successfully negotiate with the peripheral device to operate in any other mode.

The initial negotiation is done as follows: The host sets SLCT IN high and AUTO FD low. If the peripheral is 1284-compliant, it must respond by setting ERROR, SLCT, and PE high and ACK low. A simple Centronics parallel port device will never respond this way, so this is a unique signature. When the host sees this response, it requests a new operating mode by sending a code to the peripheral on the data lines. For example, sending a code value of 01 (by placing the value on the data lines and pulsing the STROBE line) requests Byte Mode. If it supports the requested mode, the peripheral responds by setting SLCT high and driving PE low.

If the mode isn't supported, both SLCT and PE go low. The one exception to this is Nibble Mode, for which the peripheral should respond with SLCT low. This negative response for nibble mode allows very simple "dumb" peripheral devices to support the 1284 standard at the lowest level of bi-directional communications without having to examine what's on the data lines during the mode negotiations.

When operating in Nibble Mode, the various signal lines take on new meanings as indicated in the table below. Data bytes can now be transferred from the peripheral to the host in two data transfers using PtrBusy, AckDataReq, Xflag, and DataAvail for bits 3-0 and then bits 7-4. A high-to-low transition on the PtrClk line serves as the strobe signal to indicate that a new nibble value is on the lines, and a low on the HostBusy line is the handshake signal coming from the host that indicates when the next nibble can be sent. After each pair of nibbles is sent, the state of the DataAvail line at the time when PtrClk goes back from low to high tells the host whether additional data bytes are available to be sent (a low on DataAvail means more data is available).

TABLE :Signal redefinitions for the various IEEE 1284 parallel port modes. Active low signals are indicated by a minus sign preceding the signal name.

Pin	Driven by	Centronics	Nibble	Byte	ECP	EFF
2-9	Host or peripheral	DO-D7	DO-D7	DO-D7	DO-D7	ADO-AD7
1	Host	-Strobe	HostClk	HostClk	HostClk	-Write
14	Host	-Auto FD	HostBusy	HostBusy	HostAck	-DStrb
16	Host	-Init	-Init	-Init	- Reverse Request	-Init
17	Host	-Select In	1284 Active	1284 Active	1284 Active	-AStrb
15	Peripheral	-Error	- DataAvail	-DataAvail	- PeriphRequest	User defined
13	Peripheral	Select	Xflag	Xflag	Xflag	User defined
12	Peripheral	Paper End	AckDataR	AckDataRe	- AckReve	User

	ral		eq	q	rse	defined
10	Periphe ral	-Ack	PtrClk	PtrClk	PeriphCl k	Intr
11	Periphe ral	Busy	PtrBusy	PtrBusy	PeriphAc k	-Wait

To do bi-directional communications using Nibble Mode, the host sends data to the peripheral in Compatibility Mode and must negotiate with the peripheral to switch into Nibble Mode each time it wants to receive one or more bytes of data from the peripheral. The host can tell when the peripheral has no more data to send from the DataAvail line as just described, and can then return to Compatibility Mode (indicated by puffing 1284Active low) to send more data to the peripheral or wait for more data from the peripheral by going into an idle phase of Nibble Mode (by setting HostBusy low). Thus, by switching back and forth between Nibble Mode and Compatibility Mode, two-way communications can be maintained between the PC and an external device.

In Byte Mode, data is sent from the peripheral to the host in much the same way as for Nibble Mode, except that a byte at a time is transferred on the data lines DO-D7. The handshake signals are used in the same way as for Nibble Mode, except that the host briefly pulses the HostClk line low after each byte has been sent by the peripheral to indicate that the byte was received. As was the case for Nibble Mode, the host must switch between Compatibility Mode and Byte Mode to implement two-way communications.

A 1284-compliant parallel interface that supports Byte Mode may also support DMA transfer operations as specified for IBM PS/2 Type 3 parallel ports. Using DMA transfers, entire blocks of data can be read into or written out of the parallel interface by the DMA controller, leaving the CPU free to do other tasks. Two more registers, an interface control register and an interface status register, are typically used to control DMA operations. Bits in these registers are used to enable DMA transfers, to start DMA transfer of a block of data, and to detect when a data block I/O operation has completed. Hardware within the parallel interface takes care of generating and detecting the STROBE and ACK handshake signals needed to synchronize the data flow with the external device. In order to do DMA transfers, of course, the parallel port driver software must also program the DMA controller, as described in Section 7-8. Data transfer rates as high as 2MB/sec can be reached using DMA transfers. To be completely compatible with the Type 3 port, a parallel interface also needs to

support expanded interrupt capabilities, which allow interrupts to be generated when the state of any selected status register signal changes.

Using Nibble or Byte Mode to implement bi-directional communications works reasonably well if the data is mostly transmitted in large blocks or if the data flow is largely in just one direction. However, because it takes an appreciable time to do the mode negotiation for each change in data flow direction, this approach gives rather poor performance for general purpose communications in which many short messages are exchanged between the host and the peripheral.

22.6.3 Extended Capabilities Port

A parallel port that supports Extended Capabilities Port (ECP) Mode offers higher performance and better support for general-purpose bi-directional communications. In ECP Mode, unlike Nibble and Byte modes, the interface doesn't have to change modes when the direction of data flows between the host and the peripheral changes. After negotiating with ECP mode, the interface is set up for forward data transfers (that is, the host sends data to the peripheral). Forward data transfers are coordinated using an interlocked handshake. The host pulls HostClk (see Table above) low to indicate new data is available, and the peripheral sets PeriphAck high to acknowledge that it sees the new data. Upon seeing the acknowledgment, the host sets HostClk back to high, and the peripheral completes the transfer sequence by pulling PeriphAck back low when it's ready to accept another byte. This is virtually the same handshake used on the SCSI bus.

Any time the external device wants to perform reverse data transfers (that is, send data from the peripheral to the host), it asserts PeriphRequest, and the host will then enable reverse transfers by pulling ReverseRequest low when it is ready to accept them. The pair of interlocked handshake signals PeriphClk and HostAck are used to synchronize reverse data transfers. When it wants to switch the data flow direction back to being forward, the host sets ReverseRequest back to high, and the peripheral acknowledges the request by setting AckReverse high.

Parallel interfaces operating in ECP Mode also support two other advanced features. First, the host and the peripheral can send each other commands as well as data. For transfers in the forward direction, a byte sent to the peripheral is interpreted as a command if HostAck is low during the transfer and as data if HostAck is high. Thus, HostAck serves as a "ninth bit" that allow commands

and data to be distinguished. Similarly, for transfers in the reverse direction, `PeriphAck` serves as the ninth bit, which allows the host to distinguish between commands and data sent to it.

The second advanced feature is that run-length encoding of data is supported. If a command is sent with its most significant bit set, the low order 7 bits in the command are interpreted as a number, *N*, that is the run-length count for the next data byte sent. Thus, when the next data byte is received, the receiving port circuitry will behave as if it had received *N* copies of that byte. This data compression technique is very effective when the transmitted data is a raster image such as that produced by a page scanner.

The most powerful and flexible 1284 parallel-port mode is Enhanced Parallel Port (EPP) Mode. In this mode, the parallel port signal lines are completely redefined, as shown in the table above, and controlled by the host's circuitry in a manner similar to the lines of a computer system's bus. Thus, it's useful to think of EPP Mode as defining a bus over which information is transferred in bus cycles. Four types of bus cycles are defined for EPP Mode: address-write cycles, data-write cycles, address-read cycles, and data-read cycles. The host is always the bus master, and controls all operations. All devices attached to the parallel port are treated in EPP Mode as consisting of one or more registers, each of which has a register address. To send one or more bytes of data to a particular device register, the host performs an address-write cycle to select a particular register, and then performs data-write cycles to transfer the data. This way of doing things differs somewhat from a typical computer bus where the address for the data is sent during every bus cycle. In EPP Mode, it's assumed that you will normally be transferring many bytes of data back and forth to the same device register before switching to another address and that the added complexity of requiring an address to be sent along with every data byte therefore isn't needed.

The control signals used to perform the bus cycles are fairly simple. As an example, consider an address write cycle. To begin an address write cycle, the host places an 8-bit address on `_ADO-AD7` and pulls `Write` (indicating that the host is doing a write operation) and `AStb` (indicating that the information on `ADO - AD7` is an address) low. The peripheral device corresponding to the address responds by setting `Wait` high to indicate that it recognizes it's being addressed and is ready to receive the address byte. Upon seeing `Wait` go high, the host de-asserts `AStb`. This action signals the peripheral to read and store the byte on `ADO-AD7` to use as the register address for following data cycles. The peripheral then pulls `Wait` low to indicate that it's ready for a new bus cycle, and the host ends the current bus cycle by removing the signals from `ADO-AD7` and setting `Write` back high.

A data read bus cycle proceeds in much the same manner. The Dstb and Wait lines are the handshake signals that coordinate the data transfer, and the state of the Write line determines whether the bus cycle is a read cycle or a write cycle.

22.6.4 Enhanced Parallel Port

In EPP Mode, the signal lines PE, Error, and SLCT used in Compatibility Mode are left undefined and may be used as desired as peripheral status bits. EPP Mode also allows a device to send an interrupt signal to the host by pulsing the Intr line low. Finally, a short active low pulse on the Init line is the signal to all attached peripheral devices to terminate EPP Mode and return to Compatibility Mode. EPP Mode's bus-oriented operations provide some very significant benefits. First, once a device register address has been selected, the EPP Mode port circuitry allows subsequent data bytes to be transferred using a single I/O instruction per byte. In particular, very high burst rates can be achieved using string I/O instructions such as rep insb.

Typical EPP performance gives 1.5-2 MB/sec transfer rates—5 to 10 times greater than you can get from a simple Centronics parallel port. Another advantage is that more than one parallel peripheral can be attached to a single EPP port, with the cable being daisy chained from one device to another. EPP Mode operation is particularly advantageous for laptop computers, which have little space for multiple connectors on their rear panel, but often need to transfer large volumes of data to or from external devices.

The downside of an EPP-compatible port is that the circuitry is far more complex than for a simple parallel port and the low-level driver software is correspondingly more complex also. Nonetheless, with the high level of circuit integration currently available to chip manufacturers and the high volume market for PC systems, the benefits of EPP far exceed the costs, and it's likely that EPP will become a very well supported standard.

Chapter 23

The Serial Port In Detail

The Serial port is the second port which is standard available on any computer.

23.1 System description

Serial ports come in two flavors. You can have either the full-fledged 25-pin connector or the shrunken 9-pin connector. The port is controlled by a UART of the 8250 / 8251 or 165x0 type. The latter has more advanced features like transmit and receive FIFO's. However from a programmer's point of view these controllers all look the same(except for the FIFO's).

Since the port holds a lot of registers and is quite complicate to control low level , I'm not going to detail on that here and now. I will explain the things you need to know and how you can make an interface that works reliable all the time.

While the UART can be set to all sorts of different baud-rates , parities, stop bits , modes and so on , the only one that is really important is 9600,n,8,1 mode. This is a typical communication mode that is most widely used to talk to devices of all sorts.

Now what exactly does it mean. Well simple :

9600	Baud rates	measured in Bits per second
N	Parity	No parity. Alternative settings could be O(dd) or E(ven)
8	Databits	The number of bits transmitted at a time . Can be set anywhere between 5 and 11
1	Stopbits	Number of bits added to close the packet can be set to 1 2 or ½

You can set a port to these parameters by using the mode command from DOS.. Check out the section about the Mode command to learn more.

23.2 Port interface

A minimal serial interface needs only 3 wires . TX , RX and Ground. The UART uses a synchronization mechanism to lock itself onto an incoming data stream. The locking scheme is based on the transmission of a 'Start' bit. The UART synchronizes it's internal shift register to this bit and then samples in 6 7 8 or 9 bits depending on the settings.

There is a potential risk for data loss of the processor behind the UART does not 'offload' the data fast enough. To prevent this from happening the RS232 also has a set of handshake lines. These lines can be used to control the interaction between devices. However , in most applications where you connect a device to a Serial port these lines are not used. Some programs check these lines and simply do nothing if they don't find them to be operative. Fortunately you can trick the UART in believing that someone is there.

23.3 Flow Control

Computers often can send serial data faster than connected devices can receive and process them. If this happens, data is lost. This is troublesome at least, and fatal at worst. If possible, we should attempt to assure that this data loss doesn't happen. That's where flow control comes into play.

If a serial device (perhaps a computer or some other device in the communications link, like a modem) detects that data is going to overflow its receive buffer, it can request that the sending system stop transmitting data for a while. This gives the receiving device time to process data already received. When the receiving device has processed some or all of the 1. data previously

received, it can signal the sending system to continue to send data. This signaling system is called flow control.

Flow control comes in two flavors. These are most often called hardware and software flow control. Sometimes hardware flow control is called out-of-band flow control and software flow control is called in-band flow control, for reasons that will become obvious.

23.3.1 Hardware Flow Control

A complete discussion of hardware flow control requires the introduction of two additional terms. These terms are DTE (Data Terminal Equipment) and DCE (Data Communications Equipment). These are old-fashioned terms that almost serve to confuse as much as to elucidate. However, they are what we have to work with - so work, we will.

DTE implies a terminal (or computer), and DCE implies a modem. A DTE often connects to another DTE using an intervening DCE. Serial communications links are not bi-directional; you can only send or receive a signal on a single wire. DTE and DCE indicate which wires are used to send and receive signals. Diagrams of the various connections for DTE and DCE devices are shown in the sections on Null Modems, Cables and Adapters. Your PC and all computers, almost always are treated as a DTE, and the following discussion will assume that is the case.

Hardware flow control uses a separate pair of wires on the serial port to perform the signaling between the connected devices. Most often the two signals are called RTS (Request To Send) and CTS (Clear To Send). Occasionally DSR (Data Set Ready) and DTR (Data Terminal Ready) are used instead of CTS and RTS, respectively. In rare cases there is a mix of these two pairs of signals.

When a receiving DTE needs to signal the sending device that data flow should stop, it lowers the RTS line. A DTE raises the RTS line when it is able to receive data. Likewise, a DTE monitors the CTS line. If CTS is lowered, the connected device is signaling that it cannot receive much more data, so the DTE is obligated to stop sending data. When CTS is raised, the DTE can resume sending data.

On the other hand, a DCE reverses the meaning of these control signals. If a DCE needs to halt the flow of data, it lowers CTS. It then raises CTS to permit

data to flow. Likewise, a DCE monitors RTS. If RTS is lowered, no data should be sent by the DCE, while RTS high indicates that data may be sent.

23.3.2 Software Flow Control

One of the features of serial data transmission is that it requires only one wire to send data in each direction (ignoring signal ground). However, hardware flow control adds an additional path for each control signal. Perhaps there is another way. There is such a way, of course. That's software flow control. When a device needs to control the flow of data, it might be possible to define a special data character that it can send to halt or restore the data flow. There have been a couple of different pairs of characters defined to do this. The most common form is called XON/XOFF. An XOFF character (also called DC3 or Device Code 3) may be sent to signal that no more data should be sent, and XON (also called DC 1 or Device Code 1) is sent to signal that data may flow again.

Software flow control is also known as in-band flow control, because this control acts just like other serial data; it is sent on the same signal wires as other data. Software on each end of the path must respond appropriately to these special characters to suspend and restart the data flow.

23.3.3 Which Flow Control Method Should I Use?

Like most simple questions, this one has no absolute answer. Both methods have advantages and disadvantages. COMM.DRV has built-in support for both types of flow control.

The advantage to software flow control is that it requires no extra signals to operate. The disadvantage is that it requires software overhead to execute, so it can be slower and less reliable than hardware flow control. More importantly the software flow control is limited to situations where the characters that are used are available, that is, they are not data. So, software flow control is not appropriate when transferring binary data. In fact, one file-transfer protocol (Kermit) goes to great lengths to allow software flow control when transferring binary data. This results in a substantial reduction in performance.

An advantage to hardware flow control is that it is fast, because the UARTs themselves interpret changes of state in the input signals and can generate an

interrupt that COMM.DRV can react to immediately. Hardware flow control is out-of-band, so it is inherently compatible with binary data. But, the extra control signals required need special handling when it comes to using them with data sent over the telephone system.

So, the answer is to use software flow control only if the system that you are communicating with requires it. Otherwise, use hardware flow control. The advantages of hardware flow control are significant, and as we will see in the chapter on modems, there are ways to make it work over the telephone system.

23.4 The UART

A significant drawback of data communications techniques that send 8 or more bits in parallel (such as the PC parallel port, GPIB, or SCSI) is that the cables contain anywhere from 25-50 wires and thus are both bulky and expensive. This can pose a severe problem, especially if the cables are long. One good solution to cabling problems is to convert the bytes to be transmitted into a serial bit stream and send them out on a single wire. In many cases, you can then get away with only two wires for bi-directional communications, one for each direction, although usually you need a ground wire connecting the two devices as well. Serial communications also have the advantage of being very well standardized and having widespread support on virtually all computer systems including PCs, workstations, and minicomputers.

You would expect serial communications devices to send data at a slower rate than parallel communications devices because they send only 1 bit at a time instead of 8. In many cases this is true, and the serial port on a PC does indeed send data about ten times more slowly than the parallel port. However, serial communications can provide extremely high speed. The reason is that since you need only a single transmitter, a receiver, and three wires for a serial link, you can afford to use sophisticated, relatively expensive components for all these items. This is why computer networks like Ethernet and fiber optic links give very high performance even though they use serial bit streams.

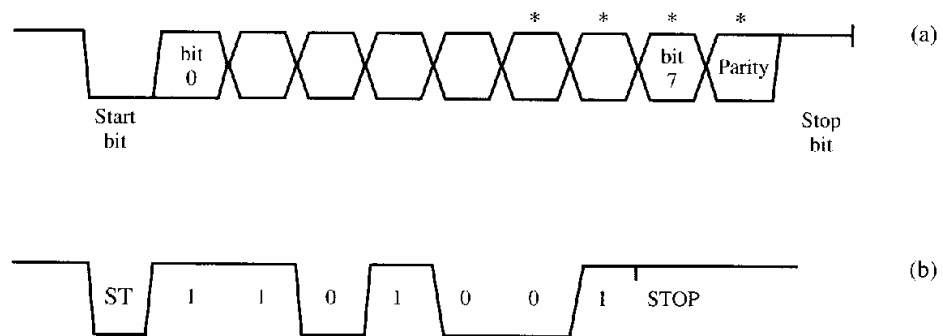
23.4.1 Basics of Asynchronous Serial Communications

The information in a serial bit stream is contained in its time-dependent waveform: the bits are represented by codes that are each transmitted for a fixed time period. The time period used to transmit each code is known as a baud period. The word baud is used in honor of a Frenchman named Baudot, who studied various serial encoding schemes in the 1800s.

The serial bit streams generated by PC serial ports use a very simple form of encoding. One bit is transmitted during each baud period, with a "1" bit represented by a TTL high voltage and a "0" bit by a TTL low voltage. Thus, the baud rate ($1/[\text{baud period}]$) of a PC serial port is equal to the number of bits per second being transmitted or received. More complex encoding where multiple bits are transmitted during each baud period are used by modems to send data over the phone lines.

To send information encoded this way, the transmitter and receiver clocks, which define the baud period, must be the same frequency and be synchronized. You'll see later how this is done. Bits are transmitted as separate groups, typically 7 or 8 bits long, called characters. The name character is used because each group of bits represents one letter of the alphabet when ASCII-encoded text is being sent. Each character is sent in a frame consisting of a "0" bit called a start bit, followed by the character itself, followed (optionally) by a parity bit, and then a "1" bit called a stop bit. The logic low start bit tells the receiver that a frame is starting, and the logic high stop bit denotes the end of the frame.

This approach to transmitting serial data is called asynchronous serial communications because the receiver resynchronizes itself to the transmitter using the start bit of each frame as discussed later in this section. New characters can be transmitted at any time, with an arbitrary time delay between characters. There are also synchronous serial communications protocols where characters are sent in blocks with no framing bits surrounding them. In this approach, the transmitter continuously transmits signals, with a special sync character being transmitted whenever there's no real data available to transmit. IBM mainframe computers have traditionally used such protocols, binary synchronous communication (bi-sync) and synchronous data link control (SDLC) for example, to connect with terminals and PCs.



The figure above show the time-related waveform for an asynchronous communication (a) Asynchronous serial bit stream format. Each character is preceded by a logic low start bit, which synchronizes the receiver and transmitter clocks. The character (5, 6, 7, or 8 bits) follows, least significant bit first. The * indicates optional bits. An optional parity bit and one or more stop (logic high) bits terminate the character. (b) Example of a serial bit stream for the ASCII character "K" (4bh) using 7 bits, no parity.

Notice that the data bits within each transmitted character are sent with the least significant bit first, each bit lasting one baud period. Serial receivers and transmitters can be instructed to send or receive as few as 5 or as many as 8 bits per character (but they must both agree on how many!). Often 8 bits are used so that a character contains an entire byte.

After each character's bits are sent out, an optional parity bit may follow. The parity bit is useful if the data line is too noisy to provide completely accurate transmission. The parity bit, P, can be chosen to give either even or odd parity. For even parity, P

1 if the number of 1s in the character is odd and $P = 0$ if the number of 1s is even. That is, P is chosen so that the number of 1s including P is even. For odd parity, P is chosen so that the number of 1s including P is odd. The local receiver checks to make sure that the parity is still the same in spite of any noise picked up by the cable. If the parity has changed, some bit has flipped its lid, and the receiver sets a parity-error flag in its status register (which the CPU can read if it wants to).

After the character and parity bits, the transmitter inserts one or more high stop bits into the data stream. Basically the line must come high long enough to allow the receiver to ready itself for the next start bit. Typically, one stop bit suffices, although transmitters can be instructed to insert 1, 1.5, or 2 stop bits

under program control. When no characters are being transmitted, the line remains at the logic high level of the stop bit.

Note that at least two (1 start and 1 stop) extra bits are required to transmit asynchronous data. So if you want to transmit whole bytes without parity (a very common choice), you actually need to transmit 10 bits. Just as in any business, you have to pay for overhead! A simple rule of thumb for estimating serial data transmission speeds is to divide the baud rate by 10. However, data compression (see Section 12-6) can increase the effective transmission rate significantly.

The way the receiving device stays synchronized to the transmitting device so that it can read the bits correctly deserves some comment. It's not obvious how this can be done, since the receiver and transmitter have independent clocks that are only nominally the same. Furthermore, the relative phases of the two clocks can be any value whatsoever. Also, the logic level changes at the beginning of each baud period can be shifted in time, owing to the limited bandwidth of the carrier medium. The standard solution to this problem is to have the receiver and transmitter use internal clocks whose frequencies are 16 times the baud rate. Then when the leading edge of the start bit is detected, the incoming serial waveform is sampled every 16 clock periods, starting with the eighth clock period after the leading edge of the start bit. This ensures that the waveform is always sampled near the middle of every baud period, making it tolerant of small edge shifts and transmitter/receiver clock frequency differences.

The standard baud rates are: 50 (ham radio-some people can decode this speed by ear!); 110 (yuck! it's an ancient teletype, better known as a clunk-clunk); 134.5 (ugh! it's an obsolete IBM 2741); 150 (way too slow); 300 (way too slow, but OK for low-rate applications like credit-card verification over phone lines), 1200 (still too slow), 2400, 4800, 9600, 14,400, 19,200 (now you're talking!), 38,400, 56,000 (actually 57,600 on PCs), and the nonstandard 115,200 (not available on the original PC). PCs, printers, and other devices often cannot function at the highest of these rates. However, if you hook up two PCs using the MS-DOS interlnk/intersvr pair and a serial port, you may well find that they use 115200 baud. Using an 8-bit character with no parity and 1 stop bit, there is a total 10 bits, so a transmission rate of 115200 baud yields a throughput of 11.5K/sec.

23.4.2 UARTs and the PC Serial Port

Section 10-5 shows that a computer can generate many kinds of waveforms. In particular, it can convert a byte into an asynchronous serial bit stream and send it to some pin of a parallel port. The output bits are then buffered so that the voltage and current levels conform to one of the conventions described in Section 23-5, then go out to a terminal or modem. Some of the very earliest personal computer systems, such as the Commodore 64, actually used this software method for serial communications. The trouble is that it really ties up the CPU and prevents the use of higher speed transmissions.

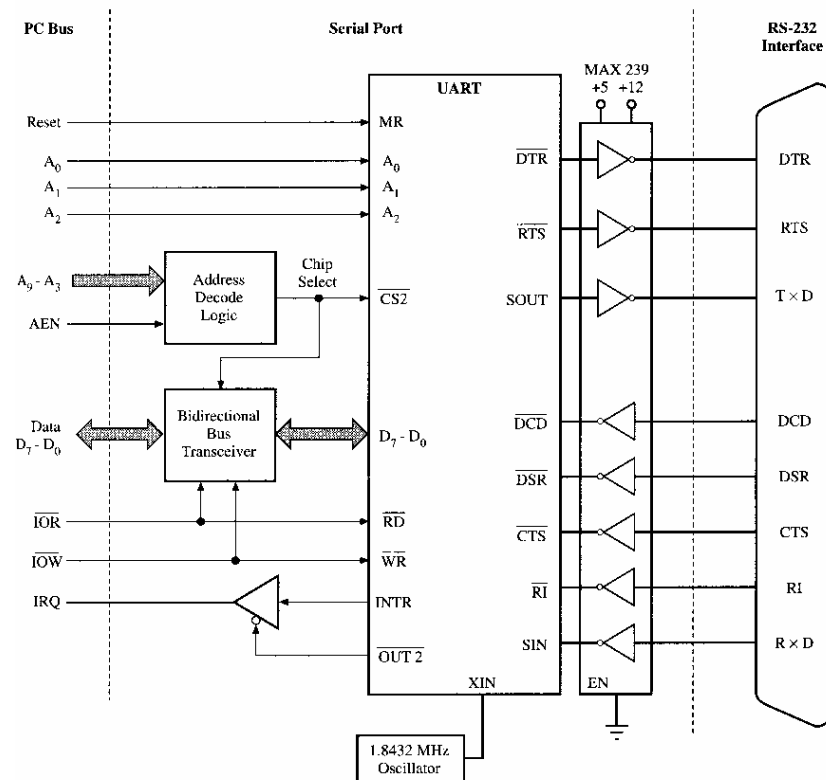
One of the earliest types of large-scale integrated (LSI) circuits was the universal asynchronous receiver transmitter (UART, pronounced "you-art"). This special purpose chip was developed to simultaneously transmit and receive serial data, perform the appropriate parallel/serial conversions, and insert or check the start, stop, and parity bits used to keep the serial data synchronized. A transmitter inside the UART converts bytes sent as 8-bit parallel data to the UART into a standard-format serial bit stream for transmission. The circuitry inside the UART that does this is basically just a parallel-in/serial-out shift register. Similarly, an incoming serial bit stream is detected by a receiver inside the UART and converted into parallel data by a serial-in/parallel-out shift register. The resulting data bytes appear as 8-bit parallel data that can be read out from the UART. The word asynchronous appears in the acronym UART because it supports asynchronous serial communications, and the name universal appears because the UART can work with all popular asynchronous serial formats.

Simultaneous conversion of an incoming and an outgoing serial data stream is called full duplex communications, which requires two separate signal lines to carry the data. A complete connection can be implemented with three wires: one for the outgoing data stream, one for the incoming stream, and the third for a common ground line. In some (uncommon) situations, half duplex is sometimes used. This allows two-way communications (hence, duplex), but only one direction is active at a time. It's similar to using walkie-talkies, for which you have to say "Over" when you're finished talking and want to let the other person talk back. The advantage of half duplex is that only one data channel is required. All UARTs provide for standard full duplex communications.

A serial port on a PC is little more than a UART that is directly connected to the PC bus. The figure below gives an example of how a typical serial port for a PC is constructed. Aside from a little bit of glue logic and a clock for the UART, the only additional hardware needed to make a complete serial port adapter card for a PC is the interface circuitry that converts the UART's TTL-level serial

input and output signals into RS-232 signals. The UART is controlled by the PC through a set of I/O ports that read from or write to the UART internal registers. Once you understand how the UART used in a PC operates, you'll understand PC serial ports.

If you actually look at a typical multifunction board containing one or more serial ports, you may be surprised to find that there's no UART chip anywhere in sight. The reason is that it's been buried inside a custom application specific integrated circuit (ASIC) that contains all the circuitry for two serial ports, a parallel port, and perhaps hard and floppy disk interfaces. Nonetheless, a clone of one of the UARTs described in the next subsection (we hope it's a 16550) is present inside the ASIC.



The above picture shows the block diagram for a PC serial port.

23.5 RS-232 and Other Serial Conventions

The previous section shows how a UART converts parallel data in the computer to and from serial bit streams. However, the TTL output of a UART like the 16550 can't be transmitted error-free over any substantial distance. The 16550's serial output doesn't have adequate drive power and, in any event, TTL serial bit streams can work reliably only over distances of a couple of meters due to noise pick-up and signal distortions induced by the cabling. To send bit streams over long distances, it's a better idea to convert TTL bit streams into some other form. By far the most common way of doing this is to use the signaling method specified in a standard known as Recommended Standard 232 (RS-232). An alternative technique is 20 mA current loop, which is useful when extremely long wires or very high noise immunity are needed. Current loop is briefly discussed at the end of this section.

23.5.1 RS232

RS-232 is an Electronic Industries Association (EIA) standard that specifies the electrical characteristics, connector requirements, and signal functions for a serial interface. The current version of the standard, adopted in 1986, is RS-232-D. Instead of TTL levels, the logic levels on an RS-232 line are -3 to -15 volts for a logic 1, and +3 to +15 volts for a logic 0. ± 12 volts are typically used in PC systems. This gives a larger voltage swing as well as a zero crossing and is a much more noise-immune signaling scheme than TTL. The signals are sent over a cable that can be any length, provided the total capacitance of the cable is less than 2,500 pF. The earlier versions of the standard specified a maximum cable length of 50 feet; with high quality, low capacitance cable, this length can easily be doubled. The serial data rate is allowed to be as high as 20K bits per second. In practice, much higher data rates are often used (up to 115,200 bps) with good success, provided the cable length is kept short (1 or 2 meters).

The RS-232 standard also defines a standard connector for serial communications, namely a 25-pin D-shell connector, also known as a DB-25 connector. Like most connectors, this connector comes in two sexes, male and female. The male version, also known as a DB-25P connector, has 25 pins arranged in two rows. The female version, also known as a DB-25S connector, has 25 little holes into which the pins of the male connector fit. On the original IBM PC, the serial port connector on the back of the machine is a male. Unfortunately, IBM also used a DB-25 connector for the parallel port output (but female instead of male), making it easy to confuse the two.

Somehow the writers of the RS-232 convention also managed to define signal functions for every single pin on this 25-pin connector! The standard supports not only the DTR, DSR, RTS, and CTS handshake lines but a host of other signals as well, such as transmit and receiver clock and secondary copies of RTS and CTS. Fortunately, a maximum of only ten pins are typically used, three pins isn't uncommon, and at low data rates you might even be able to get by with only two if you're willing to cut a few corners!

The commonly used RS-232 signals and their pin numbers are shown in the table below. Two types of interfaces defined here: a data terminal equipment (DTE) interface and a data communications equipment (DCE) interface. This scheme dates back to the early days of time-sharing computers when you had a single large mainframe computer to which you connected a terminal via a serial interface (or if you were at some other location, you connected a terminal to a modem and communicated over the phone lines). The intent here is that you would always connect a DTE interface such as a terminal to a DCE interface such as a mainframe computer or modem. As you can see from the table, the difference between DTE and DCE interfaces is that the transmit and receive pins, the RTS and CTS handshake signals, and the DTR and DSR handshake signals are interchanged. This allows you to properly connect a DTE to a DCE interface using a straight-through cable where pin 2 goes to pin 2, pin 3 to pin 3, and so on.

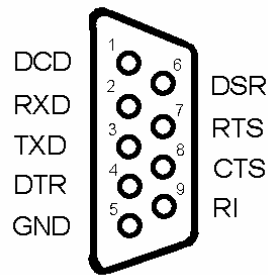
RS-232 Signal		Direction	DTE	DCE
Signal Ground			1	1
Transmit Data	(TxD)	Out	2	3
Receive Data	(RxD)	In	3	2
Request To Send	(RTS)	Out	4	5
Clear To Send	(CTS)	In	5	4
Data Terminal Ready	(DTR)	Out	20	6
Data Set Ready	(DSR)	In	6	20
Data Carrier Detect	(DCD)		8 (In)	8 (Out)
Ring Indicator	(RI)		22 (In)	22 (Out)

Chassis Ground	7	7
----------------	---	---

The idea of having different DTE and DCE interfaces might have been reasonable in the 1960s, but, unfortunately, life is not so simple in today's world, and we often want to do things that the designers of the standard never envisioned. Adding to the trouble is IBM's choice of serial interface type for the PC. Being a computer, it's obviously a DCE interface, right? Wrong! In IBM's eyes, the original PC was a terminal device, meant to be connected to a real computer like an IBM 370 mainframe. As a result, all PCs have DTE serial interfaces. This makes it easy to connect to a modem, since they all have DCE interfaces, but what do you do if you want to do something crazy like connecting two computers together by a serial link? The answer is to use a special type of cable known as a null modem, which is wired with pin 2 on one end connected to pin 3 on the other end, 3 connected to 2, 4 to 5, 5 to 4, 6 to 20, 20 to 6, and 7 to 7. The RI and DCD wires aren't needed. This arrangement gets all the correct signals connected to each other. Another problem that sometimes needs to be solved is what to do when the ends of your cable don't have the right sex for the connection you want to make. Here you can use a device called a gender changer, which consists of two male or two female DB-25 connectors connected by short straight through wires.

We've often found it useful to make a female-female gender changer using null modem wiring between the connectors. This enables you to connect the IBM PC's serial port connector to other DTE devices like another computer using a standard straight-through male-male cable. Because the number of possible ways of assigning and using RS-232 lines is so large, you may want to buy a device called an RS-232 breakout box to determine what signals go where. Ironically, RS-232 has earned the distinction of being the most "nonstandard standard" in electronics!

One very useful thing IBM did was to introduce a new 9-pin DB-9 male connector for the serial port on the PC-AT. This connector is considerably smaller than a DB-25, but still carries all the necessary signals. Its pinout has been formalized as EIA-574 and is shown in the image below. Nearly all PC serial interfaces now use this connector. For really tight spots, 8-pin RJ-type telephone connectors are also popular.



A really minimal RS-232 cable just connects up pins 2 and 3 appropriately and depends on the built-in chassis grounds of the equipment to complete the circuits. In fact, we've run 9600-baud communications around a house that way, although the dedicated ground line on pin 5 should normally be connected as well.

In order to translate the TTL signals coming from a UART like the 16550 into RS232 signals, special driver and receiver circuitry needs to be used. The parts used to do this in the original IBM PC and PC-AT were the 75150 dual RS-232 line driver and the 75154 quad RS-232 line receiver. The 1488 quad driver and 1489 quad receiver were also popular parts in early serial port boards. However, the 75150 or 1488 drivers are both inconvenient to use because they require ± 10 -volt or ± 12 -volt power supplies. Newer RS-232 drivers contain built-in circuitry to generate the -12-volt negative supply voltage internally from +12-volt using a voltage inverter circuit, and some versions also generate $\div 10$ volts from +5 volts using an internal voltage doubler. The newer ICs also combine drivers and receivers in a single package so that only a single chip needs to be connected to a UART to implement an RS-232 serial port. Note that the RS-232 drivers and receivers both are inverters so that the signals on a serial cable are inverted from their values at a UART. Thus, the handshake lines are all active high when observed on the cable. Note also that this interface works even if the cable doesn't connect all the handshake signals since input lines are pulled high by the receiver circuitry if their inputs are disconnected. Many interfaces work this way, requiring you only to connect pins 2, 3, and 5 (RxD, TxD, and signal ground) on a DB-9 connector to make a working serial interface.

To initially check out the operation of a serial hookup, connect the TxD serial output line on the serial connector to the RxD serial input line. Your serial port should then act like a slow, expensive 1-byte memory—you should be able to read back a byte that you send. If not, try connecting the UART's SOUT directly to SIN. If the serial port still doesn't act like a memory byte, check your UART

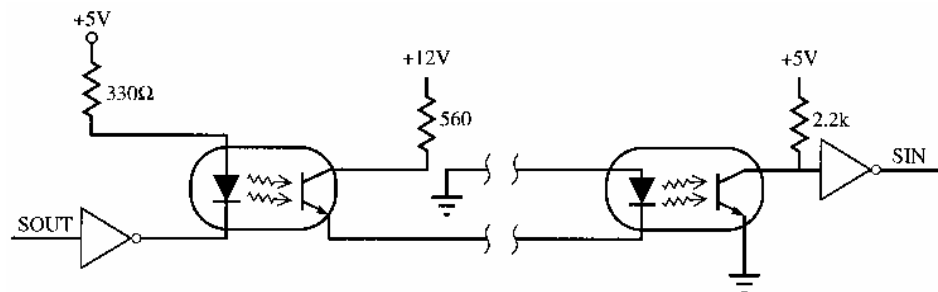
wiring and initialization code. Once you've gotten everything working this way, connect up your two devices and start worrying about the handshake lines if it doesn't work.

Here's a useful trick: you can tell whether a given pin on an RS-232 connector is being driven by measuring its voltage with respect to the signal ground on pin 5 (or pin 7 on a 25-pin connector). The output data line (2 or 3) will have about -12 volts and the input data line (3 or 2) will have zero or a positive voltage. A driven handshake line such as DTR or RTS will have about +12 volts on it.

If you don't want to use all the handshake lines in an RS-232 link or the device on one end doesn't support the same handshake as the device on the other end, you can dummy out various control lines. Specifically, connecting pin 7 to pin 8 on a DB-9 connector dummies out the RTS/CTS protocol; connecting pin 4 to pin 6 dummies out the DTR/DSR protocol. You may also have to dummy out the carrier detect line on pin 1 by connecting it to pin 4 (to make sure it's pulled high).

23.5.2 Current Loop and Other Serial Standards

An alternative to RS-232 is 20-mA current loop, a technique from early telegraphy. At low data rates, current loop signals can go across the country, which is why current loop was used for the telegraph. More realistically, you can easily run current loop at 9600 baud over a thousand feet of wire. Another advantage of current loop is that it's normally implemented using opto-isolators, which prevents wiring mistakes or pickup in very noisy environments from doing any damage. Note that the digital current loop connection being discussed here is different from the 6-20 mA current loop signaling that's widely used to transmit analog signals in industrial environments.



The figure above shows a current loop connection of one UART's serial output (SOUT) line to the serial input (SIN) line of a different UART some distance away. Both UARTs are opto-isolated from the current loop connecting path, which has its own power supply. The current loop supply voltage and resistor are chosen to maintain about 20 mA flowing in the loop when the transmitter (labeled TxD in the figure) is high. A low value yields no current flow. Therefore, a 1 is represented by a baud period of 20-mA current flow and a 0 by a period of no current. Two current loops of the kind shown in the figure above are needed for full duplex operation, so four wires are required.

Ordinarily, current loop doesn't support the modem control signals DTR, DSR, RTS and CTS, so you just tie CTS and DSR low on the UARTs. To create a handshake, you can use the XON/XOFF protocol described in Section 23-3. Although current loop has its place for special situations, it's not used very often. There is no standard for the connector pinout of a current-loop connection and it can be tricky to get a current-loop power supply connected up on one end (not both!) of each current loop. You may also have difficulty finding a serial port that supports current loop, although the original IBM asynchronous communications adapter (serial port board) for the IBM PC does.

23.5.3 RS422 / RS423

In an effort to overcome some of the length and speed limitations of RS-232, the EIA has developed several other serial communication standards. Although RS-232 is a complete specification for a serial interface and includes both the electrical and the mechanical (connector) specifications, the newer standards are split into two pieces. Two new electrical standards, RS-422 and RS-423, define only the signal levels, signal rates, cable characteristics, and set the driver and receiver specifications. A single new mechanical standard, RS-449, defines the connectors and pinouts to be used with RS-422 and RS-423. RS-423 defines driver output signal levels of +4 to +6 volts for a logic 0 and -4 to -6 volts for a logic 1. The receivers have a sensitivity of ± 0.2 volts, meaning that they can detect a logic 0 as any voltage greater than +0.2 volts, and a logic 1 as any voltage less than -0.2 volts. RS-232 receiver sensitivities are only required to be ± 3 volts. The lower drive voltage and greater receiver sensitivity allow the RS-423 cable lengths to be up to 4000 feet at 100,000-baud data rates. RS-422 uses differential signaling instead of the unbalanced (single-ended) signaling used for RS-232 and RS-423. Here a driver with two outputs and a pair of wires is used for each signal, with a voltage difference of +2 to +5 volts (for logic 0) or -2 to -5 volts (for logic 1) between the two wires being the signal. The

maximum cable length and the receiver sensitivity are the same as for RS-423, but the greater noise immunity of differential signaling allows the maximum data rate to be increased to 10 mega-baud. RS-485, an enhanced version of RS-422, extends RS-422 to use tri-state drivers and provides a mechanism for having multiple drivers and receivers on a single cable.

The RS-422 or RS-423 electrical interface standards are meant to be used in conjunction with a mechanical interface standard, RS-449, to implement a serial communications link. Unfortunately, the connector that was specified for RS-449 is a 37-pin D-type connector that is considerably bigger than the already large RS-232 25-pin connector. Furthermore, an additional 9-pin connector and cable is specified to implement a secondary serial channel (although this is optional). This is such a bulky and awkward mechanical design that it's virtually killed the use of RS-449 (along with RS-422 or RS-423) except for a few specialized uses. RS-232 still continues to be the dominant standard. One place where RS-422 is used is in Apple Macintosh computers. However, they don't use the RS-449 connector. Instead, a 9-pin D-type connector was used on the original Macintosh, with only the transmit and receive wire pairs connected up, plus a single CTS signal wire. Starting with Apple Macintosh IIs, the connector was changed to a circular 8-pin DIN-style connector and an additional DTR signal line was included.

23.6 Cabling

A lot of confusion exists on how to make cables between serial devices. This confusion is created because there are basically 2 kinds of devices (DTE and DCE) and they have a different pinning.. When connecting a DTE (computer) to a DCE (for example a modem) then you use a straight cable. That means you connect pin 1 to 1 , 2 to 2 , 3 to 3 and so on. IF you want to connect 2 DTE devices together you need what is called a null-modem cable. This name comes from the early days of computing where modems were used to connect computers together. If you delete the modems you get a direct connection thus the name "null-modem" meaning 'direct-but-without-modems'.

23.5.1 Null modem cable

To connect two device together you will need what is called a Null Modem cable. This is the simplest cable to connect 2 devices.

Pin		Pin
2	-	3
3	-	2
5	-	5
1 4 6		1 4 6
7 8		7 8

If make a cable with the above pinning (9 pole D connector) , then you can connect virtually any device to a serial port. The cable ‘emulates’ the handshaking by connecting the appropriate lines together. Actually what happens is that the UART sees his own signals and thinks someone else is out there.

23.5.2 Full connection Null Modem Cable

Pin		Pin
2	-	3
3	-	2
4	-	1,6
1, 6		4
5		5
7		8
8		7

The wiring given above forms a complete Serial communication link. Not only TXD and RXD are crossed but the handshake lines as well.

23.6 Basic Serial Operations using MSCOMM

To program the serial interface you could of course start controlling the UART directly. Fortunately , windows offers us help here under the form of MSCOMM.DLL . This standard component of the Windows Operating system handles all task related to serial communications. Besides initializing the UART it also handles interrupts coming from one or more serial ports , and it implements a software FIFO as well.

On top of this engine is an object that allows you to interface directly to the MSCOMM.DLL library. How to use this has been already explained in Part II of this manual under chapter 14 item 3.

Chapter 24 :

Plug-In boards

Besides the usage of these standard IO channels such as printer ports and serial ports you can always construct your own ISA or PCI card. I am not going to deal with the PCI bus here since it is too difficult to explain in a few pages. I Could write a book about this bus that is twice the size of this one. Furthermore the constraints imposed by the PCI specification make it very hard to construct this kind of board yourself. Another point is that the access of PCI devices uses in a different manner then accessing a standard ISA board. You mostly need special drivers for these cards and they are manufacturer supplied and card specific. So , to control PCI boards you need to stick to whatever the board maker delivers you.

24.1 Description of the ISA bus

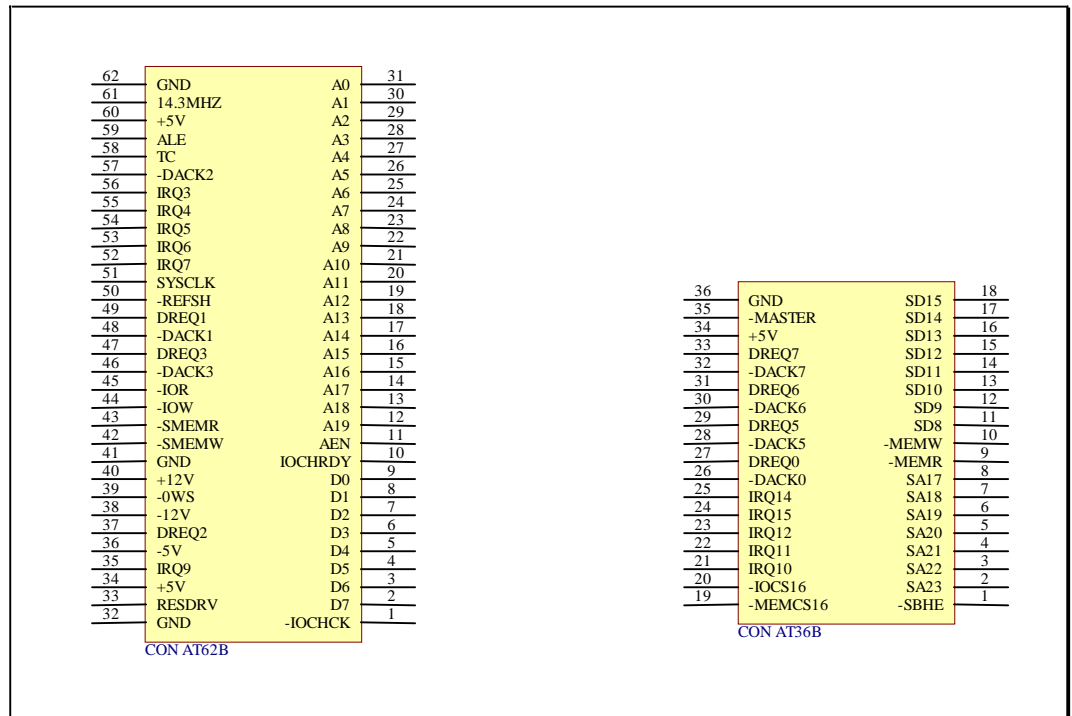
In Chapter 20 I gave you an overview of the most common buses that are in existence today. The ISA bus is one of these buses. You can find a plethora of cards that host a variety of different functions. Besides standard computer functions such as Floppy and hard disk controllers ,Serial ports, Printer ports , Video adapters and audio boards , there are indeed special boards for test and measurement functions.

These boards cover the whole spectra of T&M , from simple Digital IO , multiplexers , A/D-D/A converters to complete multi-meters and even oscilloscopes.

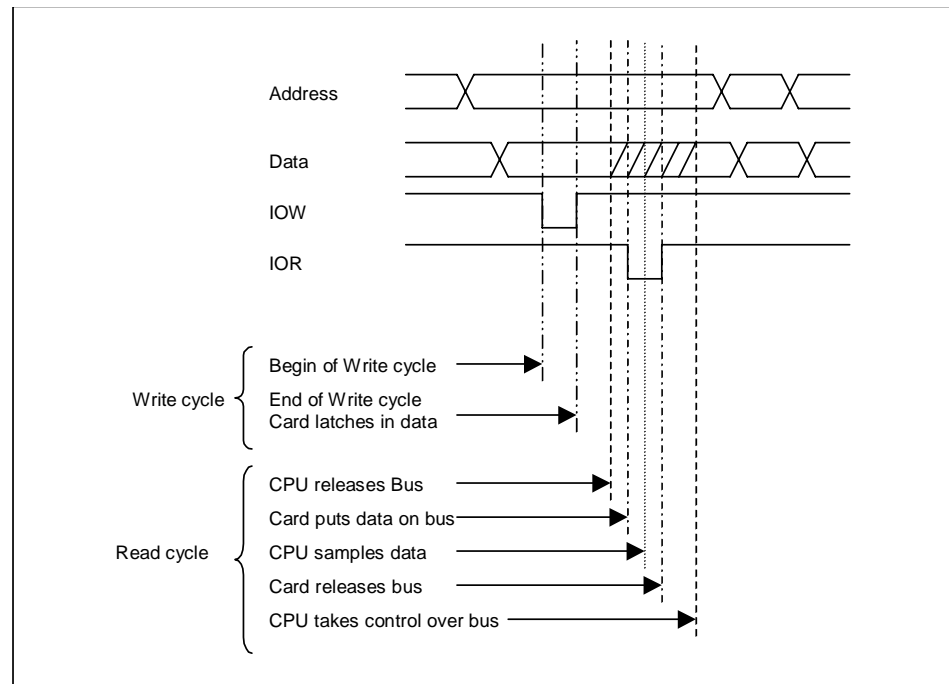
Complex boards such as multi-meters and oscilloscopes will again come with vendor supplied driver software. Simple cards mostly also have these drivers but

not necessarily for all languages. And exactly for these boards , and the boards you might construct yourselves this chapter is important.

The graphic below shows the pinning of the standard ISA bus. A lot of these signals are not generally used for IO board. Actually only very few signals are used for that purpose.



The picture above shows both connectors and their pinning. Describing on how to make a full interface would go way too far for this course. However the basics of this bus are very simple. Using the Address bus the IOR and IOW lines a simple interface can be made. When the address of the decoder matches the address on the data-bus the card is selected. Depending on IOR or IOW a read or write operation is going on. During the IOR pulse the your card must be put data on the data-bus. You can use the Rising edge of IOW to latch in data from the data-bus.



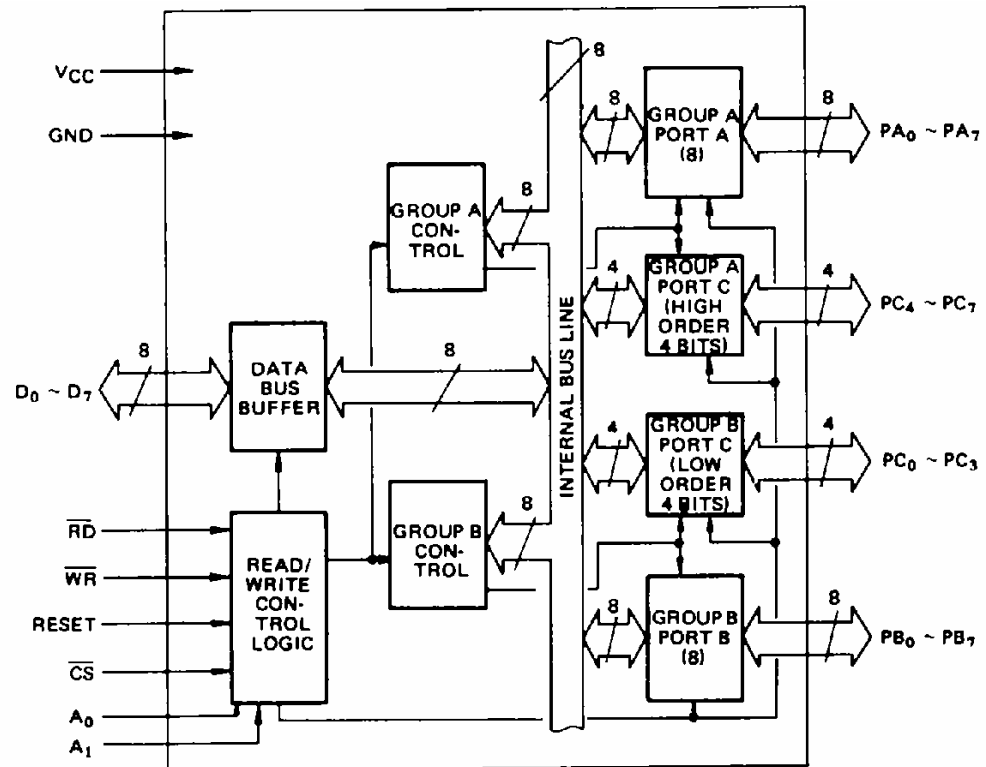
The image shows exactly what is going on during these read and write operations. The most important to remember is to sample in data on the rising edge of IOW and to put data on the bus when IOR is going low and leave it there until IOR is going high again.

24.2 common interface chips

Before we start writing code we will take a look at some common interface circuitry used on such cards. Understanding these will make it much easier to unravel the inner workings of the card and write a library of for it.

24.2.1 8255

This is probably the most widely used interface circuit around. It offers a simple way to add 24 IO lines to a computer.

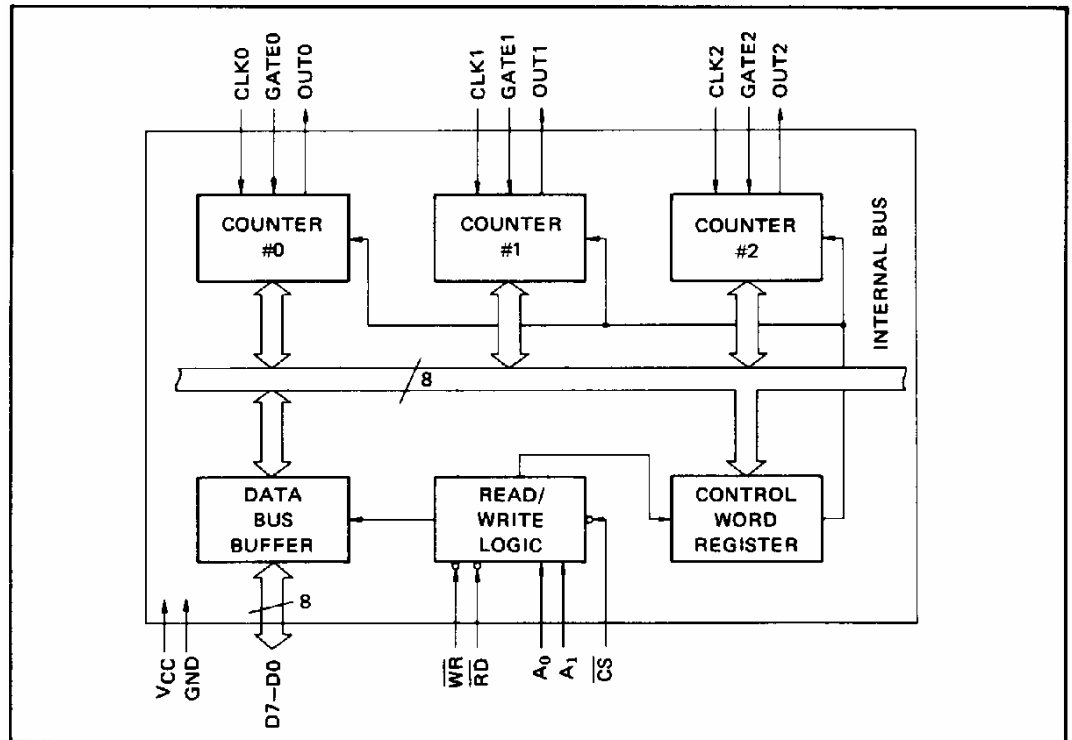


The image above shows you what is inside the 40 pin DIL or 44 pin PLCC package. The chip can be divided into three parts. Each part is a basic 8 bit IO channel that be written to or read from.

The option exists to split the third channel into two parts and merge these parts with one of the two remaining ports, thus creating 2 12 bit ports. In this mode the extra 4 lines can be used as control lines to orchestrate data IO on the two other ports. Port A has also true bi-directional capabilities. The complete explanation of this chip can be found in the Datasheet appendix.

24.2.2 8253/8254

This is the most used complement to the 8255 on IO boards. This chip features three 16 bit counters/dividers. The figure below shows you the inner parts of this chip.



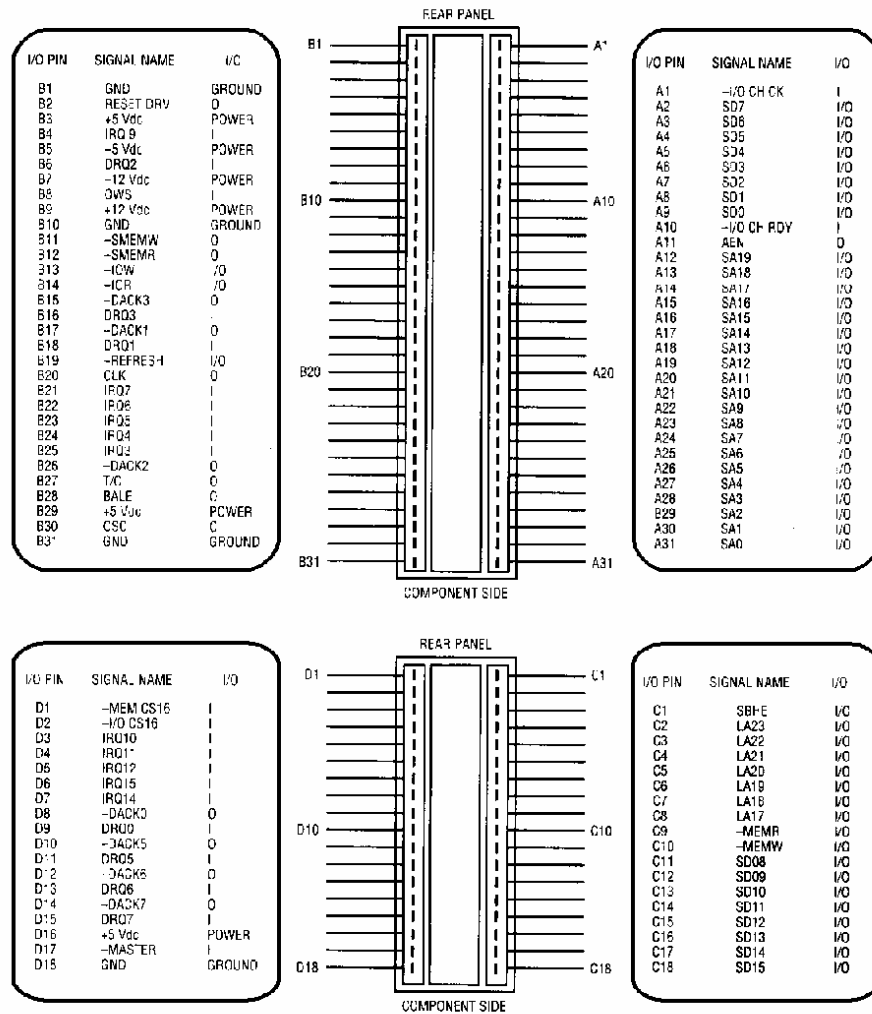
Just like with the 8255 a number of configurations are possible. Each counter can be used independently either as frequency generator, event counter or programmable divider. By cascading the counters you can make a 32 or 48 bit counter/divider. The typical use for this component is measuring time intervals, frequencies or simply the number of times a certain event occurred. Programming this component is pretty straightforward. More information can be found in the datasheet appendix.

24.3 Interfacing to ISA

Beginning about a year after IBM introduced it, other manufacturers quickly created clones of the PC-AT, and they also incorporated the PC-AT's I/O channel bus. PC-AT clones using this bus, which soon became known as the AT bus, were tremendously popular. Finally in 1987, the IEEE approved an AT bus specification that included bus timing requirements. This bus specification defines what is now known as the Industry Standard Architecture bus, or ISA bus for short.

Even when microcomputer systems based on higher-speed 80286 CPUs or on even faster 386 and 486 and Pentium CPUs became available, the ISA bus was retained so that the machines could use the thousands of different ISA bus cards being produced by literally hundreds of manufacturers. Furthermore, in order to avoid rendering the millions of existing ISA bus cards obsolete, the ISA bus speed was de-coupled from the CPU speed and left at 8 MHz.

In order to understand the ISA bus, it's essential to realize that there are two distinctly different types of ISA bus cards: 8-bit cards and 16-bit cards. Eight-bit ISA cards only plug into the rear 62-pin primary connector and are unaware that the additional signals of the auxiliary connector exist. This means that they see only eight data lines and thus can only do 8-bit data transfers.



PC AT 16-bit extension slot.

PC AT Expansion Bus slots signal/pin definitions

SD0-SD15 - System Data Bits 0 through 15
 SA0-SA23 - System Address Bus bits 0 Through 24
 IRQ2-IRQ15 - Interrupt Requests Levels 2 through 15
 DRQ0-DRQ7 - Direct Memory Access Requests 0 thru 7
 DACK0-DACK7 - Direct Memory Access Acknowledge 0 thru 7

Note: The remaining signals provide Bus timing information, status and commands. For a detailed description of those signals consult Interfacing to the PC (SAMS 1995) or the IBM Technical Reference Manuals.

The above picture shows a graphic representation of the ISA connectors and the signals present on each pin. For the purposes of pin identification on the 62-pin primary ISA connector, the front, or component side of the edge connector (the side of the board with components mounted on it) is called the A side, and the back side is called the B side. Similarly, for the 36-pin auxiliary connector, the front side of the edge connector is called the C side and the other side the D side.

24.3.1 Address & Data Lines

While the ISA bus supports a 16MB address space, which requires 24 address lines, we see from Table 7-7 that there are actually 27 address lines present! These lines are divided into two sets: the 20 bits SA0-SA19 on the primary ISA connector and the 7 bits LA17-LA23 on the auxiliary connector. Thus there are three address bits, SA17-SA19 and LA17-LA19 that overlap in the two sets. The reason for this overlap is that the ISA bus controller needs to be informed early in the bus cycle whether an 8-bit or 16-bit memory access is to be done.

24.3.2 Utility Lines

The utility lines include the various power supply voltages as well as the OSC, RESET, and IOCHK signals. Note that while +5V is all that's needed for digital logic, three other power supply voltages are present to power interface circuitry. The presence of the $\pm 12\text{V}$ power lines is particularly useful since many analog devices like op amps use signal levels in the range $\pm 10\text{V}$ thus and require $\pm 12\text{V}$ power to operate. If you do need $\pm 12\text{V}$ or -5V power for an I/O card, you may need to check its current requirements against the capabilities of your PC's power supply. Most PC power supplies are rated at about 200-250 watts, but nearly all of this power is at +5 volts (about 20-25 amperes) and +12 volts (about 8 amperes). There's typically only a few tenths of an ampere of current available from the -5-volts and -12-volts supplies. The high current available at +12 volts is there to drive the floppy- and hard-disk spindle motors.

The OSC signal is a free-running (meaning its rising and falling edges aren't synchronized to any other signals) square wave whose frequency is 14.31818 MHz. This signal is a relic from the original IBM PC that supported a color/graphics adapter (the CGA card) that needed a clock signal at this

frequency. When divided by four, 7-7 14.31818 MHz gives the 3.579-MHz color sub-carrier frequency used for NTSC color-television signals. The OSC bus signal serves no particular purpose in current PCs.

The RESET signal is asserted by the motherboard when the PC is first powered up or whenever a system-wide reset is done (for example, when you push the computer's reset button). All ISA bus cards to reset their on-board devices to a known, initialized state should use it.

The IOCHK signal is meant to be asserted by any ISA card when an unrecoverable hardware error occurs. Examples would be a parity error in the card's memory or the failure of some intelligent device on the card to properly initialize the on-card circuitry. Pulling IOCHK low triggers a non-maskable interrupt (unless setting bit 3 of port 61h to 1 has masked off the IOCHK signal).

24.3.3 Bus Cycle Definition Lines

Looking at the picture above we see that there are two memory read signals and two memory write signals on the ISA bus. The SMEMR and SMEMW signals were present in the 8-bit expansion bus of the IBM PC, to0. and they are asserted whenever a memory location in the first megabyte of address space is accessed. The other set of memory read and write lines, MEMR and MEMW, are asserted whenever a memory location above the 1 MB boundary is accessed. The reason for having two sets of signals is to ensure that older cards that contain memory and only plug into the 8-bit ISA connector don't respond to memory accesses with addresses above 1 MB. Without the extra set of lines, an 8-bit card, for example, would have no way to tell if a memory access was for address 0a0000h or for some address xxxa0000h above 1 MB, because it only sees address lines SA0-SA19.

The REFRESH line is asserted by the refresh circuitry when it's granted the bus to do a memory refresh cycle. These refresh cycles normally occur every 15 microseconds. Both SMEMR and MEMR are active during the refresh cycle so that both 8-bit and 16-bit ISA bus memory see the refresh request. The DRAM row to refresh in each memory chip is sent on the SAn address lines during this cycle.

24.3.4 Bus Control Lines

The bus control lines play a central role in determining what goes on during a given bus cycle. The time period defined by the bus clock line (BCLK) determines the basic time intervals used during the bus cycle. In particular, bus cycles always last an integral number of BCLK periods, and the various events that occur during the cycle are normally discussed with reference to which clock period they occur in. The nominal BCLK frequency is 8 MHz, corresponding to the clock frequency of early 80286 processors. To maintain compatibility with existing ISA cards, this clock frequency has remained constant since then, except that many PCs push the frequency to 8.33 MHz. This strange value came about because the first 25-MHz 386-based PCs required a double frequency 50-MHz clock and the designers of these PCs found it convenient to derive BCLK from the 50 MHz clock by dividing it by six to yield 8.33 MHz. While the BCLK frequency is fixed, the BCLK phase is not. It's resynchronized by the ISA bus controller circuitry so that its rising edge is always coincident with the start of a new bus cycle. As a result, the phase of the BCLK signal may appear to jitter if you try to look at it alone on an oscilloscope. The buffered address latch enable line (BALE) goes high for about 62 nanoseconds during the first clock period of a bus cycle to indicate that a valid address is being placed on the address lines SA0-SA19 and that the LA17-LA23 lines (which, confusingly, are asserted during the clock period before the start of a bus cycle) are stable and can be decoded by the ISA board to see if it's being selected.

The I/O channel ready line (IOCHRDY) and the no wait state line (NOWS) allow the ISA board to modify the number of wait states inserted into a bus cycle by the ISA bus control circuitry. The board de-asserts IOCHRDY (pulls IOCHRDY low) to indicate that it wants the bus controller to insert one or more additional wait states into the bus cycle. NOWS, on the other hand, is asserted by the board to indicate that the bus controller should remove the default wait states that are normally inserted into the bus cycle. This point is discussed further below when the ISA bus timing is presented.

The system bus-high-enable line (SBHE) is used to signal that the high byte of the data lines (SD8-SD15) carries valid data during the current bus cycle. As you might expect, the ISA bus controller asserts this line during 16-bit memory or I/O transfers starting at an even address. However, SBHE is also asserted for 8-bit transfers from an odd address (SA0=1). In this case, the combination of SBHE=0 and SA0=1 activates byte-swapping circuitry that swaps the bytes on the low and high half of the data lines. This behavior is necessary in order to properly perform 16-bit transfers that begin at an odd address, because the ISA

bus controller must do such transfers as a pair of 8-bit transfers (due to the fact that the transfer spans two memory words or two I/O ports).

The MASTER 16 line is asserted by an alternate processor such as a DMA controller located on the ISA bus when it's been granted control of the bus (this is done by using one of the motherboard's DMA controller channels to request the bus). Designing a board that can take over the ISA bus as a bus master is very difficult, and only a few manufacturers have been able to do it successfully. When such a processor does take over the ISA bus, it should not grab the bus for more than 15 microseconds at a time, or loss of DRAM memory contents may result due to lack of refresh.

24.3.5 Interrupt Request and DMA Lines

The interrupt request lines can be used to signal the CPU that data is waiting to be processed. The DMA lines are used to initiate DMA transfers between the card and the PC's main memory.

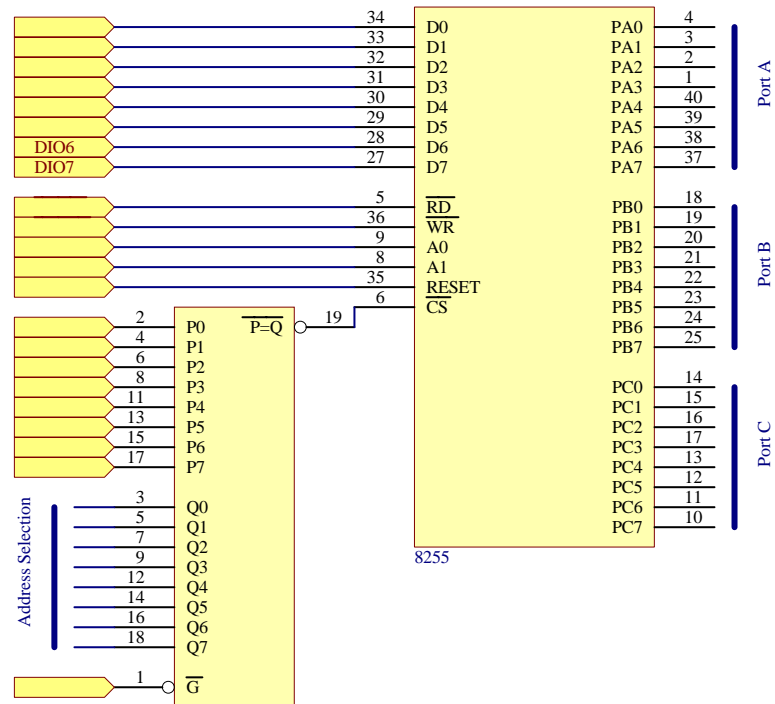
24.3.6 Basic interface schematic using 8255 I/O controller

The images below show two possible ways of building an interface with an ISA bus. First schematic uses the above described 8255 IO controller. This controller fits directly onto any Intel CPU architecture, so it is equally fit to connect with the PC's ISA bus.

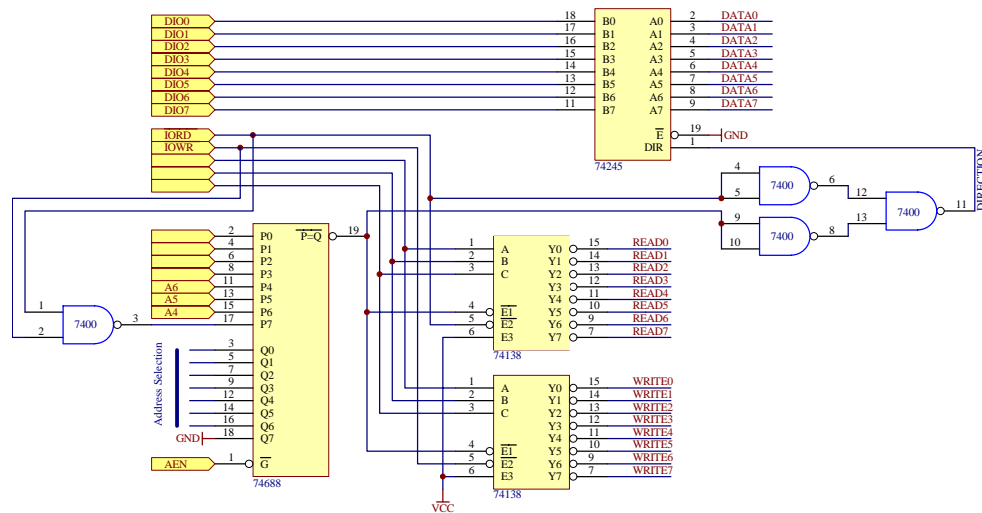
An 8 bit cascable comparator (74688) checks if the board is addressed. If address matches and the AEN line is low the Chip is selected. The RD, WR and two lowest address bits go directly to the IO controller. Depending on the state of these signals the controller decides what to do.

The address of the card is selected by strapping the address select inputs on the 74688 comparator to logic high or low.

Depending on the configuration sent to the IO controller various functions can be performed. See the chapter about the 8255 for more information or consult the datasheets in the appendixes.



24.3.7 Basic interface schematic using classic logic



This schematic shows how to make a full controller yourself. It has an address decoder similar to the one above to check the access to the card. But since there is no smart IO controller we have to make the decoding logic yourself. Two 74138 are used to decode the lowest 3 address bits into 8 strobe lines that can be used in our circuitry. Besides the address lines, these chips take the Card select signal from the 74688 and the Read or Write line. By implementing the circuitry like this one of the decoders reacts on write events and the other on read events. The 74245 buffer is put in the circuit to minimize the load on the bus. In the previous schematic the 8255 controller has this embedded in its system. Here we have to put it ourselves.

Note : Take care about the connection of A and B buses of the 74245. It seems odd but this trick is implemented to save an additional component. If you switch A and B buses (like you would expect) you would need to invert the output of the last NAND gate, thus using an additional component.

The construction using the 3 NAND gates checks what kind of operation is being performed. In case of a READ operation, it combines this with the card select and switches the direction of the buffer, so the data flows from our circuitry to the computer. The computer can then access the data being put on the data lines. In case of a write the buffer is set from computer to card.

There are numerous other possibilities to build interface circuitry for the ISA bus. It depends on what you want to do. There are a number of books and magazines that show you exactly how to do this.

24.3.8 Selecting an address for our card

The questions that rises now is : where can we map our board in the computer ? Well there are a number of so called 'experimenters area's where you can do this safely. But take care that nothing else is already there !. Our board is not plug-and-play so Windows cannot detect it.

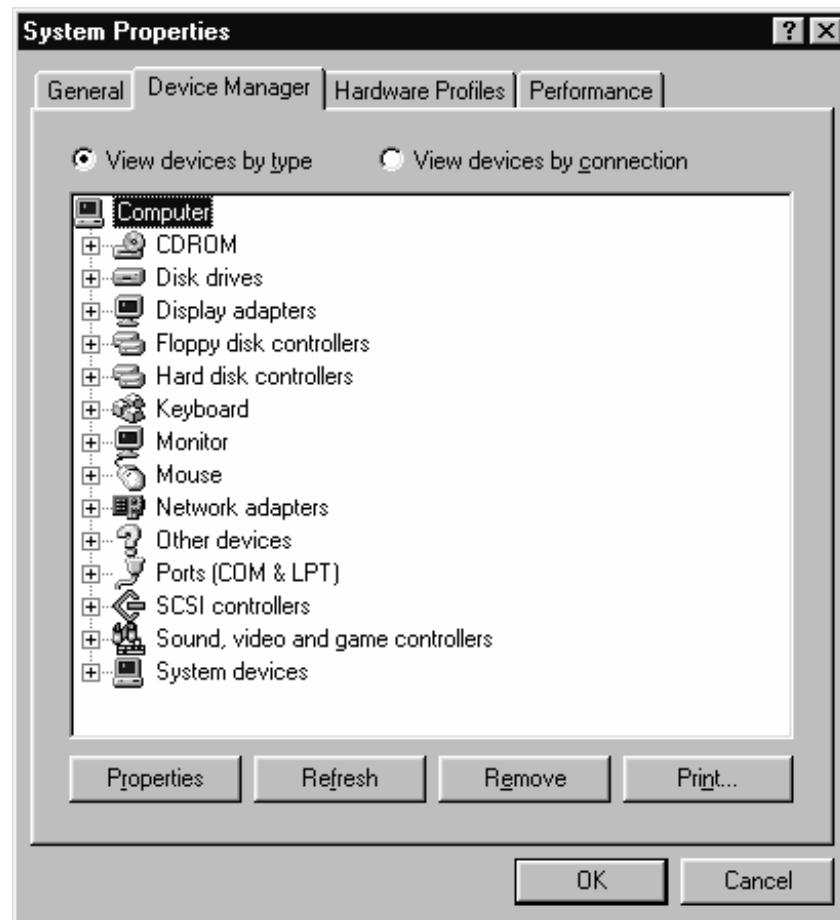
The safest area is the range 0x300 to 0x31F. It holds 32 possible addresses . This part of the IO range is clearly marked by both the IBM manuals and the official ISA spec as 'experimenters area'.

Other unused regions in the IO map are set as 'undocumented'. They can be used but it depends on the machine.

You can check the availability of addresses by using the Windows System Setting panel

Start – Settings – Control Panel and double-click on system

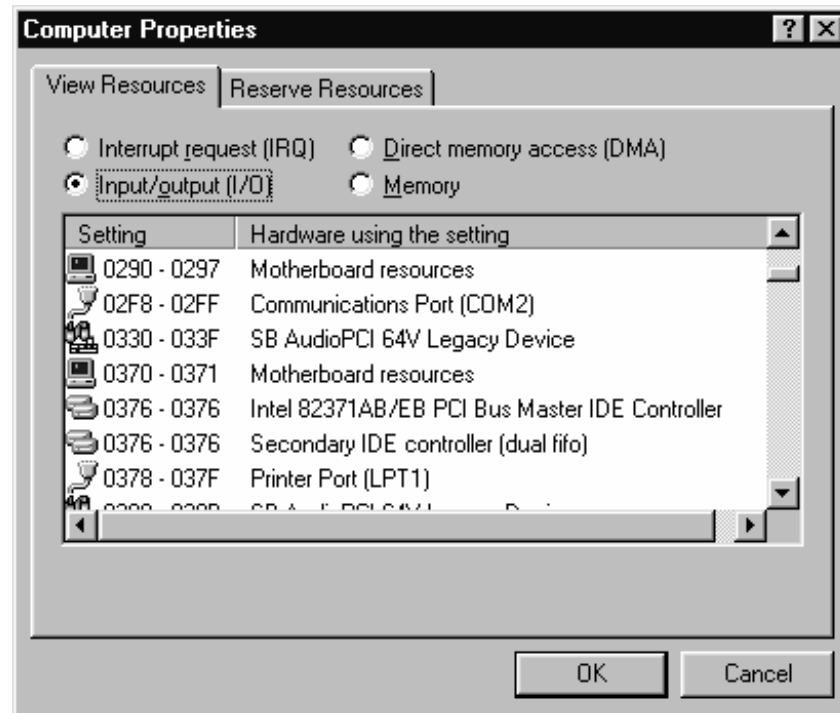
A window will open that shows the settings of your computer. Click on the tab Device manager. This shows all devices in your computer .



The devices are organized per functional group. You can of course double click on each of the devices and look for the information you want but there are easier ways to accomplish this.

To get an overview of the used interrupts , IO ranges and DMA channels you simply have to double-click on Computer.

This opens a new windows that shows the information we seek. Select one of the 4 options . In our case we want to see the Input / Output (I/O) information



The above image shows part of the resources used in a computer. The experimental range in this particular machine could be used up to the address 0x330. But don't count on it. You should check this when installing the board in a particular computer.

24.3.9 Accessing our board

Now that we have built a plug-in board and (hopefully) configured it correctly it is time to have a look at how to access it from software.

You can use the same techniques as described in the section about the printer port. After all a printer port is an IO channel that resides on the ISA bus too.

In case of the 8255 controller you can write a set of universal routines that handle the IO operations.

Example for 8255

```
Const Ioboard = &h300

Sub WritePort(portnumber,databyte)
    Select case portnumber
        case 0,1,2,3
            out ioboard+portnumber,databyte
        case else
            msgbox "Error : you attempted to access
an _                illegal port in the 8255"
    End select
End Sub

Function Readport(portnumber)
    Select case portnumber
        case 0 to 3
            Readport = inp(ioboard+portnumber)
        case else
            msgbox "Error : you attempted to
access an _        illegal port in the 8255"
    End select
End Function
```

The above 2 routines allow you direct access to any of the registers in the IO board. Note that I use a select case construction to select the validity of the portnumber. Doing this makes the code much more readable then using a construction like

if ((portnumber >0) and (portnumber <4)) then ...

Also if you need to filter out address 1 separately you can simple add a *Case* statement.

```
Select case portnumber
    case 0,2,3
        Readport = inp(ioboard+portnumber)
    case 1
        ` accessing address 1
    case else
```

```
                msgbox "Error : you attempted to  
access an _      illegal port in the 8255"  
End select
```

Chapter 25:

The GPIB bus.

In 1965, Hewlett-Packard, a major manufacturer of electronic test instruments, wanted to develop a means by which any of its instruments could communicate and exchange data with any other. The method that their engineers developed was a hardware standard and communications protocol known as the HP-IB (Hewlett-Packard Instrumentation Bus). This bus became quite popular, and in 1975 it was adopted by the IEEE as a standard—the IEEE-488 bus, also widely known as the general purpose instrumentation bus (GPIB). About 10 years later, the standard was revised to resolve a number of ambiguities that were not spelled out in the original standard. This newer version of the standard is known as IEEE-488.2.

25.1 The GPIB bus structure

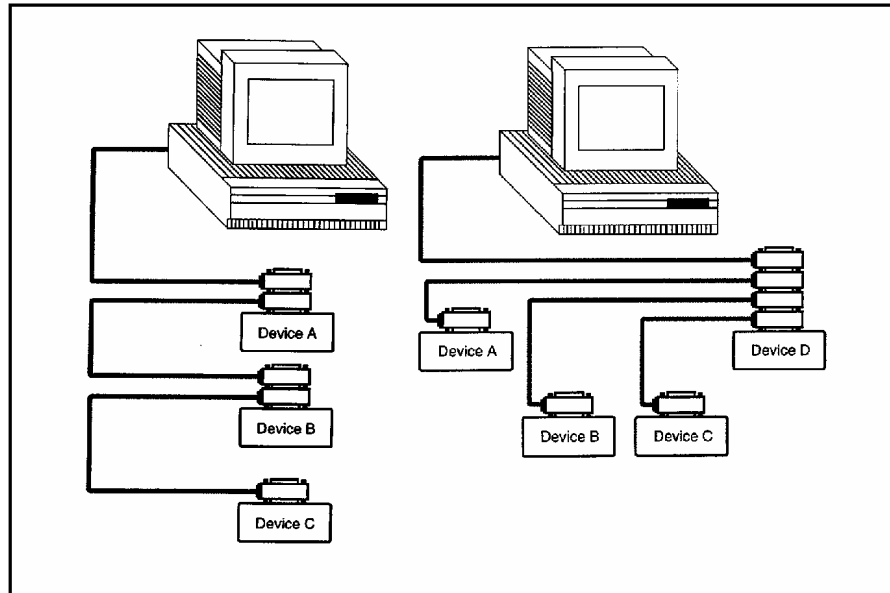
Although the GPIB interface is in fact a fairly general purpose bus, its primary use is to connect one or more GPIB-compatible instruments to a PC, and several thousand such GPIB products are available on the market. It allows data to be exchanged a byte at a time among several different devices at speeds of 100K/sec to 10MB/sec. There is also an effective handshaking protocol that synchronizes the transfers. At any given time, one device has the responsibility of being the active bus controller, which coordinates all data transfers; there is also a defined mechanism that allows another device to take over as the new controller. In practice, however, a PC with an IEEE 488 card in it is nearly always the permanent controller of the bus. The controller can assign itself and other devices to be either talkers (devices that transmit data) or listeners

(devices that receive data) with the restriction that only one device can talk at a time.

A typical example of a GPIB setup is depicted in the next picture. The PC, acting as the controller, is connected to several different devices that can be

- 1) listeners only
- 2) both talkers and listeners (at different times), or in rare cases,
- 3) talkers only.

FIGURE: An example of a typical GPIB (IEEE 488) interface bus configuration. The left part of the image shows a Daisy-chain configuration while the right part shows a star configuration.



A 16-wire cable connects the devices together, going from one device to the next to the next. There can be up to 15 different devices on a GPIB system, and the total cable length can be up to 20 meters long, provided the devices on it are

spaced no more than 4 meters apart (with a 2-meter maximum spacing strongly recommended).

The 24-pin Type 57 micro-ribbon connectors used for the GPIB have a unique stackable design, which allows one connector to be directly plugged into the back of another connector, as indicated in the figure. This 'stackability' also allows you to connect devices in a star configuration with separate cables going to each device from a single output connector on the PC's GPIB card.

To understand how a typical GPIB system works, consider an example. Suppose you want to test a frequency doubler, (a device that phase-locks to the frequency of a sine wave at its input and produces an output that's exactly twice the input frequency). What we want to know is the range of input frequencies over which the device works properly and how its operating range is affected by variations in its power supply voltage.

After initializing the system, the controller (the GPIB card in the PC) tells the waveform generator to set itself to the desired output frequency range and tells the counter to set itself to the desired input frequency range.

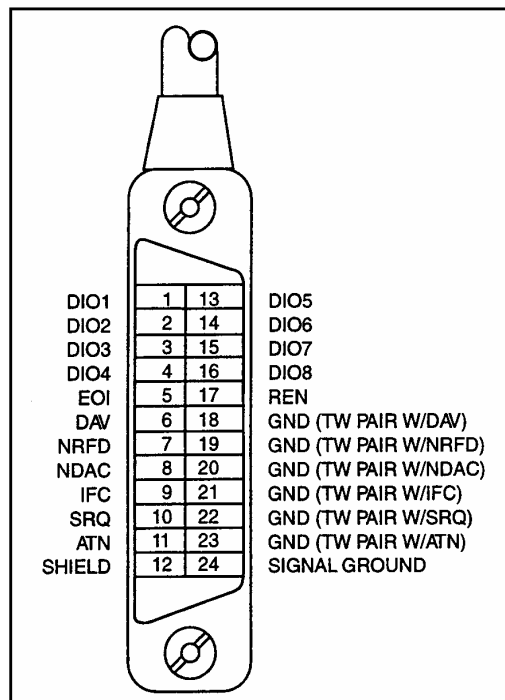
The controller then starts the test procedure by telling the power supply to output a given voltage, telling the waveform generator to output a sine wave of the desired frequency and amplitude, and then commanding the counter to read the output frequency of the device under test and send the value back to the controller. By repeating this test procedure sequence with different values of the power supply voltage and waveform generator frequency, the frequency-doubler operating range can be completely tested.

For each one of the preceding actions, the controller issues commands to make a given device (or devices) a listener so that the device can receive each message (a control setting, a frequency value, and so on) and to make another device (often the controller itself) be the talker that sends the message. This all sounds great, but you might wonder how information transfers are coordinated so that listeners having different speeds can all read messages accurately and how you program the devices to be talkers or listeners. To answer these questions, let's first look at the GPIB hardware and examine what the various data and control lines are and how they operate.

25.2 GPIB signals

The 24 pin micro-ribbon carries a number of signals that together for the GPIB bus. The GPIB has eight bi-directional data lines, DIO1-D108, three handshake

lines, and five bus management lines. The signal names, their connector pin numbers, the source that drives the signal, and a brief synopsis of their functions are given in the picture below.

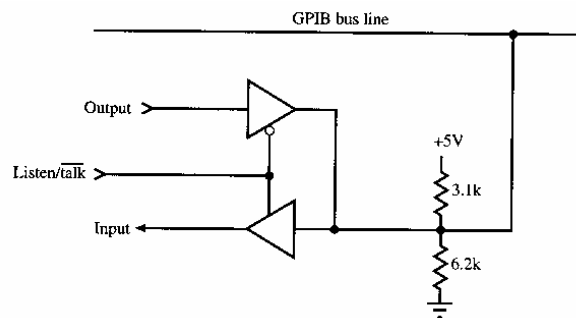


It's important to note that all the GPIB signals are active low, but the IEEE-488 standard doesn't use over-scores on the signal names to indicate this. We'll follow that convention in the section; just keep in mind that in this section, a low on a given line means the signal is asserted (i.e., true). Signals on the data lines are also inverted (1 = low).

Signal	Pin	Controlled by	Function
DIOI - D108	1-4	controller/talker	
	13-16	8-bit data/command lines	
NRFD (Not ready for data)	7	Listeners	Handshake line
NDAC (No Data Accepted)	8	Listeners	Handshake line
DAV (Data valid)	6	controller/talker	handshake line
ATN (attention)	11	controller	distinguishes commands from data
SRQ (Service Request)	10	any device	used to request service from the controller
IFC (Interface clear)	9	controller	initializes the GPIB bus
REN (remote enable)	17	controller	places device in remote mode
EOI (End or Identify)	5	controller/talker	marks end of message or (with ATN) signals parallel poll by controller

In addition to the 16 signal wires, a GPIB cable contains several ground wires. The three handshake signals and the bus management signals ATN, SRQ, and IFC have their own separate ground wires that run through the cable along with the signals as twisted pairs of wires. This helps to reduce cross-talk and noise pickup on these key control lines. There is also a signal ground wire and a shield ground wire. The latter is a ground for the cable shielding, which is a sheath of fine wire braid that surrounds the entire length of the cable and shields the wires inside from external noise sources.

A driver/receiver circuit like that shown in the figure below drives each of the signal lines.

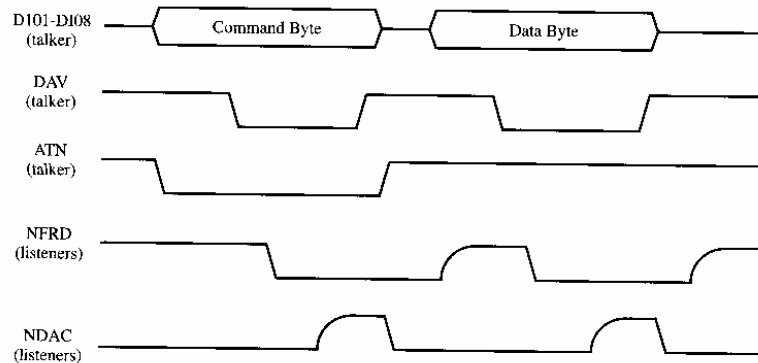


As shown in the figure above, the connection for every signal line on each device must contain its own Thevenin terminator to prevent reflections on the bus. This distributed line termination scheme is the reason why devices should be spaced no more than 2 meters apart on the bus. Open collector drivers can be used on all signal lines. The drawback is that the time required for a logic signal to make a low to high transition can be significant because the cable wiring behaves like a capacitor that must be charged up through the termination resistor to go from low to high. Tri-state drivers provide better performance and can be used on all lines except NRFD, NDAC, and SRQ, all of which need the wired-OR capability of open-collector drivers. The terminator resistors also ensure that any un-driven line is pulled up to a logic high state.

Two types of information can be transferred over the GPIB bus: commands and data. When both the controller drives Attention (ATN) and Data Valid (DAV) low, the byte value on DIOL-D108 represents a command to one or more devices. Such commands enable remote operation of devices on the GPIB and assign them to be talkers or listeners, and so on. When ATN is high and DAV is

low, the byte value on the data bus is data. Hence, ATN is a switch that identifies whether the data bus value is a command or data; DAV low means what its name says-the data on DIO1-DIO8 is valid for all listeners to read.

All bus transfers-both controller commands and talker-listener data transfers-take place using Hewlett-Packard's patented three-wire handshake. All devices must handshake when commands (ATN low) are being sent, but only listener's handshake for data transmission. This allows high-speed transmission between two fast devices even when there are much slower devices present on the same bus. The figure below presents a timing diagram for a command followed by a 1-byte data transfer, showing how the handshake signals work.



To better understand how the handshake operates, here is the sequence of events that occurs when the controller sends a command to the devices on the bus:

1. Before putting a new command on the bus, the controller checks to see if the Not Ready For Data (NRFD) line is high. Any device that's not ready to accept another data byte holds NRFD low. Thus, thanks to the open-collector connection of the NRFD line, it can go high only when all devices are ready to accept a new command byte.
2. The controller sets ATN low to indicate a command is being sent, places the command code on the data lines, and, after a delay to allow the DIO lines to settle, pulls DAV low to indicate that a valid command is present on DIO1-DIO8.
3. When each device sees DAV go low, the device pulls its NRFD line low to indicate that it knows a new byte is present, but that it hasn't yet received and stored it.

4. Once each device has stored the command byte, it releases the No Data Accepted (NDAC) line to indicate that it's accepted the byte. When the slowest device has released NDAC, this open-collector line will finally go high.
5. The controller now knows that all devices have accepted the command, so it sets DAV high and removes the command byte from the data lines.
6. On seeing DAV go high, each device sets its NDAC line low again so that it's in the proper state for the next data transfer.
7. When it's handled the command just received and is ready to receive the next byte, each device releases its NRFD line. As a result, NRFD will finally go high when the slowest device is ready.

The same sequence of events occurs when a talker sends data to one or more listeners. The only difference is that for a data transfer the ATN line is high, and only those devices that are currently configured to be listeners participate in the handshake. Non-listeners do not drive the NRFD and NDAC lines.

You may be wondering why such a complex handshake is used here. The answer is that in nearly all other bus systems such as ISA or SCSI, data is sent from a single source to a single receiver. In a GPIB system, however, there can be more than one listener and the three-wire system prevents multiple acceptance of data by a fast listener while a slow one is still busy accepting the data.

When messages (either commands or data) are sent from one GPIB device to another, the programmer doesn't need to worry about the details of the handshake. GPIB interfaces in instruments and GPIB controller cards for PCs use sophisticated ASIC's (application specific integrated circuits) that constitute a complete (or nearly so) GPIB interface on a chip. A good example of such a chip is the National Instruments TNT4882. This 100-pin chip contains both a complete ISA bus interface and a complete GPIB interface. You only need to provide a 40-MHz clock signal to create a fully functional GPIB controller. The chip appears to the PC as a set of read/write registers accessible as 16 consecutive I/O ports. Writing appropriate values into the registers allows you to control the GPIB bus any way you want.

Normally you don't program the interface chip's registers directly either. When you buy a GPIB card for a PC, it comes with software drivers that provide a set of high-level functions you can call to communicate with and control the instruments on the GPIB. Since these functions differ from one board

manufacturer to another, they won't be discussed here. Instead, we'll continue to look at the actual signals present on the bus.

25.3 Controlling a device on GPIB

Let's next see how each device in a GPIB system can be initialized by the controller and then commanded to send or receive data. To permit individual communications with the devices on a GPIB system, each device must have a 5-bit GPIB address. This address is normally set by means of switches on the back of a GPIB instrument or by using controls on its front panel. Any address between 0 and 11110 (address 30) may be used, except that by convention, address 0 is usually reserved for the controller's address. Address 31 is used for the un-talk and un-listen commands (see next paragraph) and thus can't be used as a device address. Note that even though there are 31 possible primary addresses, a maximum of 15 devices (including the controller) can be attached to the GPIB at a given time.

The 5-bit address set in the device's switches actually specifies two different address codes that are used to assign devices to be talkers or listeners. To command a device to be a listener on the bus, the controller sends a command (ATN low) with the device's My Listen Address (MLA) on DIO1-D108. The MLA is obtained by adding 20h to the 5-bit GPIB address. Similarly, a device is commanded to be a talker by sending a command with the device's My Talk Address (MTA) on DIO1-D108.

The MTA is obtained by adding 40h to the 5-bit address. For example, if the device's GPIB address is 3, then sending the command 23h (or the ASCII character '#') makes it a listener; sending the command 43h (or the ASCII character 'C') makes it a talker. The un-listen (UNL) command 3Fh (ASCII character '?') causes all current listeners to stop being listeners; the un-talk (UNT) command 5th (ASCII character '"') causes the current talker to stop being a talker.

It's possible, but not very common, for devices to be extended talkers and/or listeners. Such devices have secondary addresses that provide selective access to sub-units within the device. The secondary address is sent as a second address byte after the initial one, with 60h added to the physical secondary address to distinguish it as a secondary address.

25.3.1 Initializing a GPIB system

To initialize a GPIB system, the first thing the controller typically does is to send the Interface Clear (IFC) command by asserting the IFC line for a few hundred microseconds on the GPIB bus. All devices on the bus must monitor this line and, upon seeing it go low, must immediately cease all bus activity and go into an idle state. The controller will then usually assert the REN line. As long as REN remains low, listeners must switch into and stay in remote mode. In this mode, devices disable their front panel controls and allow the controller to send them commands that change any characteristics that were previously determined by knobs and switches on the front panel of the device. The programmed settings will override existing front panel control settings.

Note that the interface clear and remote enable operations are usually referred to as the transmission of Interface Clear (IFC) and Remote Enable (REN) commands. However, IFC and REN are unlike other commands (they're called uni-line commands) in that the DIO1-DIO8 lines aren't used and there is no handshaking. We're just talking about asserting a single control line. In most cases, the controller next sends a device clear (DCL) command to each device. The DCL command places the settings of all listener's controls into a default state determined by the manufacturer of each device. Like UNT and UNL, DCL is one of about a dozen commands that are given by placing a value on the DIO lines (14h for DCL) and asserting ATN. Several more of these commands are discussed shortly in conjunction with device polling.

25.3.2 Exchanging data

The programming of specific device controls to their desired values is done by sending device dependent messages to the device using the specific commands that instrument understands. The format of these commands is up to the manufacturer and can vary greatly from one device to another. They are typically ASCII strings such as "FL1Z0R3" (FL1 = filtering on, Z0 = auto-zero off, R3 = 1-volt range) or ":CAL:USER;RANGE 3" (CAL:USER = perform user calibration procedure, RANGE 3 = set gain range #3). These messages are preceded by a sequence of commands to select the talker and listener. For example, if the controller is at GPIB address 0 and a voltmeter to be programmed is at address 03, you can set its voltage range to 1 volt by having the controller send the commands to Un-listen, Un-talk, set My Listen Address = 03, set My Talk Address = 0, and finally send the ASCII characters "R3". All of this is done by having the controller send the string of commands 3f Sf23 55

(or equivalently the ASCII string "?# U") followed by the device dependent message "R3". It's useful to understand that a sequence of commands can be specified as an ASCII string since many GPIB systems are programmed using an extended form of BASIC; in BASIC, sending an ASCII string is often the easiest and most compact way of sending GPIB commands or messages.

25.3.2 EOI assertion

When a multi byte message such as "FL1Z0R3" is sent from a talker to a listener, the listener needs to have some way to know when the message is complete and the last byte has been sent. The end of message signal is given by requiring the talker to assert the End Or Identify (EOI) line as the last byte of the message is placed on the data lines. The reason for the EOI line's strange name is that it's a dual purpose line. In addition to signaling the end of a message, it's also used to initiate a parallel poll (see below). Another technique used by some GPIB instruments is to indicate the end of message by sending an ASCII line feed character (= Oah) as the last byte.

25.4 IEEE488.2

The original IEEE-488 standard left the codes, formats, and protocols for device-dependent messages and other kinds of information completely up to the device manufacturers. Furthermore, the GPIB capabilities of instruments from different manufacturers varied greatly. Not surprisingly, the result was that software drivers written for one instrument were incompatible with the drivers for any other instrument, even if the instruments were quite similar. When you bought a GPIB interface card, you needed to obtain software drivers that were written specifically for every instrument you were going to put in the system. To help eliminate these incompatibility problems, the IEEE-488.2 revision of the original standard specified a set of data formats, message protocols, and common commands that every IEEE-488.2 compliant device must adhere to.

In general, messages and information sent over the GPIB are in the form of ASCII strings, with well-defined formats for numbers. For example, the number 20 could be sent as any of the ASCII strings "20", "20.0", or "2.0e+1". There is also provision for sending blocks of binary data using the format #nn.<binary data> where nn is a two-digit ASCII number giving the number of bytes of binary data that follows. IEEE-488.2 also defines a set of standard protocols for instrument status reporting.

25.4.1 Common Command Set

A common command set of 39 standard commands is defined by IEEE-488.2. All compliant devices must implement 13 of these, with 5 more being required if certain instrument capabilities are present. The other common commands are optional. Each of the commands is sent over the bus as an ASCII string with ATN false, just like device-dependent messages are, and they have the form *name, where name is a three-letter mnemonic for the common command. The leading asterisk identifies the command as being a member of the common command set. Two examples of such commands are '*CLS' which clears an instrument's status registers, and '*IDN?', which causes the device to provide its device identification string. The second example here is 1 of the common command queries. The query mnemonics all end with question mark and cause the device to transmit some requested piece of information the next time it's made into a talker.

Command	Description
*CLS	Clear status command : Basically resets any errors
*ESE	Standard Events Status Enable
*ESE?	Standard Events Status Enable Query
*ESR?	Standard Events Status register Query
*IDN?	Identification Query
*OPC	Operation Complete Command
*OPC?	Operation Complete Query
*RST	Reset command
*SRE	Service request enable command
*SRE?	Service request enable query
*STB?	Read status byte query
*TST?	Self test query

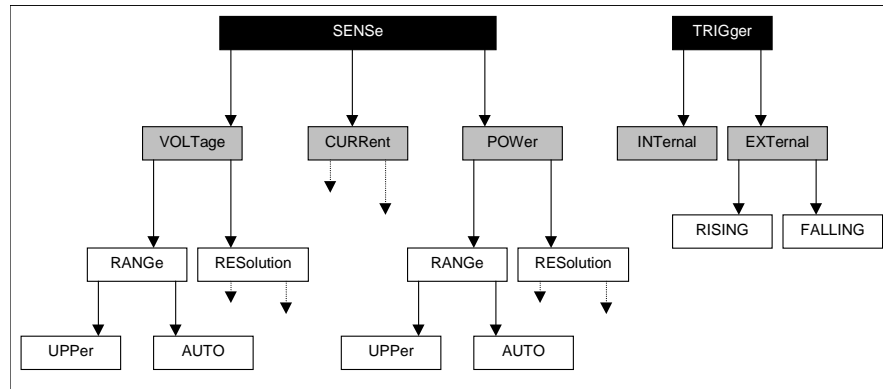
*WAI?	Wait to continue command
-------	--------------------------

25.5 SCPI

At about the same time IEEE488.2 was introduced a formal proposal was made to further structure the software level of the GPIB interface. The idea was to define a uniform command dictionary that could be understood by all instruments implementing similar functions. This proposal was formalized into the Standard Commands for Programmable Instruments (SCPI) document. Today, almost every new instrument being developed follows this specification.

SCPI organizes and orders the commands an instrument understands into functional groups. A rich 'keyword' set has been developed that allows you to design an instruction set for any kind of instrument. However due to the complexity of modern instruments, the SCPI standard is evolving as well.

SCPI commands obey to a strict hierarchical structure. The SCPI command set is to be seen as a tree originating at the root command and dispersing into different branches depending on the function required. Another feature of the SCPI specification is that parts of the command keyword can be omitted. Everything that is specified in UPPER case is required. Everything written in lowercase may be omitted. This allows for speed optimization over the GPIB bus without really compromising readability.



Each command is preceded by a colon (:). In the above case the following would be valid commands:

```

:SENSe:VOLTage:RANGe:AUTO<EOI>
:SENS:VOLT:RANG:AUTO<EOI>
:SENS:POWer:RANG:UPPer<EOI>
:TRIG:EXTernal:FALLING<EOI>

```

The following would be an illegal command:

```

:SENS:POWer:EXTernal:FALLING<EOI>

```

The command tree does not allow you to descend along this particular path.

The colon preceding the SCPI command forces the instrument to start at the root level. If you don't send a colon before a command you remain at the same level as the last issued statement in the command.

```

:SENS:VOLT:RANG:AUTO<EOI>
UPPER<EOI>
AUTO<EOI>

```

The first command takes the instrument down to the range configuration for sensing volts and sets it to AUTO. The second command only sends the UPPER command. Since there is no preceding colon, the instrument stays at its current level in the branch of the command tree. Thus actually it will execute the command

```
:SENS:VOLT:RANG:UPPER<EOI>
```

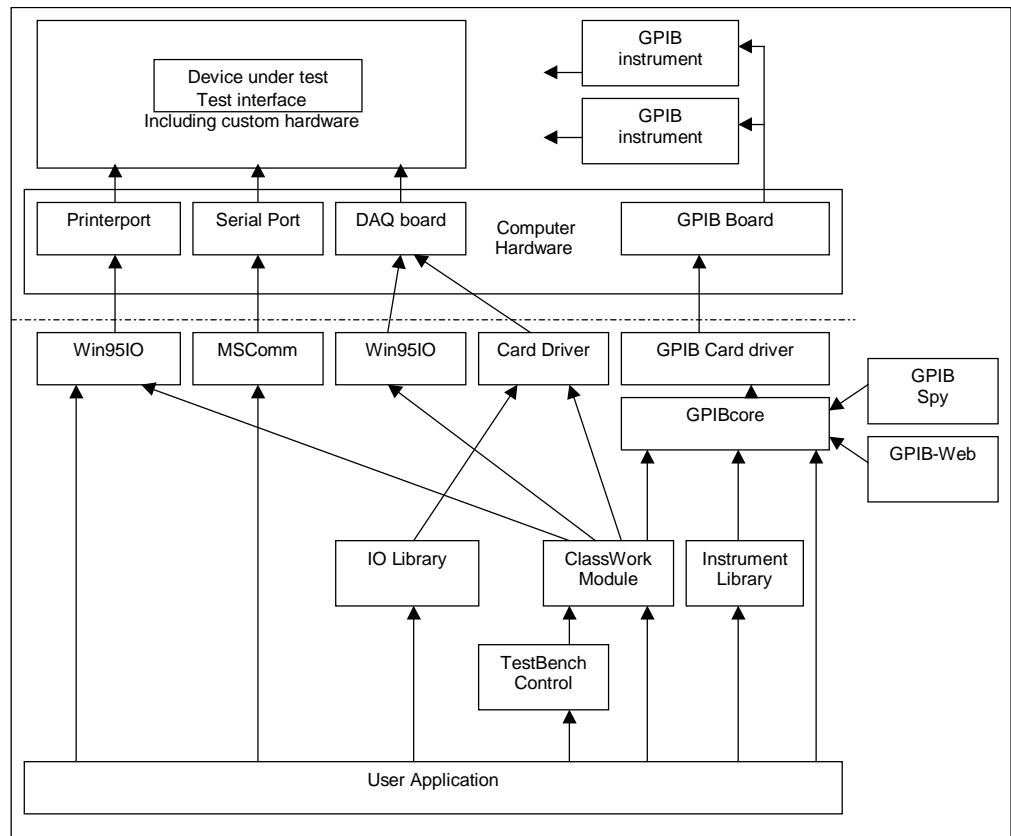
Due to this flexibility the amount of data that needs to be sent over GPIB can be reduced to a minimum if you apply clever code optimization.

Chapter 26:

Vision

This chapter will present a collection of routines , objects , controls and classes that together form the Visual Instrumentation Solution or Vision for short. (You could also read it as Vincent's Instrumentation Solution). The image below shows you how all parts of the Vision system fit together.





As you can see the number of options at your disposal is rather high. All the low level hardware interfacing is done using tools already described before : MSComm for Serial port IO and Win95io for Hardware access. Besides these two already known modules a number of other modules exist.

- GPIB card driver : this driver is supplied by the board maker. It offers an access path to the GPIB hardware.
- Card Driver : A driver supplied by the board vendor of the IO or DAQ board.
- 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 well monitor, 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 (Printerport , 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.

26.1 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. But 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. A lot of 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. The GPIBcore currently exists in 2 versions, and more will follow.

- GPIBcore 4.5: Latest incarnation of the 16 bit GPIBcore.
- GPIBcore 5.0: First 32 bit NI compatible release.

Both releases support ALL National Instruments GPIB boards. Future releases will handle other boards. The GPIBcore does not require nor interfere with other software layers such as VISA, SICL or IVI.

Note : All version of GPIBcore rely on drivers supplied by the board maker. These drivers might or might not be compatible with things like VISA, SICL or IVI. The 16 Bit version requires DOS level board drivers to be installed that are mapped to GPIB0. These card-drivers are supplied from the board manufacturer and might or might not be compatible with Windows drivers from the same or other manufacturers. To avoid porting problems : use Vision 5.0.

26.1.1 GPIBcore features

The GPIBcore uses the native Windows based card drivers. This means that GPIBcore is compatible with other tools using GPIB as well. The GPIBcore behaves according to the driver specification for your board. This means that, if your driver is compliant with Windows NT that any program written with GPIBcore will work on Win-NT as well.

GPIBcore uses the standard debug console of Visual Basic: The Immediate Window. Once you compile your program to an executable all commands accessing this debug console are automatically clipped. This resulting in a faster program.

A separate debugging window is available. When you want this debugger you simply add the form to your program and call the function VisionStart.

26.1.2 Installing GPIBcore

Simply copy the entire directory to your hard disk. The preferred location is a directory on the C disk called Vision or VisualGpib. There is no setup program to be run since all modules are supplied as source code and as such need no special installer.

26.2 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.

26.2.1 GPIB functions

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

26.2.2 GPIBinit

Syntax :

GPIBinit

Description:

This function initializes the GPIB stream. It sets up communication with the card vendor specific driver. Furthermore it resets the GPIB subsystem of the computer , and unlocks all attached devices on the bus. You can close the bus by executing a GPIBbye command. No GPIB commands are executed before the call of this function. Therefore you should call this during the startup of your program. Attempting to access the GPIB before this command has been executed will show warning messages on the immediate windows of Visual Basic. The program will neither stop nor crash.

26.2.3 GPIBbye

Syntax :

GPIBbye

Bye

Description:

By issuing this command you terminate the GPIB operations. The GPIB stream is released in an orderly manner. All devices are brought back to local state. Once the GPIB stream is closed all calls to GPIB functions other then GPIBinit will simple not be executed. (No error will be generated. They simply are ignored)

26.2.4 GPIBopen

Syntax :

GPIBopen <address>,[descriptor]

Example:

GPIBopen 5
GPIBopen 5,"Multimeter"

Description:

This command opens a channel to the device at the specified address. This function places the machine in remote mode and initializes it as a Listener. You can specify your own name for the instrument in the optional descriptor variable. If you do not supply a name then GPIBopen will interrogate the machine to find out its exact description. (not all machines support this, in particular older machines.).

When a device has been successfully opened a brief report is generated on the immediate window of Visual Basic. The countering command is GPIBclose.

26.2.5 GPIBclose

Syntax :

GPIBclose <address>

Example:

GPIBclose 5

Description:

This command will free a device from the GPIB bus. It places the machine back to Local mode. It is good practice to close all machines before exiting the

program. However, starting with release 2.5 of GPIBCore all slaves are automatically closed upon program termination.

26.2.6 GPIBtimeout

Syntax :

GPIBtimeout <time in seconds>,[address]

Example:

GPIBtimeout 1
GPIBtimeout 30,2

Description:

This sets the timeout value for bus communications. IF a machine does not respond within the selected timeframe an error is generated. When setting a timeout value without specifying an address you are setting the timeout level for the board level operations. If you want to control the instrument level timeouts, then you need to specify the address of the device as well.

26.2.7 GPIBreset

Syntax :

GPIBreset

Description:

This command resets the GPIB board in your computer and attempts to free a stuck bus. THIS CLOSES ALL DEVICES. This is different from the previous versions.

26.2.8 GPIBdefer

Syntax :

GPIBdefer <state as Boolean>,[address]

Example:

```
GPIBdefer True
GPIBdefer false,22
```

Description:

This command defers the transport of GPIB calls. Any GPIB operations will be emulated when GPIBdefer is set to true.. This means that the command is not sent to the bus but simply denied. However they do show up in the console. This allows you to check syntax of the commands you are sending. You can also use this to write and debug code on a machine which does not contain GPIB card.

A special case is the GPIBread function used when Deferring is switched on. The ib\$ will contain the string "-VOID-". The value contained in ibret variable is a random number between 0 and 100. This allows you to check the functionality of your programs. If you specify an address then the DEFER state is altered for the specified address only. This is called a local defer, while a defer operation without specified address is a global defer. The local and global states are logically OR-ed together to decide whether there is access to a machine or not.

26.2.9 GPIBSinglestep

Syntax :

GPIBSinglestep <state as boolean>

Example:

```
GPIBSinglestep True
```

Description:

This functions allows you to switch to single step mode for all GPIB operations. You can also change the mode using the console menus. This feature is useful for tracking GPIB timing problems. Whenever a GPIBcommand has been executed a messagebox pops up to prompt you for further action. You can decide to take the next step , stop the program , or abort stepping and continue at normal speed.

26.2.10 GPIBtroff

Syntax :

GPIBtroff [address]

Example:

GPIBtroff
GPIBtroff 5

Description:

This command turns of tracing for all or one address at a time. Using this you can eliminate the commands for the machines that you don't want to trace. By default all addresses are traced.

26.2.11 GPIBtron

Syntax :

GPIBtron [address]

Example:

GPIBtron
GPIBtron 5

Description:

This command is the opposite for the GPIBtroff command: You can enable all or only selected addresses to be traced.

26.2.12 GPIBwrite

Syntax :

GPIBwrite <address>, <command as string>

Example:

```
GPIBwrite 5,"*RST"  
GPIBwrite 10,":FUNCTION:SINE"  
GPIBwrite 5,"H2"
```

Description:

This command transports commands to the machine at the designated address. The command should be formatted as a string. If you want to send numbers to the machine you should convert them using the str\$ or sStr\$ function

```
GPIBwrite 5,"RANGE "+sstr$(x)
```

26.2.13 GPIBread

Syntax :

GPIBread address,[command]

Example:

```
GPIBread 5  
GPIBread 5,"RESULT?"
```

Description:

This command allows you to read data from the GPIB stream. Data is returned in 2 global variables. Ibret contains the numerical value of the returned information. IB\$ contains the whole unformatted string of data returned by the machines. Optionally you can specify a command .this command is sent to the target before the read is attempted. Typically you need to send some command to the device before it returns data. By specifying this command in the GPIBread function you avoid having to issue a GPIBwrite first. It saves some lines of code in your program

26.2.14 GPIBfind

Syntax :

X = GPIBfind <address>

Example:

X = GPIBfind(5)

Description:

This command checks the presence of a device at the specified address. If a device is found the functions returns TRUE else it returns FALSE

26.2.15 Other GPIB functions

- GPIBthunkwrite
- GPIBthunkread
- GPIBparse
- GPIBrespawn
- GPIBterminator

These functions have been removed in version 5.0 an upward. These functions were required to control older instruments that did not behave 100% according

the GPIB specification. The Core knows how to handle these internally now. The Parse command has been removed entirely. The thunked access (GPIBthunkread and GPIBthunkwrite) is automatically redirected to normal Read and write operations. Accessing removed functions will result in a message box appearing with the notification what function call was attempted. This allows you to patch your code more easily.

26.3 GPIBcore I/O functions

The GPIBcore also provides means to interact with the PC's hardware on a low level basis. Visual basic provides access to disk , Com ports and printers in a standard way. However if you want to use or 'abuse' these ports in a non-standard way you will need a means of accessing the hardware registers. This functionality can be found in WIN95io.DLL.

This release of the core fully embeds this library into the GPIBcore.

26.3.1 OUT

Syntax :

OUT address, data

Example:

Out &h378,88

Description:

The out command allows you to write a data-byte to a specified IO address in the PC's IO space. This works in exactly the same way as in regular DOS based basic languages.

NOTE be very careful where you are writing, after all you could very quickly find yourself faced with the 'blue screen of death' if you are writing to certain addresses.

26.3.2 OUTW

Syntax :

OUTW address, data

Example:

OUTW &h378,&hffff

Description:

This command is the WORD version of the OUT command. Observe here that you can only write to EVEN addresses , and that you must have a card that allows this to do. Typically you will need an AT card which has a 16 bit Data bus.

NOTE: Be very careful where you are writing , the Blue screen is already lurking behind the corner.(see also the OUT command).

26.3.3 INP

Syntax :

X = INP (address)

Example:

Result = inp(&h379)

Description:

The INP command allows you to read a data-byte from a specified IO address. This works in exactly the same way as in regular DOS based basic languages.

NOTE Contrary to OUT and OUTW , you can read any location in the IO map without any problem.

26.3.4 INPW

Syntax :

X = INPW (address)

Example:

Result = inp(&h379)

Description:

This command is the WORD version of the INP command. Observe here that you can only read from EVEN addresses , and that you must have a card that allows this to do. Typically you will need an AT card which has a 16 bit Data bus.

NOTE: Same story as with INP : No problem reading anywhere.

26.4 GPIBcore Miscellaneous support functions

Visual Basic does a lousy job when it comes to processing binary data. There are no dedicated functions for this. Of course you have the basic logical operators , but hey , you can't expect us to write logical routines every time we need something simple.

GPIBcore now has built in functions that allow you to perform common tasks in binary data manipulation.

26.4.1 setBIT

Syntax :

Setbit variable,bit

Example:

X=5

Setbit x,3 ' x becomes 13

Description:

This function will set the specified bit in any integer variable. If the variable cannot hold the resulting number it will automatically be sized so it can hold the data. Bits range typically from 0 to 31 (32bit numbers).

26.4.2 *clearBIT*

Syntax :

`clearBIT variable,bit`

Example:

`X=5`

`clearBIT x,2 ' x becomes 1`

Description:

This function will RESET the specified bit in any integer variable. Bits range typically from 0 to 31 (32bit numbers).

26.4.3 *flipBIT*

Syntax :

`flipBIT variable,bit`

Example:

`X=5`

`flipBIT x,1 ' x becomes 7`

Description:

This function will INVERT the specified bit in any integer variable. This allows you to easily flip or toggle a specified bit in a variable. Bits range typically from 0 to 31 (32bit numbers).

26.4.4 swapBIT

Syntax :

swapBIT variable, bit1, bit2

Example:

X=5

swapBIT x, 2, 1 ' x becomes 3

Description:

This function will swap two bits from place. This allows you to change to bit order of an integer. If the variable cannot hold the resulting number it will automatically be sized so it can hold the data. Bits range typically from 0 to 31 (32bit numbers).

26.4.5 BITset

Syntax :

X = BITset (variable, bit)

Example:

X=5

X = BITset (x,2) ' returns TRUE

Description:

This function will check if a specified bit is set to 1 in a variable. Bits range typically from 0 to 31 (32bit numbers).Result is TRUE or FALSE.

26.4.6 BITclear

Syntax :

X = BITclear (variable, bit)

Example:

X=5

X = BITclear (x,2) ' returns FALSE

Description:

This function will check if a specified bit is RESET to 0 in a variable. Bits range typically from 0 to 31 (32bit numbers).Result is TRUE or FALSE.

26.4.7 swapNIBBLE

Syntax :

swapNIBBLE variable

Example:

X=&h5F

SwapNIBBLE X ' X is now &hF5

Description:

This function will swap the NIBBLE order in a BYTE. If you pass larger numbers they will be truncated to a byte before the operation takes place..

26.4.8 IoNIBBLE

Syntax :

LoNIBBLE variable, nibble

X = LoNIBBLE (variable)

Example:

X=&hf5

Y = LoNIBBLE (x) ' y = 5

LoNIBBLE x, 7 ' x is no &hF7

Description:

This is a DUO function. You can either op to retrieve the LOW nibble , or change the LOW nibble of a BYTE. By calling it like a function you retrieve the nibble. By calling it like a SUB you set the nibble to the specified value.

26.4.9 hiNIBBLE

Syntax :

hiNIBBLE variable, nibble

X = hiNIBBLE (variable)

Example:

X=&hf5

Y = hiNIBBLE (x) ' y = &hF

hiNIBBLE x, 7 ' x is no &h75

Description:

This is a DUO function. You can either op to retrieve the HIGH nibble , or change the HIGH nibble of a BYTE. By calling it like a function you retrieve the nibble. By calling it like a SUB you set the nibble to the specified value.

26.4.10 SwapBYTE

Syntax :

SwapBYTE variable

Example:

X=&h5FF5

SwapBYTE X ' X is now &hF55F

Description:

This function will swap the BYTE order in a WORD. If you pass larger numbers they will be truncated to a WORD before the operation takes place.

26.4.11 loBYTE

Syntax :

LoBYTE variable, byte

X = loBYTE (variable)

Example:

X=&hf55F

Y = loBYTE (x) ' y = &h5F

LoBYTE x, 7 ' x is no &hF507

Description:

This is a DUO function. You can either op to retrieve the LOW byte , or change the LOW byte of a WORD. By calling it like a function you retrieve the BYTE. By calling it like a SUB you set the BYTE to the specified value.

26.4.12 hiBYTE

Syntax :

hiBYTE variable, byte

X = hiBYTE (variable)

Example:

X=&hf55F

Y = hiBYTE (x) ' y = &hF5

hiBYTE x, &h70 ' x is no &h705F

Description:

This is a DUO function. You can either op to retrieve the HIGH byte , or change the HIGH byte of a WORD. By calling it like a function you retrieve the byte. By calling it like a SUB you set the byte to the specified value.

26.4.13 Delay

Syntax :

Delay seconds

Example:

Delay 5

Description:

This command relies on the internal system timer to provide for accurate timing sequences. It stops program execution until the specified number of seconds has elapsed.

26.4.14 Microdelay

Syntax :

Microdelay milliseconds

Example:

Microdelay 200

Description:

Same story as with the delay command except this one time milliseconds.

26.4.15 SStr\$

Syntax :

```
string = sstr$ ( value expression )
```

Example:

```
A$ = sstr$ ( 5 )
```

Description:

This command is similar to the str\$ function already present in Visual basic. Except that this flavor strips off the whitespace at the beginning and end of the returned string. Useful if you have a lot of

```
'x$ = trim$ ( str$ ( something )) style stuff in your code..
```

26.4.16 Bin\$

Syntax :

```
String = bin$ ( value expression )
```

Example:

```
X$ = bin$ ( &h55 )
```

```
X$ = bin$ ( 99 )
```

Description:

This command will return a string containing 0 and 1 that represents the binary notation of an integer number. The command can handle negative numbers. They are returned in standard 2 complements notation.

26.4.17 vVal

Syntax :

```
String = bin$ ( value expression )
```

Example:

```
x = vVal ( &h55 ) ' x = 55h
x = vVal( &b10011001 ) ' x = 99h
```

This routine converts a string to a number. It operates in a similar manner as the Val command from Visual Basic , except that it can handle binary numbers as well.

26.4.18 Logentry

Syntax :

```
Logentry string
```

Example:

```
Logentry "Hello world"
```

Description:

This is simply a command that allows you to write information to the console. This can be useful for debugging purposes.

26.5 Instrument and IO libraries

The GPIBcore provides the access channel to the GPIB board. But you can't expect us to look up the instrument functions in the users manual every time. So a set of modules (.BAS files) has been created that provides an interface to the most common instruments.

The construction of these modules is pretty straightforward. It's simply a collection of routines that allow you to control a particular instrument.

Example :

```
Option Explicit
' -----
'   Hewlett Packard 34401
'   Digital Multimeter
```

```
' -----  
' Winvision Instrument library  
' Initial release: 27/10/98  
' Written by : Vincent Himpe  
'  
' Modification history :  
' 30 / 08 / 1999 : Verified for Winvision  
Release IV V.Himpe  
' -----  
  
Public Sub HP34401VoltsDc(address)  
    GPIBwrite address, ":CONF:VOLT:DC"  
End Sub  
  
Public Sub HP34401VoltsAc(address)  
    GPIBwrite address, ":CONF:VOLT:AC"  
End Sub  
  
Public Sub HP34401CurrentDc(address)  
    GPIBwrite address, ":CONF:CURRE:DC"  
End Sub  
  
Public Sub HP34401CurrentAc(address)  
    GPIBwrite address, ":CONF:CURRE:AC"  
End Sub  
  
Public Sub HP34401ohms4(address)  
    GPIBwrite address, ":CONF:FRES"  
End Sub  
  
Public Sub HP34401Frequency(address)  
    GPIBwrite address, ":CONF:FREQ"  
End Sub  
  
Public Sub HP34401Period(address)  
    GPIBwrite address, ":CONF:PER"  
End Sub  
  
Public Sub HP34401ohms2(address)  
    GPIBwrite address, ":CONF:RES"  
End Sub  
  
Public Sub HP34401Trigger(address, mode)  
    Select Case mode  
        Case 1      ' external
```

```
        GPIBwrite address, "TRIG:SOUR:IMM"  
    Case Else ' internal  
        GPIBwrite address, "TRIG:SOUR:EXT"  
    End Select  
End Sub  
  
Public Function hp34401measure(address)  
    GPIBwrite address, "INIT"  
    GPIBread address, "FETCH?"  
    hp34401measure = ibret  
End Function
```

When writing this kind of modules it is good practice to follow the strictest possible syntax. So please force yourself to declare every variable. Simply specify on top the *Option Explicit* modifier. If you forget a declaration the Visual Basic IDE will prompt you.

By constructing these simple well defined modules you can easily build a very modular program. Besides that you can use the modules you or someone else made over and over again.

26.6 ClassWork

ClassWork is a style of instrument drivers developed for use in conjunction with VISION. It builds on the concept of classes in Visual Basic to provide a uniform and easy access to instrument and hardware functions. Any existing instrument interfaces whether plug-in or GPIB based can be implemented as a ClassWork module.

26.6.1 The ClassWork concept

A ClassWork module is the basic piece of code, consisting of procedures, functions and variables, that together form the interface to an instrument. This module is implemented as a class. By adhering to the concept set forth in this manual you will easily construct your own modules and use existing modules.

The whole concept is constructed to provide a uniform and easy access channel to T&M (test and measurement) equipment from a programmer's point of view.

While originally conceived for Visual Basic you can imply this style on other languages as well.

The concept is such that it is taking away some of the particularities involved with each instrument. Not only the way in which you gain control over a device is standardized but also some of its basic functions. For instance all power supplies have operators that allow you to program voltage and current. For any brand and model this function has the same name. As a result of this I can immediately swap supplies with a different model and brand by simply redefining the class to which a particular instrument should belong. Confused? The example below will clear things out.

Setup 1:

Uses a machine from brand ABC model 12a.

This machine has a command to set the voltage called SV. To set the voltage to 5 volts I would be required to send 'SV5' to the machine. The machine is set to respond to GPIB address 5.

Setup 2:

Uses a supply from brand XYZ model 99z

This command to set the voltage is OUTPUT: VOLTS. And the instrument has a dual output. To set this instrument to 5 volts you need to provide it with a string containing not only the voltage but also the channel. ': OUTPUT: VOLTS: CHANNEL1: 5V'.

The machine is set to respond to GPIB address 22.

Suppose you have a program developed for case 1, and your supply malfunctions. You want to use the supply from setup 2. The following problems will arise:

- The supply has a different command set so you need to adapt you program.
- It has multiple channels requiring additional information to be sent.
- The GPIB addresses are different. So you need to fix these up as well

This is where ClassWork comes in.

26.6.2 The ClassWork solution

ClassWork defines a uniform set of commands to control these supplies. Both modules (one for each of the above described supplies) contain a function to set the output voltage. This function will format the supplied data in a style that can be processed by the instrument. By doing this you are abstracting the instruments.

ClassWork is breaking multi-channel devices into independent entities. Any device containing more than one channel is broken apart into single channel devices. Each of these single channels is controlled independently!

ClassWork still requires the address of the machine once and only once. During definition you set the address and the required channel, and from then on this item becomes a true standalone object you can use throughout your program.

A nice side effect is that, while all of the above makes migration and maintenance easier, it also provides you with easier to understand code. In the past you accessed machines using addresses.

Sure you stored them in variables or constants, which in turn needed to be global. But you still used to write things like:

```
Const supply=5
ABCvolts (supply, 5)
Or
Const supply =22
XYZvolts (supply, 1, 5)
```

While solving this, ClassWork goes a step further. It treats your instrument as an object. Just like you have buttons and textboxes, you now have access to your instruments as objects.

You now simply define your supply and assign it an address and channel

```
Dim Supply as new ABC
Supply.address =5
Supply.assignto =1
```

If tomorrow your supply from ABC breaks down and you need to use brand XYZ all you do is change a few words.

```
Dim Supply as new XYZ
Supply.address =22
Supply.assignto =1
```

ABC becomes XYZ and the 5 becomes a 22. Now, if in the unlikely event that channel 1 of this XYZ would be broken too and you were forced to use the second channel, you only change the assignto parameter and your program is running again.

```
Dim Supply as new XYZ
Supply.address =22
Supply.assignto =2
```

26.6.3 Programming using ClassWork

Writing a program using ClassWork is just like writing any other program. The only thing that differs is the way you approach instruments. Since ClassWork considers instruments as objects (a ClassWork instrument is logically an object derived from a Class) , you can reference them just like you would do with a checkbox or a textbox.

Typically you create a new project. The next thing you do is add the GPIBcore module to your project. You need this module always, since ClassWork objects also use the same handler to perform GPIB I/O. So far you have done nothing new (if you were already using the Vision system).

To load instrument libraries you now select Project - Add Class Module. The modules are located in the ClassWork directory of the Vision installation.

You still need to write your startup and shutdown code

```
Private Sub Form_Load( )
    GPIBinit
End Sub
Private Sub Quitprogram_Click( )
    Bye
End
End Sub
```

From now on things change. You need to derive instruments from the loaded classes. This is done by defining a new variable as a new <instrumentclass>.

```
Dim Voltmeter as new HP34401
Dim Supply as new HP6624
```

In your startup code you add a piece of code that sets the address and assign to parameters.

```
Private Sub Form_Load()
    GPIBinit
    Voltmeter.Address = 22
    Supply.Address = 3
    Supply.AssignTo = 3
End Sub
```

Note

For single channel instruments it is not required to use the assign to parameter. Per default this parameter is set to 1.

You will notice that the Visual Basic environment will show you a list with possibilities you can select, just like you were using any other control. That is exactly what is happening. Your instrument has been turned into an object.

26.6.4 A Sample ClassWork program

The program below defines 3 instruments and performs a voltage sweep while plotting the voltage and current through a network.

```
` REM ClassWork testprogram
Dim Voltmeter As New HP34401
Dim Currentmeter As New HP34401
Dim Supply As New HP6624

Private Sub Form_Load()
    GPIBinit
    Voltmeter.Address = 22
    Currentmeter.Address = 23
    Supply.Address = 5
```

```

        Supply.Assignto = 3      ' we use channel 3
    of the supply
End Sub
Private Sub Quitprog_Click()
    Bye
End
End Sub
Private Sub sweep_Click()
    For x = Val(startvalue.Text) To
Val(stopvalue.Text)
        Supply.voltage = x
        volts = Voltmeter.measure
        Current = Currentmeter.measure
        display.Text = display.Text & volts & ":"
& Current &
                                _ vbclrf
    Next x
End Sub

```

The initialization section is limited to establishing the GPIB link and assigning addresses to the instruments. From then on you simply treat your instruments as any other control.

26.6.5 Developing ClassWork Modules

Whilst ClassWork comes with a number of modules, you might need to write some yourself. In order to maintain the functionality of ClassWork there are a number of rules to follow.

A ClassWork module is a piece of Visual Basic code that resides in a Class. Whatever functionality you want to implement is up to you. But, in order for a Class to be a real ClassWork module the following thing should be in place:

26.6.6 Module Header

The ClassWork header should contain information about the library and the instrument covered by the library. A sample header can be found below. It marks clearly that it belongs to the ClassWork framework in the first two lines.

Next it specifies that this library is for an HP34401 System multi meter from Hewlett-Packard. The initial release of this particular piece of code was done on 15/11/1999 by Vincent Himpe, and some changes have been made on a later date. More detail could be given about exactly what and why but this is at the developer's discretion.

```
' *****
' *****
' ClassWork Library
' Released under OpenSource Policy
'
' Instrument : HP34401 System multimeter
' Manufacturer : Hewlett Packard
'
' Initial release   15/11/1999
John.D.Designer
' Update           25/02/2000
John.D.Designer
' *****
' *****
```

26.6.7 Internal ClassWork variables.

The next thing to do is to declare the two internal variables that are required by ClassWork.

These can then be followed by the definition of the variables you might require. Since all derived objects run in their own memory space they will each use their respective copy of the variable.

```
Private v_address    ' GPIB address for this
device
Private v_entity     ' entity in multichannel
devices
```

v_address:

is used as an internal placeholder for the GPIB address assigned to an object derived from the class.

v_entity:

Is used to indicate which part of an instrument is targeted in case of a multimodule instrument. Multimodule instruments are defined as instruments that share the same physical GPIB port but have multiple in or output's all behaving in the same manner. For instance: a dual or triple channel power supply or a 10-channel multimeter.

26.6.8 Initialize and Terminate events

Class_Initialize :

Whenever an instrument is derived from the class (this happens the moment the program executes) the initialize event will be fired. ClassWork uses this event only to notify the user that an object has been derived from this class. The message is being sent out using the standard Logentry command belonging to GPIBcore.

Besides this you can implement whatever startup code might be required for your class.

Now what is the point of sending this comment? It informs the user how many instruments his program uses and of which type they are.

Example:

```
Public Sub Class_Initialize ()  
    Logentry "'ClassWork spawned a HP34401  
    instrument"  
End Sub
```

Class_Terminate :

The Class_Terminate is fired when the object derived from the class is destroyed. In case of a GPIB device this event is used to close the GPIB channel and release the address.

Example:

```

Private Sub Class_Terminate ()
    GPIBClose v_address
End Sub

```

26.6.9 Address assignment

In case of a GPIB device the address property should be implemented. You are free to implement a 'get' statement but this is not required.

Whenever the address is assigned (this can be during startup or it can also mean a change of address) this should cause the current address to be released, and the new address being assigned. The assigned address should be stored in the internal variable v_address. Throughout the rest of the code you must use this v_address when referring to your instrument

```

Public Property Let address (addr)
    GPIBClose v_address
    v_address = addr
    GPIBOpen v_address
End Property

```

26.6.10 AssignTo assignment

This function might not always be applicable to your instrument but it **MUST** be implemented to maintain the highest possible level of compatibility. Simply store the number in the internal v_entity variable. Whenever you need to refer to a certain channel you use this v_entity variable. Again, here you are free to supply a 'get' command but it is not required.

```

Public Property Let Assignto (channel)
    v_entity = channel
End Property

```

26.6.11 Global Lead-in code overview

The whole picture looks like this:

```

! *****
*****

```

```

' ClassWork Library
' Released under OpenSource Policy
'
' Instrument : HP34401 System multimeter
' Manufacturer : Hewlett Packard
'
' Initial release   15/11/1999
John.D.Designer
' Update           25/02/2000
John.D.Designer
' *****
*****
Private v_address ' GPIB address for this
device
Private v_entity ' entity in multichannel
devices
Public Sub Class_Initialize ()
    Logentry "'ClassWork spawned a HP34401
instrument"
End Sub
Private Sub Class_Terminate ()
    GPIBClose v_address
End Sub
Public Property Let address (addr)
    GPIBClose v_address
    v_address = addr
    GPIBOpen v_address
End Property
Public Property Let Assignto (channel)
    v_entity = channel
End Property

```

26.7 General Rules for ClassWork module development

While the previous chapter described the required criteria to develop a ClassWork library, this section will describe an additional framework. It is advised to follow these guidelines to insure maximum compatibility. This chapter will try to show you when to use a property, method or event when creating module functions.

26.7.1 Properties

RULE: If the item is not a real result of a primary function of the instrument, or the item is a setup parameter that configures the common (not directly measurement related) behavior it should be implemented as a property (using Property Let). The data applied should be of the numerical type

(Either a number or a Boolean). The data can also be contained in an array.

Examples:

A power supply is not a real 'measurement' device. It does not really 'measure' but it supplies you with power. Sure you can read back the current it is actually delivering, but that was not the main goal of the instrument. So voltage and current are not real results of the instrument. To the supply the Voltage and Current set are 'properties'. When retrieving them you can of course really read them back and return these as result of the action.

In a multi-meter, the number of digits used does not really determine the nature of the measurement. It has of course effect on the precision of the measurement but it has nothing to do with the physical quantity that needs to be measured. Selecting the physical quantity, deciding between voltages, current, resistance is a different matter. Here you are changing the nature of the measurement and thus the measurement related behavior of the instrument.

26.7.2 Methods (Sub)

RULE: A Sub(routine) is used whenever you change the instruments measurement related behavior of the measurement but don't perform an actual measurement. The functionality desired cannot be expressed using numbers.

Examples:

Selecting physical quantities like Voltage or Current for a multimeter can be classified under this.

The same applies to selecting a function for a Waveform generator. Here you can implement Subroutines to specify the kind of waveform to be generated.

Of course you could make a list with constants defining the Voltage = 1, Current = 2 etc... But this will cause problems. Not everybody will use the same conventions and this will then lead to code that is again not portable.

26.7.3 Methods (Function)

RULE: A Function is used whenever you retrieve a primary measurement result. The result is a single number or Boolean value. The function can take arguments.

Examples:

The result of a measurement performed by a multimeter. The multimeter was previously set up with a number of digits (using a Property construction) and an indication of the nature of the measurement (VoltageDC using a Subroutine)

26.7.4 Special Cases

Sometimes you will run into cases where you need to perform operations that cannot easily be catalogued as one of the above, or that return different kinds of information than defined in the rules.

Example: Returning arrays:

This is a typical example. You retrieve a table with data representing a waveform from an oscilloscope.

Visual basic does not allow you to return this type of data using a Function.

In this case implement it as a Subroutine and change the content of the arguments from within the subroutine.

In all other cases, implement your functions as Subroutines (if not returning data) or as Functions (if returning single numbers)

26.7.5 ClassWork implementation of the HP34401 driver

```

' *****
' ****
' ClassWork Library
' Released under OpenSource Policy
'
' Instrument : HP34401 System multimeter
' Manufacturer : Hewlett packard
'
' Initial release 15/11/1999 V.Himpe
' *****
' ****

Private v_address ' GPIB address for this
device
Private v_entity ' entity in multichannel
devices

Public Sub Class_Initialize()
    logentry "'Classwork spawned a HP34401
instrument"
End Sub

Public Property Let Address(addr)
    v_address = addr
    GPIBopen addr
End Property

Public Property Let Assignto(channel)
    v_entity = channel
End Property

Public Sub VoltsDc()
    GPIBwrite v_address, ":CONF:VOLT:DC"
End Sub

Public Sub VoltsAc()
    GPIBwrite v_address, ":CONF:VOLT:AC"
End Sub

Public Sub CurrentDC()
    GPIBwrite v_address, ":CONF:CURR:DC"
End Sub

```

```
Public Sub CurrentAC()  
    GPIBwrite v_address, ":CONF:CURR:AC"  
End Sub  
  
Public Sub ohms4()  
    GPIBwrite v_address, ":CONF:FRES"  
End Sub  
  
Public Sub Frequency()  
    GPIBwrite v_address, ":CONF:FREQ"  
End Sub  
  
Public Sub Period()  
    GPIBwrite v_address, ":CONF:PER"  
End Sub  
  
Public Sub ohms2()  
    GPIBwrite v_address, ":CONF:RES"  
End Sub  
  
Public Sub Trigger(mode)  
    Select Case mode  
        Case 1      ' external ( vanachteren )  
            GPIBwrite v_address, "TRIG:SOUR:IMM"  
        Case Else ' internal  
            GPIBwrite v_address, "TRIG:SOUR:EXT"  
    End Select  
End Sub  
  
Public Function measure()  
    GPIBwrite v_address, "INIT"  
    GPIBread v_address, "FETCH?"  
    measure = ibret  
End Function
```

If you compare this block of code with the implementation as a standard module you will find a lot of similarities. Actually it is very simple to convert a standard module to a ClassWork class. Just glue on the header, change some information and maybe clean up the code a bit. The hardest part will be deciding how to implement a certain function. Will it be a Method (Function, Sub), a property or an Event. If you follow the guidelines layed out before this will not be that hard either.

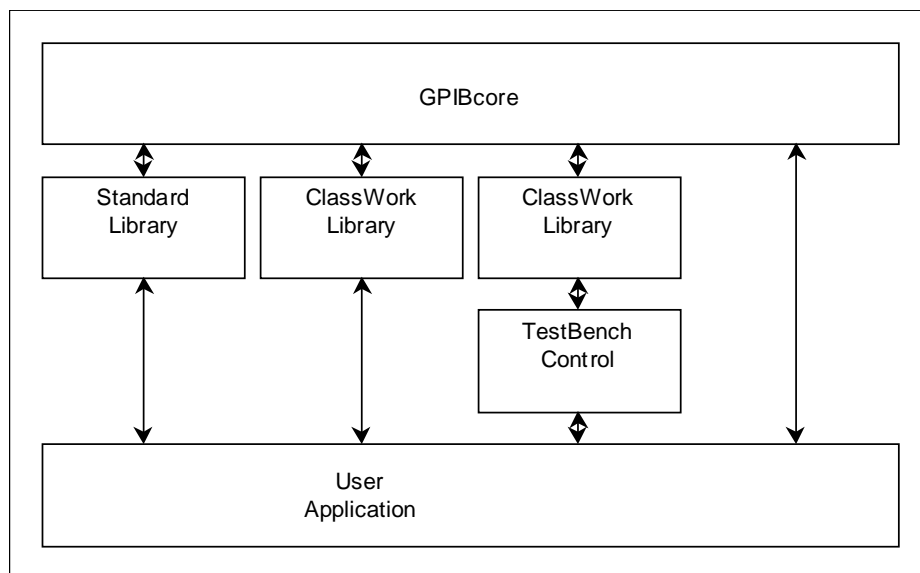
26.8 TestBench

TestBench is an add-on to ClassWork. It provides a simple way to build control panels for your instruments. TestBench objects are implemented as ActiveX controls and distributed in source-code format.

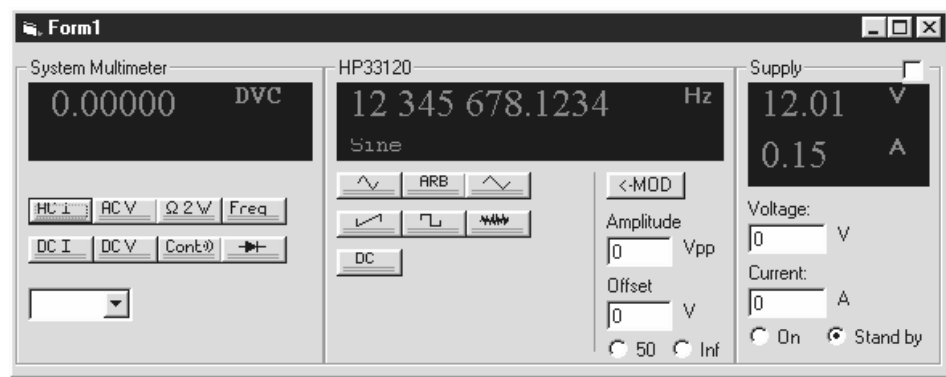
A TestBench control interacts directly with a ClassWork library. The following example shows how to link a TestBench control to an instrument. The name of the TestBench object is DVM1.

```
Dim MyVoltmeter as new HP34401  
Set DVM1 = Myvoltmeter
```

TestBench controls implement a basic set of functions commonly found in the instruments they can be mapped to. A Testbench control works as a pass-thru channel for the instrument controls



The above picture shows the implementation possibilities of all possible instrument control libraries. A program using TestBench can look like the image below



The TestBench controls feature a similar look and feel. Of course you will not find any special features of the instruments here. The controls are written in such a way that they can interface with any ClassWork target.

It speaks for itself that you don't have to try to attach the Supply control to a multi-meter !. This will immediately lead to errors. Having the TestBench control installed and locked to a ClassWork target does not mean that you lose access to the ClassWork library . You can still call special functions in the ClassWork library to set up items that the TestBench control does not handle for you.

Chapter 27 :

Designing Test Programs

Writing a test program is not unlike writing an ordinary program, however there are some differences in the construction and programming style required. The most important thing to keep in mind is to write clean code. It doesn't hurt to write 5 extra lines of code if it makes the program more readable. On the risk of becoming boring I will repeat the most important standard programming techniques here

27.1 Clean code

Make simple code. 5 simple commands are easier to understand, modify and support than 1 complex command.

27.1.1 Modular programming.

Divide your program in functional subroutines and/or functions. Subroutines that must be accessible everywhere throughout the program should be contained in separate modules. Give your subroutines and functions an understandable name. Also, when passing variables to and from subroutines, give them a meaningful name. Remember that the Visual Basic IDE will show you the variable names while you are writing code that calls the function or procedure. Also where possible you should typecast your variables

Example:

```
Function FindBiggestInteger (value1 as integer,value2 as integer) as integer
```

```
Function Fbi(v1,v2)
```

The first example shows a meaningful function name, a clean typecasting of the two expected variables, and a typecasting for the return value. The second line shows a crappy declaration for the same routine. While the name FBI (Find Biggest Integer) might mean something for the original programmer of the code , it might confuse someone who has to maintain or alter the code.

27.1.2 Documenting code.

Wherever possible write down some information about the code you are writing. In particular this is important if you are doing mathematical or logical operations. It makes the code understandable for someone else or even for you. You might find yourself wondering how on earth a certain piece of code works, even if you wrote it yourself just weeks before. Supporting an undocumented piece of code can be very hard. Writing documentation takes only a minimal amount of time, it doesn't waste space in the final program and it doesn't take away execution speed.

27.1.3 Use indentation and CamelWriting.

This improves the overall readability of the code. It becomes clearer which lines are contained between decision blocks of code. Also, limit the number of commands to one per line.

```
Function  
thisisanunreadableandlongfunctionname(x,y)  
x=x+x:y=y+1:x=x/y:x=int(x/y)  
thisisanunreadableandlongfunctionname=x+x/sin(x*  
y)  
end function
```

The above could be rewritten as

```
Function DoSomething (x,y)
    x = x+x
    y = y+1
    x = x/y
    x = int (x/y)
    DoSomething = x+x / sin (x*y)
End Function
```

Fortunately the Visual Basic editor helps you out using different colors for variables , commands , and comment. It also enforces CamelWriting for its internal function names. Besides the above mentioned items there are some additional rules you might want to follow

27.2 Accessing instruments and hardware

In order to keep a modular program that can be maintained for a long time you should divide it into functional units

27.2.1 Accessing instruments

Whenever you access instruments, try to use the provided function libraries (standard libraries or ClassWork objects). IF you find yourself in the situation when there is no ready made solution then build a library of your own. Don't write low level code to access an instrument anywhere I your program. Instead build the library. In future programs it might be useful , and it makes the program far more readable.

27.2.1 Accessing hardware in the computer

The same rules as for accessing instruments can be applied here. Try to shield the real program for the low-level work by using intermediate layers.

27.3 Collecting data versus Analyzing

The first aim of a test and measurement program is to collect data , not analyze it. To analyze collected data there are far more powerful tools available (Excel

,MathCAD , Mathlab etc). Therefore you should concentrate your programming effort on just that. Store the retrieved data in log-files and post process them later.

27.4 Creating log files

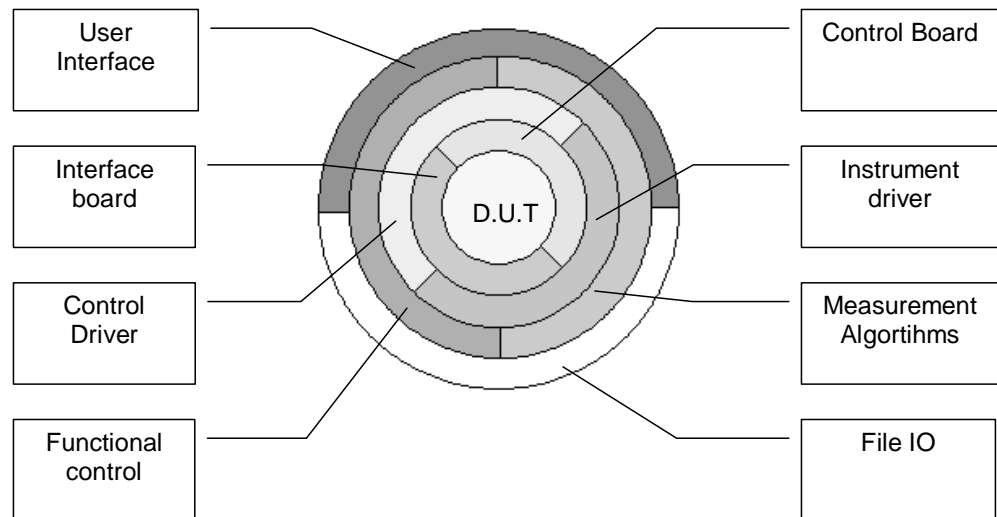
When you are writing a program to collect data , you will need a means to off-load this data. Saving measurement data typically happens in so call 'log-files'. Besides the measurement data it might be interesting to store additional information as well :

- **The name of the tester**
This can prove helpful . In case of any problems you can always contact this guy for more information.
- **The date and time**
To do some kind of version control on test reports
- **The number of the sample**
In case you find bizarre results you might want to re-test this particular devices
- **The temperature at which it was tested**
All electrical parameters drift over temperature. So it's wise to know this. Furthermore some test requires temperature sweeps.
- **All initial conditions of the test**
In case you find crazy data this can prove helpful to reproduce the exact conditions of the test. This information can be of utmost importance to recreate a certain effect in a component.

In the next chapter some special programming techniques will be described. Amongst them is the generation of CSV files. This is a format particularly useful to create log files. It can be read by almost any data processing program like Excel MathCAD etc.

27.5 Anatomy of a well structured test-program

A well structured program looks like an onion. You can peel it away layer by layer until you reach the core : the component under test.



Layer 1:

At the outmost layer is the user interface and the data storage handlers. Depending on the actions of the user and/or files (scripts !) messages will be passed to certain measurement algorithms (U/I sweeps, level detection, ADC measurement etc) and to the functional control routines that control the behavior of the chip. The returned data will be displayed on the user interface or stored to disk.

Layer 2:

Measurement Algorithms will either talk to the instrument drivers, to collect data or setup instruments, or the control drivers to configure the chip and or test-board. The Functional control routines will implement the operations required to transport information to and from the chip (serial

buses , SPI etc, test-pins). Retrieved information is delivered to the Measurement algorithms and functional control algorithms.

Layer 3:

The instrument driver will control instruments and make them apply the correct voltages, currents and signals to the component. The control driver will orchestrate the operation of all hardware on the test-board and the interface to the computer.

Layer 4:

The interface board will switch these signals to the appropriate pins of the component. The control board (Printerport or DAQ) will apply digital stimuli to our chip directly from the computer. All of this happens under total control of the instrument drivers and the control drivers.

Core :

And finally we find our component under test. This little critter will take all of the stimuli delivered to it , process them and (hopefully) return us with some information that then can be offloaded and presented to the user and stored for later evaluation.

Every module in any layer should be constructed in such a fashion takes information from above and passes it down below. Anything coming back from the layer below is processed and passed on to the layer above. However , you must avoid at all times to 'skip' layers. The program loses modularity and portability. Every subroutine must have an unambiguous task. By following these rules you will have a program that has a lowest level the hardware interface. The moment something changes there you can update the entire project by patching that and only that layer.

Chapter 28:

Special Programming techniques

During the development of test programs you will encounter some specific problems. Most of these problems can be tackled using some logic thinking. But some problems can be hard to find a solution for. This chapter will attempt to clarify some of the specific problems you might struggle with when developing your program.

28.1 Stream Interpreting

What is stream interpreting. Simply said it is the generation of a data-stream to a target. This stream can be accompanied by a number of control signals such as a select line and a clock or strobe line. When the output of the stream generator is fed to an output port then we are talking about a bit-banged interface. You are in fact manipulating bits to 'emulate' one interface onto another interface.

28.1.1 Monolithic Program

Let's take an example with a shift register.

A shift register has a clock input a reset line and a data input. We will connect this shift register on a standard parallel port (In this case the Printerport, but it could be an IO port of the 8255 controller on our home-built IO board as well).

The reset line will be connected to bit D0, the clock line will be attached to Bit D1 and the data in of the shift register will be connected to D2 of the io port.

Thus we can already define some constants

```
Const Shift_RST = 1
Const Shift_CLK=2
Const Shift_DIN=4 ' bit 2 is worth the decimal
value 4
```

The first thing we should do is to build our stream from the supplied data. The easiest way to do this is represent the data as a string containing 1's and 0's. Fortunately the GPIBCore contains the Bin\$ function.

The stream could also come from a graphical user interface for that matter. Anyhow , for the moment the origin of the stream is of no importance.

To interpret the stream we have to make some sort of scanning algorithms that checks the value of a certain character in the string and sets the output bits accordingly.

```
Sub SendStream(stream$)
  for x = 1 to len(stream$)
    if mid$(stream$,x,1)="1" then
      out Ioport,shift_din
      outIOPort,Shift_Din+Shift_CLK
      out Ioport,Shift_Din
    else
      out Ioport,0
      outIOPort,Shift_CLK
      out Ioport,0
    end if
  next x
End Sub
```

The above block of code will send scan the stream independent of its length and send out the appropriate bit. A clock pulse is generated after the update of the Din pin as well. To reset the shift register we can simply put 2 more commands before we start scanning the string.

The final code would look like this:

```

Sub SendStream(stream$)
    out Ioport,Shift_RST
    out Ioport,0
    for x = 1 to len(stream$)
        if mid$(stream$,x,1)="1" then
            out Ioport,shift_din
            outIOPort,Shift_Din+Shift_CLK
            out Ioport,Shift_Din
        else
            out Ioport,0
            outIOPort,Shift_CLK
            out Ioport,0
        end if
    next x
End Sub

```

28.1.2 Modular program

The disadvantage of the above program is that it is one block of monolithic code that is hard to port to different hardware. Of course you can change the pin definition by altering the definition of the 3 constants involved, but that is not real portability. What if you need a number of instructions control a certain pin.

Therefore you need to make your program modular. If you rewrite the above example in the following way you will get a truly portable program. Besides the fact that it is better portable it is also better readable. Even someone who never programmed before or doesn't understand the BASIC language (I can't image they exist) can understand this code.

```

Sub SendStream(stream$)
    RSThi
    RSTlo
    for x = 1 to len(stream$)
        if mid$(stream$,x,1)="1" then
            DINhi
            CLKHI
            CLKLO
        else
            DINlo
        end if
    next x
End Sub

```

```
        CLKhi
        CLKlo
    end if
next x
End Sub
```

Now all we have to do is define the routines that hook our operations to the hardware.

```
Dim mask as integer

Sub RSThi
    Out Ioport+1,128
End Sub

Sub RSTlo
    Out Ioport+1,0
End Sub

Sub CLKhi
    mask = mask or 1
End Sub

Sub CLKlo
    mask = mask and (255-1)
End Sub

Sub DINhi
    out ioport+2,(inp(ioport+2) or 8)
End Sub

Sub DINlo
    out ioport+2,(inp(ioport+2) and (255-8))
End Sub
```

The above code shows that the code to handle the RST pin is changing a bit in a certain register. The other 2 bits physically reside in another register.

The CLK pin is a physical pin on a parallel port. But we are not allowed to change that status of the other pins, and we can't read from the port. So we need to declare a variable that holds the status of our IO port. Therefore the declared variable *mask*. Other routines in our program may use this Mask bit as well.

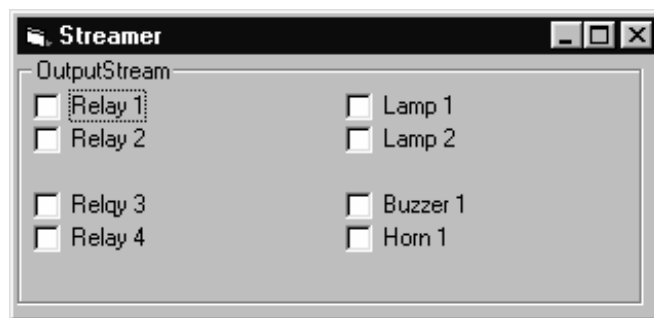
The DIN pin belongs to a port that can be read back. There I used code that retrieves the current setting and alter them. This is called a read-modify-write operation.

As you can see this code is truly portable and adaptable to any situation possible. It is sufficient to adapt the layer below the stream interpreter to port it.

28.1.3 Creating the stream

So far I've made the interpreter and the io routines. But where do we get the stream ? There are a number of options : you create it by converting a value to a binary stream using the BIN\$ function in the GPIBcore. Or you can ask the user to type the stream (duh). Or maybe you can provide him with a graphical user interface. And the latter is exactly what I'm about to do here.

Remember the arrays of objects described in one of the very first parts of this book ? Well that is what I am going to use. Simply make an array of checkboxes. Each checkbox represents one bit in the stream. If you order them logically then the element 0 will represent bit 0 , element 2 represents bit 1 and so on.



The creation of the stream is then attached to the Click event of the array of controls.

```
Sub Bitstream_Click(index as integer)
    stream$=""
    for x = 0 to number_of_elements
        if Bitstream(x).value =1 then
            stream$="1"+stream$
        else
```

```
        stream$="0"+stream$  
    end if  
next x  
sendstream(stream$)  
End Sub
```

And there you have it : a graphical interface to our stream interpreter. If you now assign a meaningful name to every checkbox you have a very user friendly program.

28.2 Report generating on a printer

Most people are afraid of creating professional reports on a printer. Sure , sending just text to a printer is easy , but to create a professional print-out including graphics is a different matter ... or is it ? Normally the development of code that makes a good looking hardcopy can be cumbersome but , if you use some tricks the job gets a whole lot easier.

The first thing to keep in mind is that the printer is treated as an object in Visual Basic. Actually the printer is part of a collection of printers. All manipulations you can do with the data to be printed can be performed on a form as well. That means : making a preview is a snap. On a printer you can even program a coordinate system !.

28.2.1 The Printer Object

The Printer object enables you to communicate with a system printer (initially the default system printer). The Printers collection enables you to gather information about all the available printers on the system.

```
Printer  
Printers(index)
```

The index placeholder represents an integer with a range from 0 to Printers.Count-1.

Use graphics methods to draw text and graphics on the Printer object. Once the Printer object contains the output you want to print, you can use the EndDoc method to send the output directly to the default printer for the application.

You should check and possibly revise the layout of your forms if you print them. If you use the PrintForm method to print a form, for example, graphical images may be clipped at the bottom of the page and text carried over to the next page.

28.2.2 The Printers Collection

The Printers collection enables you to query the available printers so you can specify a default printer for your application. For example, you may want to find out which of the available printers uses a specific printer driver. The following code searches all available printers to locate the first printer with its page orientation set to portrait, then sets it as the default printer:

```
Dim X As Printer
For Each X In Printers
    If X.Orientation = vbPRORPortrait Then
        ' Set printer as system default.
        Set Printer = X
        ' Stop looking for a printer.
        Exit For
    End If
Next
```

You designate one of the printers in the Printers collection as the default printer by using the Set statement. The preceding example designates the printer identified by the object variable X, the default printer for the application.

Note If you use the Printers collection to specify a particular printer, as in Printers(3), you can only access properties on a read-only basis. To both read and write the properties of an individual printer, you must first make that printer the default printer for the application.

28.2.3 NewPage

This method ends the current page and advances to the next page on the Printer object.

object.NewPage

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

NewPage advances to the next printer page and resets the print position to the upper-left corner of the new page. When invoked, NewPage increments the Printer object's Page property by 1.

28.2.4 EndDoc

Terminates a print operation sent to the Printer object, releasing the document to the print device or spooler.

object.EndDoc

The object placeholder represents an object expression that evaluates to an object in the 'Applies To' list.

If EndDoc is invoked immediately after the NewPage method, no additional blank page is printed.

28.2.5 Example

This example uses the EndDoc method to end a document after printing two pages, each with a centered line of text indicating the page number. To try this example, paste the code into the Declarations section of a form, and then press F5 and click the form.

```
Private Sub Form_Click ()  
    Dim HWidth, HHeight, I, Msg ' Declare  
    variables.  
    On Error GoTo ErrorHandler ' Set up error  
    handler.  
    Msg = "This is printed on page"  
    For I = 1 To 2 ' Set up two iterations.  
        HWidth = Printer.TextWidth(Msg) / 2 ' Get  
        half width.
```

```
        HHeight = Printer.TextHeight(Msg) / 2 ' Get
half height.
        Printer.CurrentX = Printer.ScaleWidth / 2 -
HWidth
        Printer.CurrentY = Printer.ScaleHeight / 2 -
HHeight
        Printer.Print Msg & Printer.Page & "."
        ' Print.
        Printer.NewPage      ' Send new page.
    Next I
    Printer.EndDoc  ' Printing is finished.
Exit Sub
ErrorHandler:
    MsgBox "There was a problem printing to
your printer."
Exit Sub
End Sub
```

Chapter 29:

Building user interfaces.

An important point of concern is building a user interface for your test program. Designing an intuitive easy-to-use program is not an easy job. Too many programs are simply 'kludged' together into a working state and then used 'as-is'.

This should not be the case for your programs. By the time you have reached this chapter you should already know a lot about the BASIC language, about designing windows programs, controlling instruments, writing structured and expandable programs. And it is exactly all of the above knowledge you will need to create a friendly and easy-to-use interface for your program.

29.1 Build a splash screen and design a logo and icon.

No kidding! No time to be modest now. It's okay to brag a bit about the program. Design a catchy icon (16*16 and 32*32 bit in 16 colors) to put on the title bar of your program. Make the same icon in a bigger format to put on a splash screen.



The splash screen should contain at least the name of the program , copyright information , and a version number. You can also put some info about the company or person who wrote it. And a nice graphic doesn't hurt. As an example you can take a look at splash screen from Word or Excel or even Visual basic itself.

Graphics can mostly be downloaded from the web. To construct your graphic you can mix several images and texts together using a program like Paint Shop pro that can be downloaded for free from the internet.

The Splash screen is at the same time a good spot to start allocating the memory you need , loading any data you need and initializing any instruments you need. You can do this most easily by loading the sub-forms of your program into memory without showing them. This will effectively allocate whatever resources are needed.

If you have clean partitioned code you can write a subroutines that initializes the hardware you will use , like GPIB or printer ports , to a known state as well. When all this is done you simply pass control to the main form of the program and unload the splash screen.

```
Sub Splash_load()  
    DoEvents ' Make sure we are being shown !
```

```
load frm_MainForm
' update a statusbox here
label1.caption = "Loading forms ..."
DoEvents
load frm_Setup
DoEvents
load frm_ExtraForm1
DoEvents
load frm_EveryOtherForm
label1.caption = "Initializing system"
DoEvents
InitializeSystem
DoEvents
mainform.show
DoEvents
Me.hide
End Sub
```

The above block of code does all of this and shows some status information to the user as well. Make sure you don't forget the DoEvents statement between every load command. This allows Windows the necessary time to handle its internal management. Some of this time is used to manage the just loaded form.

Besides giving your program a professional look it also speeds up the perceived speed of your program.

29.2 Constructing the Main form.

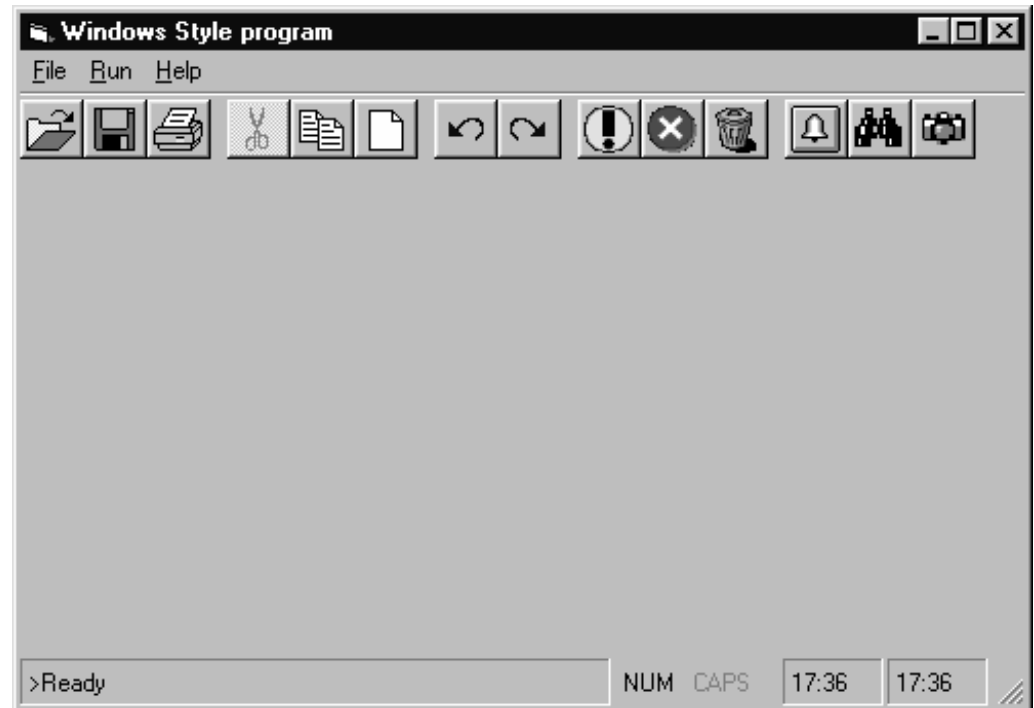
The main form is the most important piece of your program. You should think about the way you want to organize it. After all this is the place that the user will be looking at most.

29.2.1 The Workplace of your program

A good main form is designed as a switchboard. You have all the things you use the most in front of you. All extra information is hidden in additional screens. Remove any superfluous information from the screen and into sub-forms.

Depending on the time you have to write the program you could implement 'dockable' toolbars and other fancy stuff but this is not a must for a good

program. Sometimes too many gizmo's can be annoying too. After all a test program is bound to be used by technical minded people , and they don't care about funky colors (well ... most of them that is) .



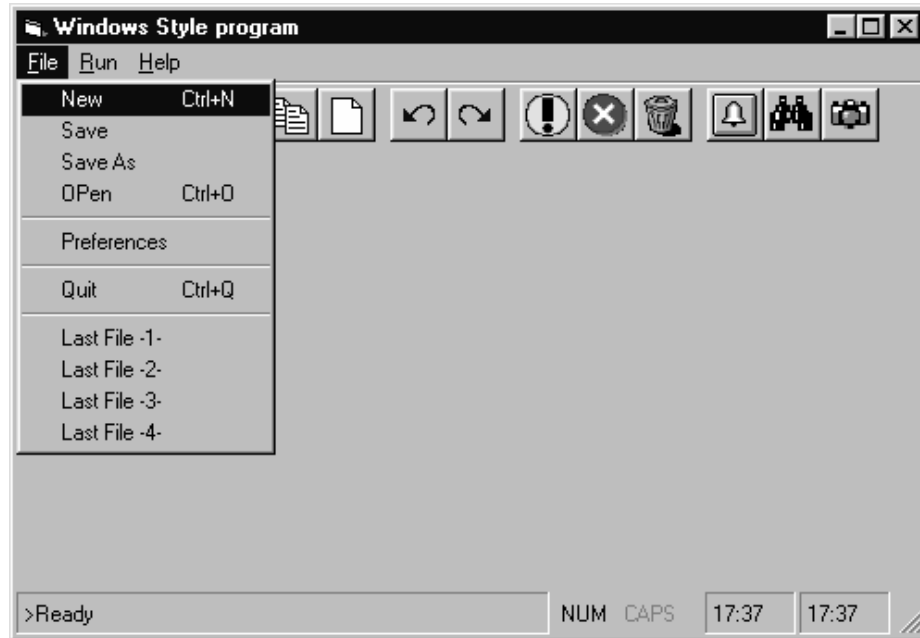
A nice to have thing is a status bar. Show the date and time and a single line of info. This gives your program that little 'extra' touch and doesn't cost much. The info-line can be used to tell the user something about what he is doing or going to do.

29.2.2 Construct a Decent Menu

A decent menu should have a clean layout. Sort the items on a menu per category. File operations should be put under the File menu , Tools under a Tools menu and Help under a Help menu. The golden rule here is : Keep it logical. Wherever possible , try to assign hotkeys. It makes the program friendlier and it doesn't cost you a single line of code. The VB compiler takes

care of the hotkeys. Look at standard windows programs to add hotkeys to your menus.

Images on menus can look cool but take too much time to implement. It's better to construct a separate toolbar than embedding the icons in the menu itself.

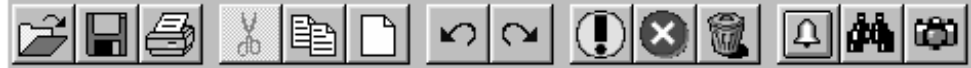


29.2.3 Tooltips

Try to put some meaningful information in the ToolTipText property of the controls on the screen. It's not a lot of work but can provide that extra bit of information to the user when he needs it. This can literally save you a bunch of calls for support from your user. And it does not require you writing any code.

29.2.4 Toolbars

A toolbar can be a very handy thing. A single icon might mean more than a thousand words. But a superfluous toolbar will scare the user as well. Put only the real tools on the toolbar, and provide additional information using the ToolTipText property of the toolbar control. The icons on the toolbar should have a clear meaning. Don't put the symbol for Cut (scissors) if you are going to use it to paste text. Sounds pretty obvious ? Yes in this case. But what about things like Print and Print setup ? The answer is simple:



A toolbar contains only single-action objects. A printer setup button is out of place on a toolbar. Every button should perform an action that needs no further information from the user. No pop-up boxes or fly-outs. Just plain and simple one-click does it all actions. Typical examples are:

Save current file, Print current file, Cut, Copy, Paste, Undo, Redo, Help, Run, Stop, Break, Continue etc.

Things that are totally out of place on, a toolbar :

Color selectors, printer selectors, setup screens for part of the program.

Furthermore the toolbar should remain constant throughout the use of the program. That means that no parts of it should suddenly lose functionality.

29.3 Organizing Objects and controls.

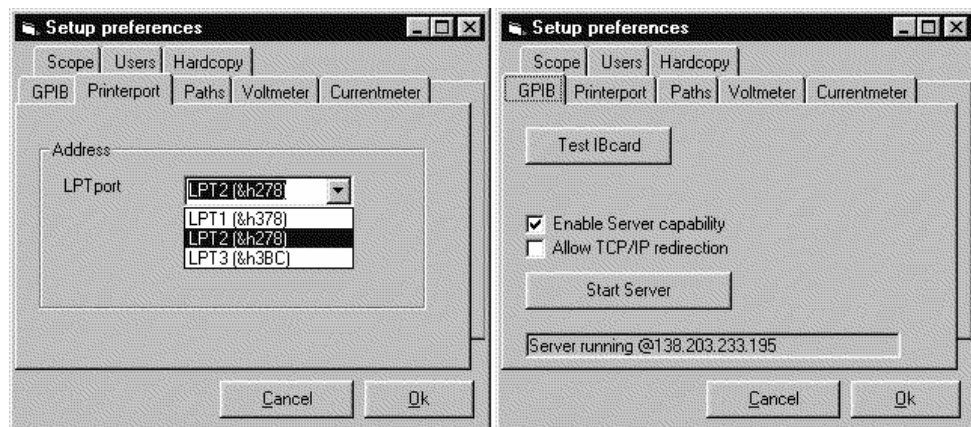
The visual appearance of a program is heavily depending on the arranging of objects and controls. To make your life as a programmer easier you should frequently use the frame object. Group all items that belong together on a frame. Then start constructing sub frames on this frame and rearrange the objects again. You can dynamically show and hide frames and sub-frames. This avoids having to pop-up smaller windows all the time and it keeps the desktop orderly.

Other parts of your program should be where the user expects them Toolbars at the top , just below the menu , Status bars at the bottom. And in between the m the central working area of your program. Try to keep your layout style as

consistent as possible with what is commonly used in any Windows based program.

29.4 Configuration and tool forms

Whenever you want to provide extra forms to allow the setup of several parameters, try to be as specific as possible. Don't make one huge form. Break it down into small chunks of information, but at the same time avoid 500 different setup forms. The best way is to make a separate setup form that contains a tab-strip. Depending on the actual stuff the user wants to configure he can click on one of the tabs and then perform the setup for that section. A typical example would be the following :



29.6 Help files

Whenever possible construct a help file for your program. This involves writing some text in a specified format and running it through a so called Help-Compiler. The details about this process are a bit too extensive for this manual but can be found in the on-line help of Visual Basic.

For most of the test programs developed the use of ToolTipText is far more useful. When you feel additional help is required then you always have a pop-up box display some more text.

Chapter 30:

Some more case studies

SPI stack on LPT

This sample shows a practical implementation of stream generation on a printer port. It shows both methods , one functional and on graphical.

Data export to file

Formatting data in an orderly way is not always easy. A sample application that formats and writes data to CSVfiles (Excel format)

Building a U/I plotter using standard GPIB

A sample that builds a simple curve-tracer using standard GPIB calls and instrument libraries

Building a U/I Plotter using ClassWork

A sample that builds the same curve-tracer but now using ClassWork libraries

Building a U/I Plotter using TestBench

The same sample all over again but now using a TestBench front panel for ClassWork.

Case Study 11 : SPI stack on LPT

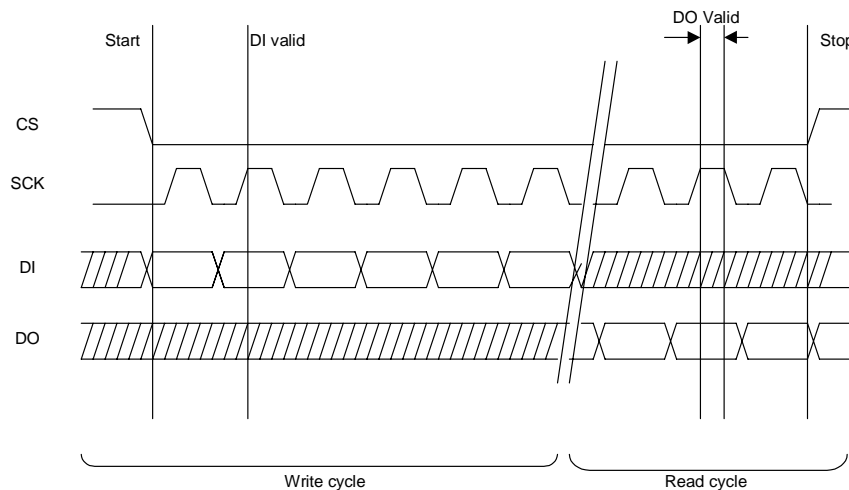
A typical operation you might perform is to shift data in and out of a component. Most likely you will use a printer port for this purpose , since it is standard property on any computer. You could as well implement this stack for a custom IO board or ready made IO board. The only layer you would have to adapt is the hardware access.

If we follow the rules of a good structured program we will begin by writing the low level interface code that will glue all routines to our hardware.

The SPI bus defines 4 pins :

- SCK : Serial Clock
- CS : Chip Select
- DO : data to the chip
- DI : data from the chip.

The SPI bus uses a strict sequencing diagram that tells us that , in order to talk to a component , we should have the clock line low and then take the chip select line low.



The above diagram shows us the timing information of the SPI bus on every consecutive clock pulse (1 rising and one falling edge) the chip will take in or send out data. Data is taken in and sent out on the rising edge. The outgoing data remains valid during the time that the SCK pin is HI. To terminate a transaction you leave the SCK line low and then make CS high.

In our example we will use bits D0 to D2 as output and the BS bit as input. There are a number of ways you can implement the transport routines. Check out chapter 28 for more information. In this example I will use the ClassWork library that controls printer ports

```

Dim LPT1 as New PrinterPort
LPT1.port= &h378

Sub SCKhi
    lpt1.D0 = True
End Sub
Sub SCKlo
    lpt1.D0 = False
End Sub
Sub Cshi
    lpt1.D1 = True
End Sub
Sub Cslo
    lpt1.D1 = False

```



```
End Sub
Sub Dohi
    lpt1.D2 = True
End Sub
Sub Dolo
    lpt1.D2 = False
End Sub

Function SampleDI
    sampled_i = LPT1.BS
End function
```

In case you don't want to use this approach you always 'bit-bang' using 'AND' and 'OR' operations. Next step is to create some higher level routines that perform simple tasks.

```
Sub OpensPI
    Cshi
    SCKlo
    Dolo
    CSlo
End Sub

Sub CloseSPI
    SCKlo
    Dolo
    Cshi
End Sub
```

The above routines generate the timing diagram required to initiate transport and terminate transport. Now all we need are the effective transport routines. Since our routines can be used on several data-lengths we might want to make them adaptive.

```

Sub SendData (dta$)
  For x = 1 to len(dta$)
    if mid$(dta$,x,1)="1" then
      DOhi
    else
      DOlo
    end if
    SCKhi    ' clockpulse generation
    SCKlo
  Next x
End Sub

```

The above routine will always transit an amount of bits equal to the number of characters in the DTA\$.

The final routine is the one retrieving data from the SPI bus. It accepts the number of bits desired.

```

Sub ReceiveData(number_of_bits)
  y$=""
  for x = 1 to number_of_bits
    if SampleSPI=true then
      y$=y$+"1"
    else
      y$=y$+"0"
    end if
  next x
End Sub

```

The final functions will implement the entire SPI frame now. Our final chip could accept a 5 bit address and 8 bit datastream and would then return 4 bits of status information.

```

Function SPI(address$,dta$)
  OpenSPI
  senddata address$
  senddata dta$
  SPI = receivedata (4)
  closeSPI
End Sub

```

The above function would effectively accomplish this entire data exchange protocol. Furthermore ,due to the layered architecture of our library ,it can be maintained and adapted very easily to different hardware platforms. The only things that need to be changed are the routines that glue the SPI stack to the hardware. If we don't want to use the ClassWork library then we simply call other functions from somewhere else , or we implement our own logic there. The entire stack is expandable in all directions: to the user level , to the hardware level and in the functionality level.

The next step might be the construction of a graphical control of the bits you are sending. The easiest way is to create an array of objects that can be set to 1 or zero. Best would be to simply put a number of checkboxes on the screen. When you create these as an array then you can assign the element 0 to bit 0 , element 1 to bit 1 etc. Need more bits ? Put more check boxes. Since all checkboxes will fire the same piece of code the construction of the transport mechanism is simple

```
Sub SPIcheckbox_click(index as integer)
    tmp$=""
    for x = 0 to 7 ` amount of checkboxes
        if SPIcheckbox(x).value=1 then
            tmp$=tmp$+"1"
        else
            tmp$=tmp$+"0"
        end if
    next x

    OpenSPI
    Senddata(tmp$)
    CloseSPI

End Sub
```

And presto ! Instant user access to the entire SPI stream without limiting the adaptability of the code.

Case Study 12 : Data export to file

Most test programs will collect data and have the need to store it somewhere. You can of course develop your own file format but , it might be more interesting if you could use some standard format supported by a lot of programs. Question is , what format ?.

There is one format recognized by almost any data processing program that is even readable for humans : the CSV format or Comma Separated Values format. If you give the filename the extension CSV then these programs will know exactly how to treat these files , and import them in a consistent way into their internal format. Nice side effect is that the file remains readable by your programs as well as by any text viewer too. You could even edit it manually if you would.

Structure of CSV files

```
<entry>,<entry>,<entry>,<entry>,<entry>[CR LF]
<entry>[CR LF]
<entry>,<entry>,<entry>,<entry>,<entry>[CR LF]
<entry>,<entry>,<entry>,<entry>[CR LF]
<entry>,<entry>,[CR LF]
<entry>,<entry>,,<entry>[CR LF]
```

The above syntax shows you immediately all you need to know. Every line contains a number of entries separated by a comma and terminated with a [CR LF] (carriage return-line feed &h13 &h10) pair. The CR-LF you will get automatically if you simply use the print command without terminator (, or ;) to write to a file. So there are no pitfalls there. It is even allowed to have a comma without an entry followed by CR-LF or even two commas without anything in between. Note that for maximum compatibility it is wise to put at least a space in between.

To the CSV import filter the comma means nothing else as 'go to the next column. The data contained in an entry field can be numeric , alpha or alphanumeric. If an entry contains a valid number (1 , 1.2 1.2E+122 , -1.12e-99) and nothing else then this , then it will be correctly imported as a number. If there is any other information in the entry then it will be interpreted as text.

This means that the following CSV file :

```
10.2 , volts , 3.12,Amps[CR LF]
11.5 Volts,12.7 Amps [CR LF]
```

will be read as a

10.2	Volts
11.5 Volts	12.7 Amps

Lets look at an example files

```
This , Is , A , Text [CR LF]
This,is,a,text,too , [CR LF]
This,,is , , also,, valid,,[CR LF]
1,2,1.23,a,b,c[CR LF]
```

When imported this will look like the following :

This	Is	A	Text			
This	is	a	text	too		
This		is		also		valid
1	2	1.23	a	b	c	

Not that trailing commas without text will be stripped from the import. (Some older tools don't do this correctly , but it doesn't matter anyway)

Generating CSV files.

The following piece of code might make life very easy for you

```
Dim CSVfilename$ ` holder for the filename

Sub CSVtext (txt$)
    x = freefile
```

```
        Open CSVfilename$ for append as #x
        print #1,txt$ + ",";
    Close c
End sub

Sub CSVnumber (number)
    x = freefile
    Open CSVfilename$ for append as #x
        print #1,str$(number) + "," ;
    Close c
End sub

Sub CSVnewline
    x = freefile
    Open CSVfilename$ for append as #x
        print #1,""
    Close c
End Sub
```

You simply put the target filename in the variable CSVfilename\$. To add entries you use the subroutines CSVtext and CSVnumber. To terminate a line you simply call CSVnewline.

Case 13 : A U/I plotter using GPIBcore operations

This little program will perform a voltage sweep and measure the current through a load. The resulting data will be stored in an array for later processing.

The first thing we need to do is find out the addresses of our instruments and store them in the program. Next thing is to write the GPIB code to initialize the bus and open the instruments. It is a good practice to close the GPIB bus upon exit, so a small blurb of code will be attached to a Quitprogram menu entry as well.

```
Const PSU =5
Const DVM =22

Sub Form_load( )
    GPIBinit
    GPIBopen PSU
    GPIBopen DVM
End Sub

Sub Quitprogram
    bye
end
End sub
```

The next thing we need to do is create a form that holds entry fields for the start, stop and step value of the voltage sweep. The textboxes will be called respectively STARTval, STOPval and STEPval. Also a textbox called *report* will be used to output the logged data. And finally we need a button to trigger all of this action.

Now we need to know the commands to set the voltage of the supply. Note that these commands are instrument dependent and need to be looked up in the user manual of the instrument

Powersupply : "VSET <channel>: <V.vvv>"

Multimeter : "RANGE:CURRENT DC" and "MEASURE?"

In our case we are going to use channel 1 of the power supply so we need to send it the string 'VSET 1:' followed by the desired voltage . Now that we have this information we can write the main loop of the program.

```

Sub Sweep_click()
    gpibwrite DVM,"RANGE:CURRENT DC"
    for x = val(startval.text) to
val(stopval.text) _
        step val(stepval.text)
        gpibwrite PSU,"VSET 1:"+str$(x)
        GPIBread DVM,"MEASURE?"
        report.text=report.text +str$(x)+" /
"+str$(ibret)
    next x
End Sub

```

The net result is a program that plots exactly what we want to a textbox. Now it's up to you to write file IO or even a graphic charter using an Mschart object , or even your own charter using graphics operations..

Of course in the above example we could have used some of the modules existing for those instruments

Suppose we have a module for a multi-meter (say a HP34401) loaded. The sweep routine could then look like this :

```
Sub Sweep_click()  
    HP34401CurrentDC dvm  
    for x = val(startval.text) to  
val(stopval.text) _  
        step val(stepval.text)  
        gpibwrite PSU,"VSET 1:"+str$(x)  
        HP34401Measure  
        report.text=report.text +str$(x)+" /  
"+str$(ibret)  
    next x  
End Sub
```

As you can see the dedicate calls have been replaced by calls to functions and subroutines in the Library for the instrument.

Case 14 : A U/I plotter using ClassWork operations

This little program will perform exactly the same function as the previous example , except that this time it will use the ClassWork libraries to handle the instruments

The User interface looks exactly the same but most of the code changes

Lets take a loot at the system startup code:

```

Dim PSU as new HP6624 ` Create an object of
class HP6624
Dim DVM as new HP34401 ` Create object from
HP34401 class

Sub Form_load()
    GPIBinit
    PSU.address = 5 ` address of the supply
    PSU.Assignto = 1 ` use first channel of this
supply
    DVM.address = 22 ` address of the DVM
    DVM.CurrentDC ` select Current DC range
End Sub

Sub Quitprogram
    bye
    end
End sub

```

As you can see the initialization block looks a bit different. All of the settings for the instrument are done here. The address has been assigned , the appropriate channel for the supply has been selected and the range for the multi-meter has been specified.

Now it's time to have a look at our sweep function :

```

Sub Sweep_click()
    for x = val(startval.text) to
val(stopval.text) _
        step val(stepval.text)

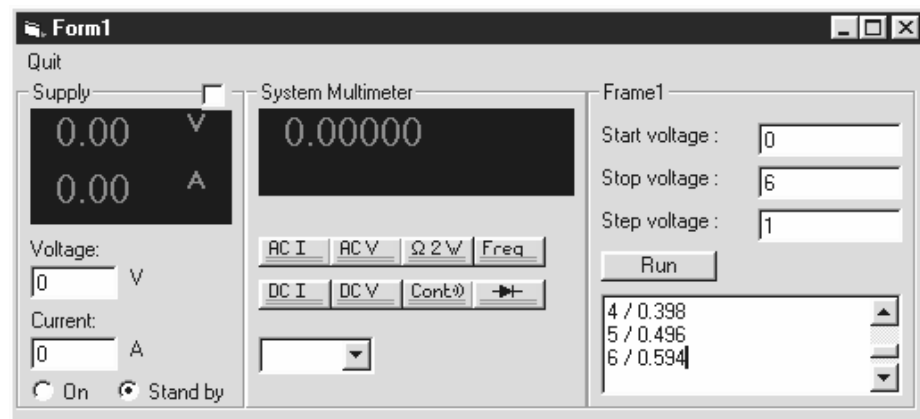
```

```
        PSU.Voltage x
        value = DVM.Measure
        report.text=report.text +str$(x)+" /
"+str$(value)
    next x
End Sub
```

You can see that the way to approach the instruments is exactly like you approach any other object in your program. That is the idea behind ClassWork : it exposes the instruments as objects to your program.

Case 13 : A U/I plotter using TestBench operations

This last example shows you how to add TestBench controls to your program. The code is the same as the previous example with a minor difference in the startup section. But let's take a look at the screen first:



The controls for our program have been moved to a little panel. So nothing special there. In fact it is not necessary to move them to a panel but it looks more consistent with the TestBench controls. Next to the panel are the two inserted TestBench controls: One for the power supply and one for the multi-meter. The power supply has been named PSU and the multi-meter has gotten the name DVM.

Of course now we have to use different names for our supply and multi-meter when we derive them from ClassWork objects.

```

Dim MyPSU as new HP6624 ' Create an object of
class HP6624
Dim MyDVM as new HP34401 ' Create object from
HP34401 class

Sub Form_load()
    GPIBinit
    MyPSU.address = 5 ' address of the supply

```

```
        MyPSU.Assignto = 1 ` use first channel of
this supply
        MyDVM.address = 22 ` address of the DVM
        MyDVM.CurrentDC ` select Current DC range

        Set DVM.Target = MyDVM ` TestBench link to
ClassWork
        Set PSU.Target = MyPSU ` TestBench link to
ClassWork
End Sub

Sub Quitprogram
    bye
    end
End sub
```

The reason I changed the name of the ClassWork objects is so that I don't have to rename anything else in my project. All calls will now be done to the TestBench controls which in turn will pass them to the ClassWork object , that will talk to GPIBCore , which will perform the GPIB I/O until all returning data has been handed back to TestBench.

Sounds complicated ? No it's simple logic applied to a very modular program.

Appendixes

Appendix 1 : Suggested Reading List

Appendix 2 : Datasheet for 8255 controller

Appendix 3 : Win95io users guide

Appendix 1: Suggested Reading List**The Programmers PC Sourcebook Second edition by Thom Hogan**

Microsoft Press ISBN1-55615-321-X

This book is a collection of listings and tables describing every nook and cranny of the IBM PC. From hardware bus connectors to single bit memory location. It even details on the Windows memory usage. The Bible of PC register and memory maps

PC Intern 4 by Michael Tisher

Easy Computing ISBN 09-5157-027-3

Reference book that explains the little known regions of the PC. This provides a good in depth knowledge on how the machine works. This tells the story behind all the registers and interrupts that you can find in the PC sourcebook.

PC Intern 5 by Michael Tisher and Bruno Jennrich

Easy Computing ISBN 90-5167-079-6

While this is the successor of PC intern 4 you really can't live without the old book. This book details more on the Windows environment. A lot of material has vanished (it is still included on the CD ROM that comes with the book) . But when you are on the job it's easier to have NR 4 altogether.

Inside the PC by Peter Norton

Sams Publishing ISBN 0-672-30624-7

A good book that provides a lot of background information about the PC. Includes a lot of sample programs and questionnaires that allow you to evaluate yourself on your knowledge.

The PC inside out by Murray Sargent and Richard L Shoemaker

Addison Wesley ISBN 0-201-62646-2

Provides not only the workings of the PC but gives a crash course on electronics as well. A great book if you're into building your own hardware to fit in to , or attach to the PC. Comes with a nice debugging tool.

The Undocumented PC by Frank van GILLUWE

Addison Wesley No longer available.

This turns the PC inside out and reveals a lot of hidden and little known stuff about the PC. It takes you into uncharted terrain . It details on bugs in BIOS and chip sets. It even points out the BUGS in the CPU's . The accompanying disk contains the most powerful disassembler : SOURCER. This is the only tool that every programmer wants ,but will not ask for it. Because he doesn't want you to know that it exists .

A Must for a die-hard assembler programmer

Undocumented DOS by Frank van Gilluwe

Addison Wesley No longer available.

While the other book by this guy turns the PC upside down , this book explores DOS .Every nook and cranny of the operating system is explored and many chapters go into uncharted space. All the secrets that nobody wants to reveal are finally exposed in this book.

A Must for a die-hard assembler programmer

The IBM PC / AT technical Manual

IBM press :PC/AT Technical manual

This book contains all the schematics , BIOS listings ports mapping and everything else about the original PC XT and PC AT. It is the SPEC of the PC. Unfortunately it is not easy to obtain. You have to be a certified IBM developer to get this.

Appendix 2 : Datasheet for 8255 controller

Appendix 3 : Win95io users guide

Visual Basic Second Edition

for Electronics Engineering Applications

Vincent Himpe

➤ No Previous programming experience required

➤ Learn How to make Real-World programs in Visual Basic

➤ Build Event Driven - Object Oriented Programs

➤ Communicate using Serial ports, TCP/IP, and DDE

➤ Make custom Objects and Controls

```
Dim MyPSU as new HP6624 ' Create an object of class HP6624
```

➤ Control Hardware using Serial and Parallel Ports

```
Dim MyDVM as new HP34401 ' Create object from HP34401 class
```

➤ Control Test equipment via the GPIB bus

```
Sub Form_load()
```

➤ Vision : a complete GPIB Instrument and I/O control using Classes and Objects Controls

```
MyDVM.address = 22 ' address of the DVM
```

```
MyDVM.CurrentDC ' select Current DC range
```

Full Royalty Free Source in Visual Basic included

➤ How to Optimize programs for Modularity and re-usability

```
Set DVM.Target = MyDVM ' TestBench link to ClassWork
```

```
Set PSU.Target = MyPSU ' TestBench link to ClassWork
```

➤ Building Better Test automation programs

```
End Sub
```

```
Sub Quitprogram
```

➤ CDrom with all Examples, a PDF version of the book and Vision included

```
End sub
```

