Visual Basic Does Lego®

Martin L. Griss, Robert R. Kessler*
Software Technology Laboratory
HPL-95-107
September, 1995

component, graphical programming, instrument control, kit, object-oriented programming, Visual Basic

We have been investigating Microsoft's Visual Basic® and how to use it to develop software for the control of instruments. We have taken the approach of defining a graphical programming environment, similar to that provided by HP's Visual Engineering Environment system, yet with the openness and flexibility provided by a full featured language such as Visual Basic. We have attempted to apply our concepts of component software, software reuse and kits to construct such an environment. This paper describes our experiment. We also discuss the unique instruments we have chosen to demonstrate the system: computer-controlled Legos®.

* *University of Utah,* Salt Lake City, Utah. Robert R. Kessler was on sabbatical at HP Laboratories, 1995.

1. Introduction

We have been involved in system software, tools, and software engineering for over fifteen years, primarily working with Lisp, object-oriented programming, and UNIX systems. Our research at HP Laboratories into software reuse has taken us into investigating PC platform-based software systems for instrument control.[2] We chose Visual Basic (VB) for its natural appeal as a foundation for componentry and software reuse. We decided from the outset to attempt to stay entirely within VB in order to push it to its limits and thus gain an understanding of its strengths and weaknesses. We were pleasantly surprised that using a number of "tricks" and techniques, we could construct our entire system with VB.

Given an instrument domain, we had to choose a specific set of instruments that we could control. We decided to focus on the electronic control of devices constructed with LEGO® parts. These parts offer many of the same types of problems and constraints of standard instruments, yet demonstrate a level of visual expressiveness that helps to cement the subliminal message that "software components should be as easy and as fun to use as LEGO." We are still working to demonstrate the benefits of software reuse in this environment, but offer this paper as a snapshot of our current state and to pass along what we have learned so far.

We note that a secondary goal of our work is to use this technology as the basis for teaching a senior software engineering class emphasizing reuse at the University of Utah beginning in the Fall of 1995. This class will teach students the concepts of software reuse-based software engineering, components, frameworks, and kits. It will also have the side benefit of exposing them to the PC environment and power that is available from rapid program construction using a glue language like Visual Basic and OLE-enabled components. In this paper, we describe our experimental environment and architecture, mention several useful programming idioms that we found to be essential in building the system and conclude with where we would like to see Visual Basic headed in the future.

2. The Experiment

After investigating several possibilities, we decided to choose an "Automated Automobile Safety Inspection and Emissions Testing" scenario as a suitable demonstration of our software reuse ideas. The grand vision would be a test facility with several testing bays, automated robots, and advanced sensoring technology that would provide a standard, controlled, and efficient environment for performing automobile safety and emissions tests. If such a facility were to really exist, it could offer standard tests, and could modify those tests based on the year and type

[2] For more information on software reuse, see "Hybrid domain-specific kits", Martin Griss and Kevin Wentzel, Journal of Systems and Software, to appear September 1995.

of automobile (for example, older autos may have less restricted requirements) and even permit testing for vehicles registered in other states based on the requirements of their home state. The environment could also be reconfigured based on status of robots, number of operators, expected customer demand for that day, etc.

Using this scenario, we constructed a subset of the environment with one car, several robots, and various sensors using parts from a standard LEGO Dacta™ Control Lab kit as shown in Figure 1. The kit includes: output devices which are motors, lights, and horns; input devices including push buttons, temperature, light, and rotation sensors; as well as standard LEGO bricks, wheels, rods, pulleys, etc. In this scenario, a "LEGO-mobile" drives onto the ramp and the system uses the robots to position sensors and manipulate the test environment to generate measurements simulating safety and emissions data.

Figure 1: LEGO car and robots.

The various sensors and motors are connected by wires to a control box with 8 input devices and 8 output devices (the box at the top of Figure 1). The control box connects to a PC via RS232. The parts and the control box are available as a kit, called the LEGO Dacta Control Lab.[3]

The system comes with a DOS software environment based on the LOGO language. The DOS software is a closed, monolithic program designed exclusively for driving the control box using the LOGO language. We, on the other hand, were interested in an open system, controllable from Visual Basic and other languages and tools. So, using descriptions of the communication

[3] Distributed in the USA by: LEGO Dacta, LEGO Systems, Inc., 555 Taylor Road, Enfield, CT 06083-1600, (800)527-8339 and distributed in Canada by: LEGO Dacta, LEGO Canada, Inc., 380 Markland Street, Markham, Ontario L6C 1T6, (800)-387-4387.

protocol and commands as published on the Internet[4], we designed and built a flexible, open, software environment for controlling the LEGO parts entirely from Visual Basic (VB).

3. The Views

As part of the demonstration of the system (which we call Toast[5]), we created three views, operator, process engineer, and programmer. Each is discussed below.

The first is the operator's view of the execution of the system. The idea is that the operator would be responsible for several important interactions during the test and could also monitor the progress via various panels that pop-up to inform and instruct.

Our current Toast test scenario, as seen by the operator and car driver, is as follows:

- The system is initialized and the robot arm with the exhaust sensor is positioned in a known "home" location.

- An auto arrives and is driven onto the rack.

- The type of automobile is identified by operator, and an Access database consulted for car parameters. Figure 2 shows the operator view at the point of identifying the automobile. The operator status panel at the left identifies the current test phase and the time required for each test.

- The robot is positioned to the "tailpipe" location computed from the information retrieved from the database.

- The auto is run at varying speeds and measurements of the "exhaust emissions" are taken. This information is then plotted using Excel, when execution reaches the point of having measured the values, the waveform at the lower right of Figure 2 will show the plot.

- The robot is then moved out of the way and the auto is driven off the rack.

---

[4] Andrew Carol published the original protocols and also wrote some C-based software to drive the system. The original files were found on `earthsea.stanford.edu` in the `pub/lego` directory.

[5] A thin slice of a complete system.

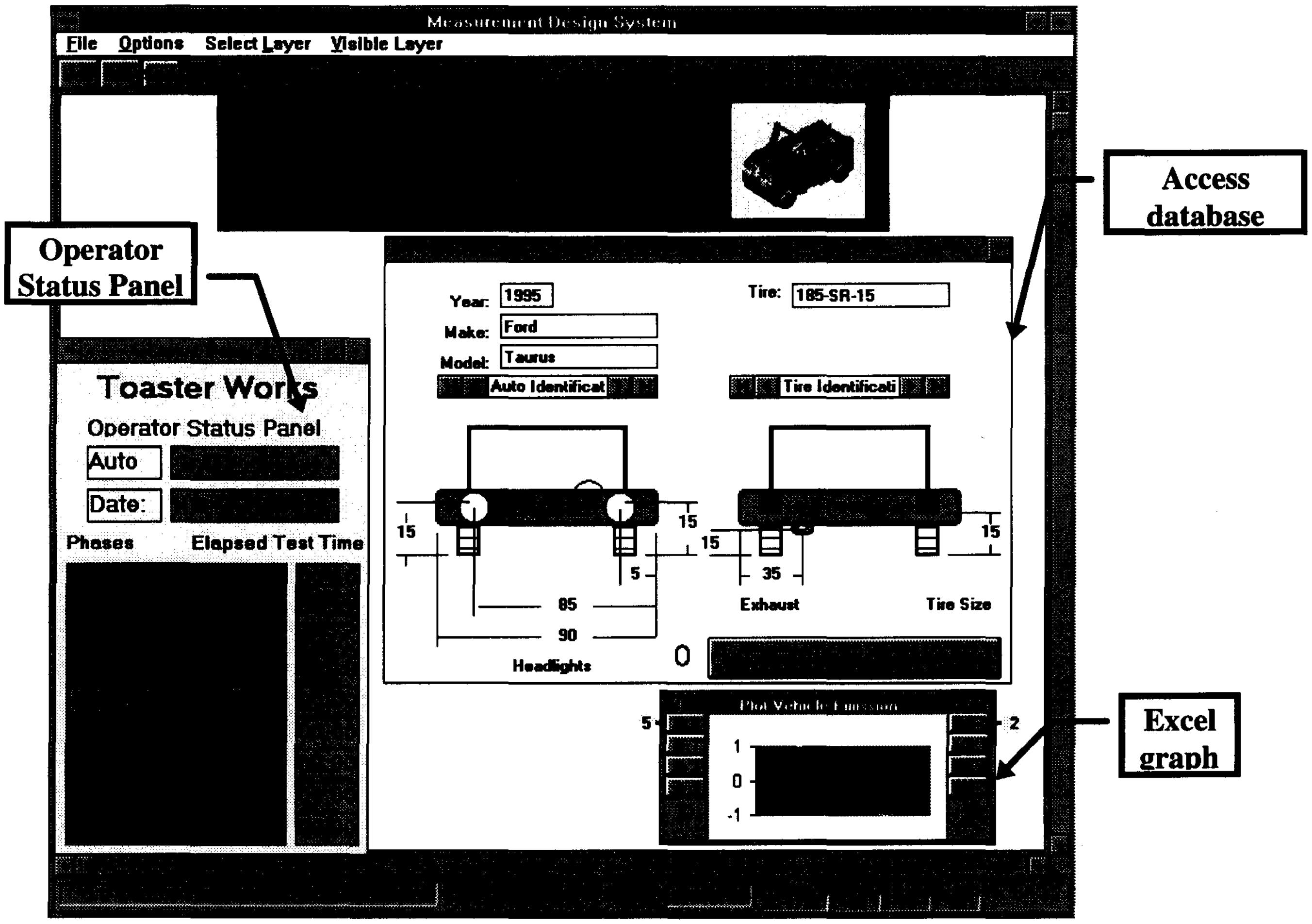


Figure 2: Operator view of Toast.

The second is the process engineer's view. This person typically would be the expert in automotive testing who would have helped to specify and design the processes used to perform the test. This view shows the high-level flow through each test cycle. The process engineer could monitor the execution to verify that the appropriate processes are being followed and watch which of the many branches are taken. This view also allows the process engineer to make simple modifications. Figure 3 illustrates the process engineer view. As each component executes, its associated box is highlighted.

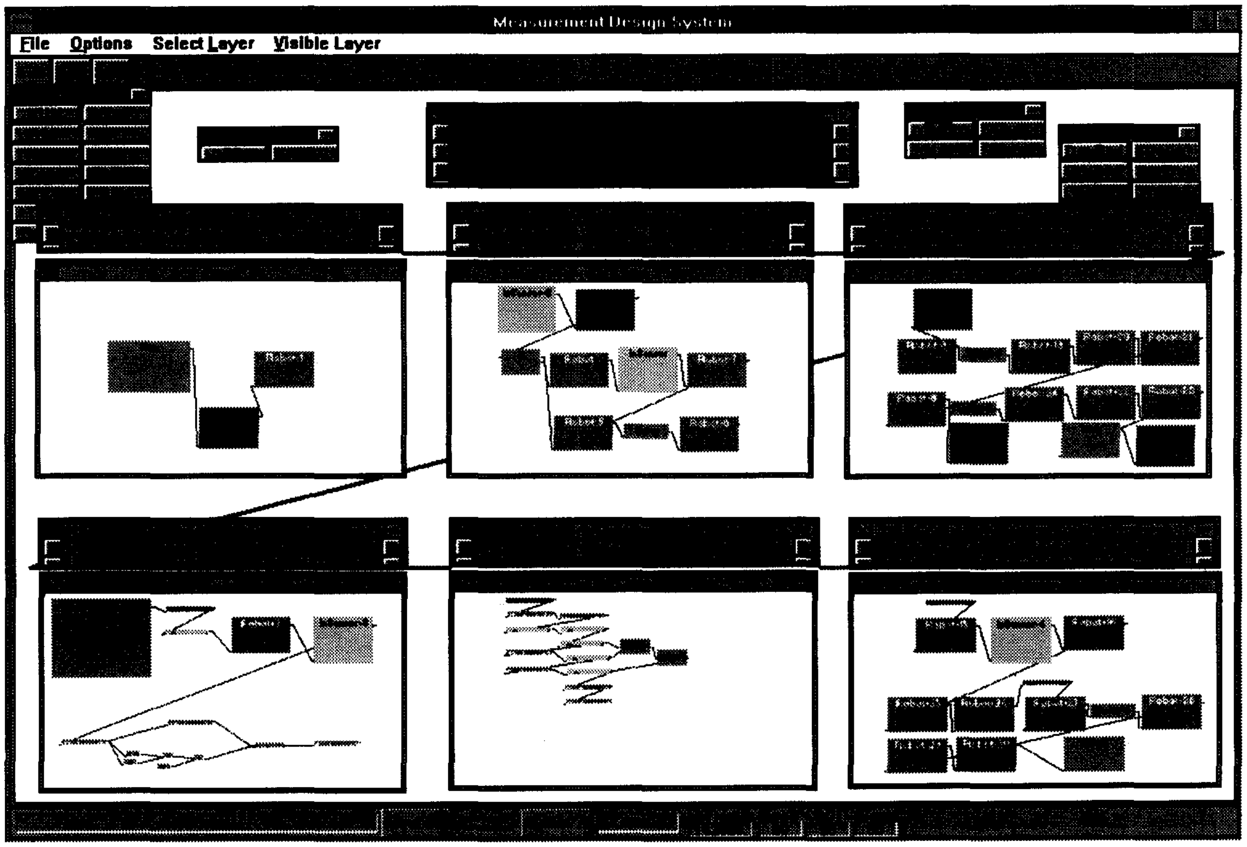


Figure 3: Process engineer view of Toast.

The third is the programmer's view (as illustrated later in Figure 7) which uses a "click and wire" metaphor. The programmer selects components off the palettes and wires them together to indicate data and control flow of the program. In many cases, the process engineer and the programmer are the same person.

Components used in this system would typically be created by a component engineer. In our current system, all of the components are constructed using VB, (either manually or with a simple wizard, see Section 5.7 for a description of our wizard). However, it is easy to see that components created in C/C++ or as prepackaged VBXs would be quite possible and easy to integrate into the system.

4. The Architecture

With a vision of what the system is capable of doing, we can dive deeper into understanding how it goes about doing this. The architecture is illustrated in Figure 4. Each oval represents a separate Visual Basic (VB) process in this client/server architecture.

- Wave - the visual programming environment that is used to write programs to control the LEGOs (upon which Toast is built).

- Auto Control - a VB program that is a LEGOmobile "front panel" that allows a driver to manually operate the auto (e.g. placing the car in gear, turning on the lights, pressing the accelerator).

- Box Server - another VB program that is the communication linkage between the higher-level programs (like Wave and Auto Control) and the physical LEGO Dacta hardware.

Since the Wave and Auto Control client processes and the Box Server are all written in VB we had to utilize DDE for their communication. The current version uses straight DDE, with extension to Network DDE planned for the next version. This would allow control of the LEGO system from remote processes. If future versions of VB supported OLE as both a client and a server, an obvious extension would be to use OLE for communication. Wave communicates with EXCEL via OLE automation and with the Access database via data-aware controls. The Box Server communicates with the Control Lab over a 9600 baud RS-232 serial line. It sends multi-byte commands to control the output devices and receives and processes 19 byte frames of sensor data every 50th of a second. The Box Server must also send a "keep alive" byte at least every two seconds.

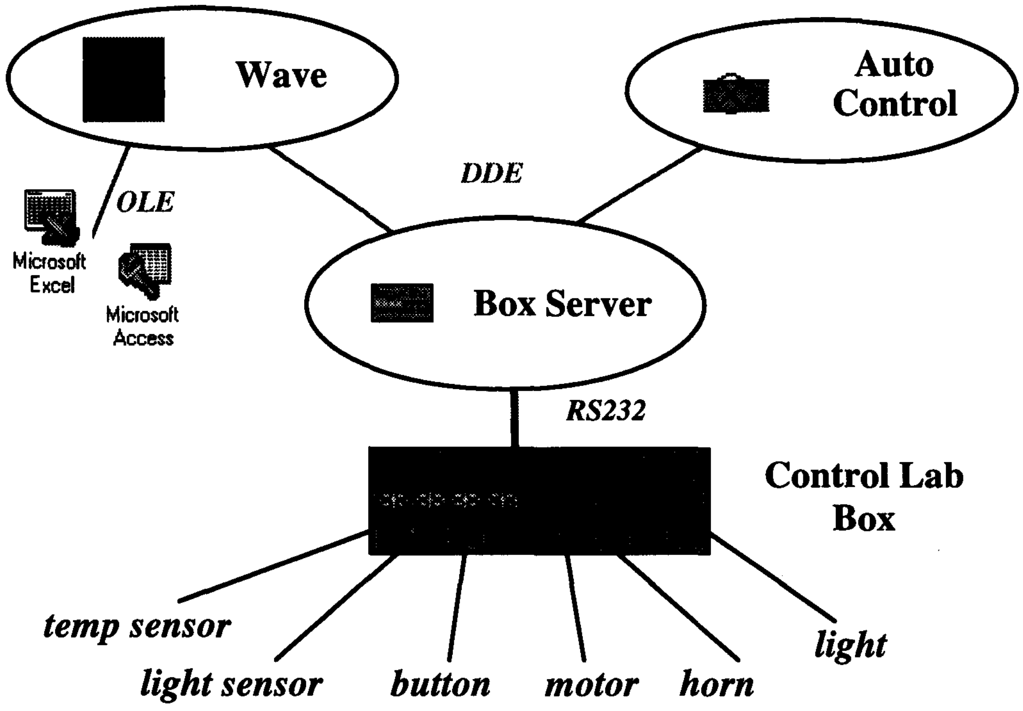


Figure 4: The overall system architecture.

Note, each instance of a Box Server program talks to one COM port on the PC. Thus, if a single computer system were to be attached to several LEGO Dacta Control Labs, a new Box Server process would be running for each COM port. The Control Lab box permits connection of up to 16 devices: 8 output for driving motors, lamps, and beepers; and 8 input for sensing temperature, button pushes, light, and angle.

4.1. The Box Server Process

The box server process has a fast loop that monitors and controls the physical hardware. It accepts a set of higher level commands from other processes, and includes a panel of its own to show which commands are being executed and what results are being returned to the clients. These commands include things like: "watch until the temperature sensor connected on device 3 exceeds 75 degrees, then signal me".

The Box Server process is a carefully coded VB program that attempts to perform real-time operations in a non-real-time operating system. This has forced the development of software that must still be robust in the face of not being able to achieve real-time deadlines. In the future, we envision recoding this in C for higher performance, but we wrote the initial version in Visual Basic for flexibility and rapid development. Interestingly, the current version is fast enough for our experiments.

The main operations of the Box Server process are to:

- Initialize the communication with the hardware, making sure to reinitialize the interface if it recognizes that the hardware is no longer communicating due to loss of the keep-alive command.[6]

- Manage a request/acknowledge DDE-based protocol between itself and the client programs.

- Interpret high-level output device commands and send several low-level commands to perform the requested operation. For example, a high-level command might issue the command "turn on the motor connected to device 3 at speed 5 in the clockwise direction." Three low-level commands would be generated to set the speed of the device 3 to 5, set the direction of device 3 to clockwise, and then finally turn on device 3.

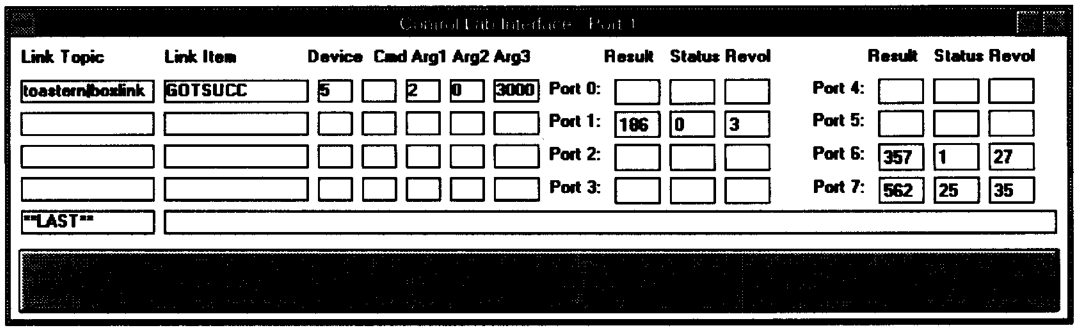
- Return the current sensor values for a particular sensor type. For example, "what is the state of the push button on device 5?"

- Monitor a particular sensor to see if it exceeds, meets, or falls below a particular value. For example, "watch the temperature sensor on device 7 and tell me when it exceeds 75 degrees Fahrenheit."

- Monitor a particular sensor and return all sensor values for some period of time. For example, "for the next five seconds, gather and return all sensor data for the light sensor attached to device 4."

- Send a keep-alive byte as often as necessary.

[6] The only successful method that we have found is to have the inner time-out loop keep a count of the number of successive times that no input frame is available. When the count exceeds a heuristically determined threshold, the program reinitializes the interface. We considered using a separate timer/time-out process, but that was additional overhead that we wanted to avoid in the main inner loop.



Control Lab Interface - Port 1

Link Topic	Link Item	Device	Cmd	Arg1	Arg2	Arg3		Result	Status	Revol		Result	Status	Revol
toastern|boxlink	GOTSUCC	5		2	0	3000	Port 0:				Port 4:			
							Port 1:	186	0	3	Port 5:			
							Port 2:				Port 6:	357	1	27
							Port 3:				Port 7:	562	25	35
LAST														

Figure 5: The Box Server panel view.

Figure 5 shows the panel view of the Box Server. Normally, the form is minimized, but having a view of the inner workings of the server is a helpful debugging aid. There are three main parts. The first, in the left two columns, is used to show the link from the client, with each row representing one client interaction (thus, as you can see, each Box Server can handle up to four clients). The second part is the middle five columns, which are the commands being issued by the client, while the third part is the six columns on the right, that are used to monitor the sensor values of the eight input devices on the Control Lab box. For each input, the right columns show the **result** and **status** values returned directly from the sensor frame and **revol,** which is a computed count of the number of angle sensor revolutions or the count of the number of button sensor pushes. A hidden listbox also keeps track of the commands received for each client. Finally, the last element is the large "STOP ALL DEVICES" button, that sends the command sequence that turns off all output devices. This is particularly useful if the robot is about to knock the car off the rack or the car is attempting to fall onto the floor. ☺

A full description of the Box Server and its clients is contained in Appendix A. It focuses on the low-level details required to implement the client/server interface between each of the processes.

4.2. *Auto Control*

Auto Control is a simple, two form, client process, hand coded in VB. The purpose of the program is to allow the automobile to be manually controlled. This allows the "driver" to manually drive the car (controlling the accelerator speed and forward/reverse direction), turn on the horn and set the lights to various levels. As shown in Figure 6, we used sliders and radio buttons to create an appropriate interface. Note, the controls attempt to be as realistic as possible. For example, you can't start the car unless it is in park and you can't switch from reverse to forward without going through neutral. An additional form allows configuration of the COM port and specification of the Control Lab output device locations for the engine motor, headlights, and horn devices.

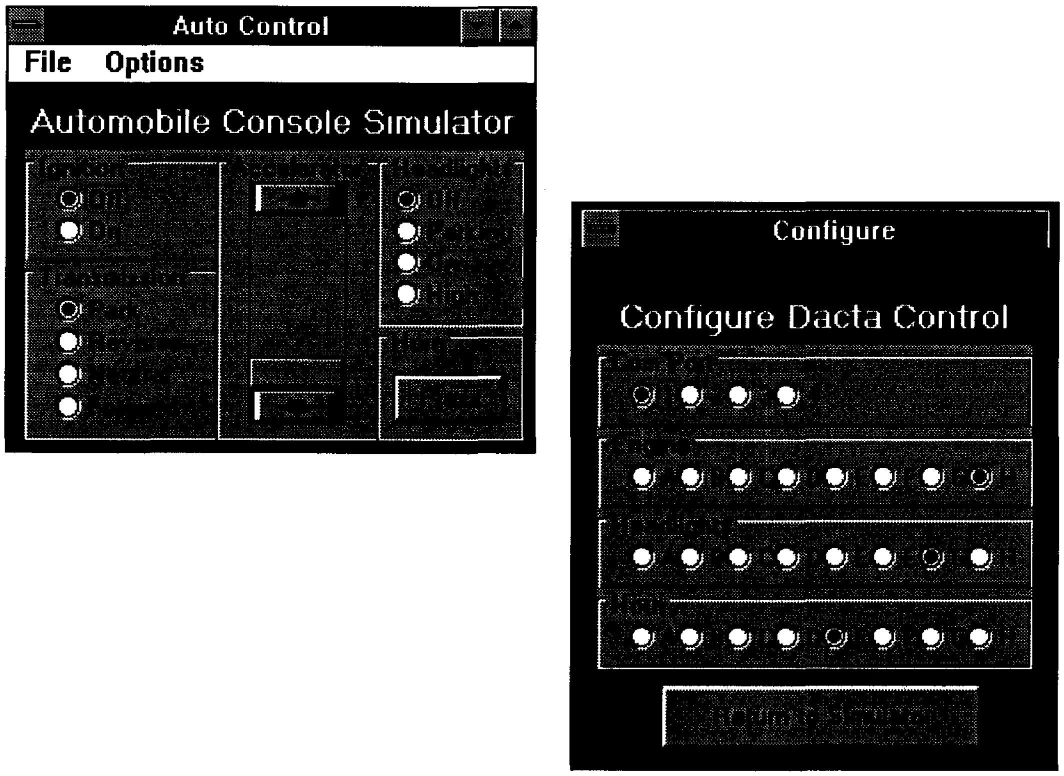


Figure 6: The Auto Control main and configuration forms.

4.3. Wave

The main client process is the test center control, which was designed using a graphical programming environment we call Wave, "Wired Assembly Visual Environment." Wave, written in Visual Basic, allows the construction of "click and wire" programs that interacts with the user, the robots, motors and sensors. Figure 7 shows a Wave panel that defines the instructions to rotate the exhaust sensor robot to its home position with the exhaust arm fully extended.

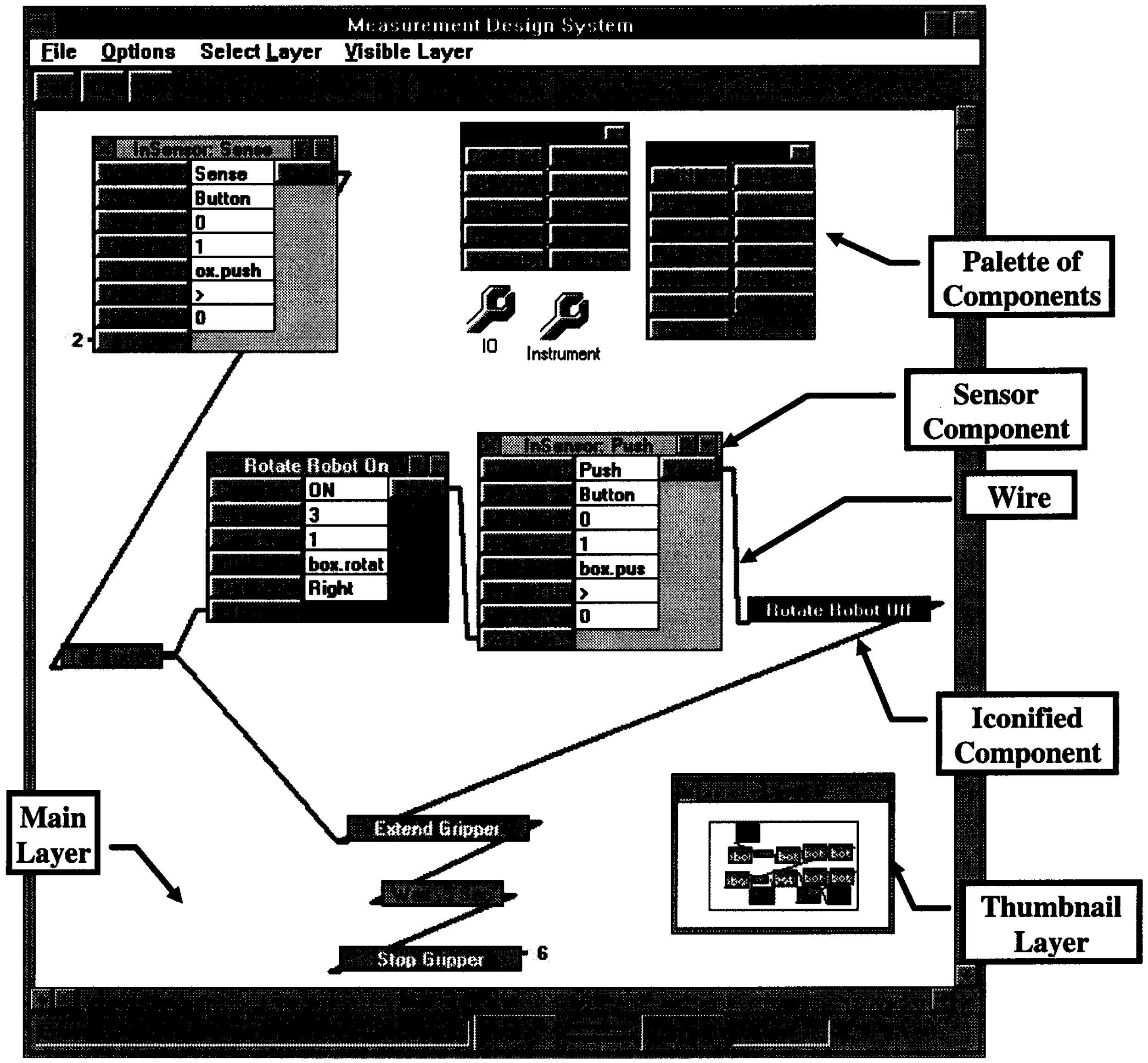
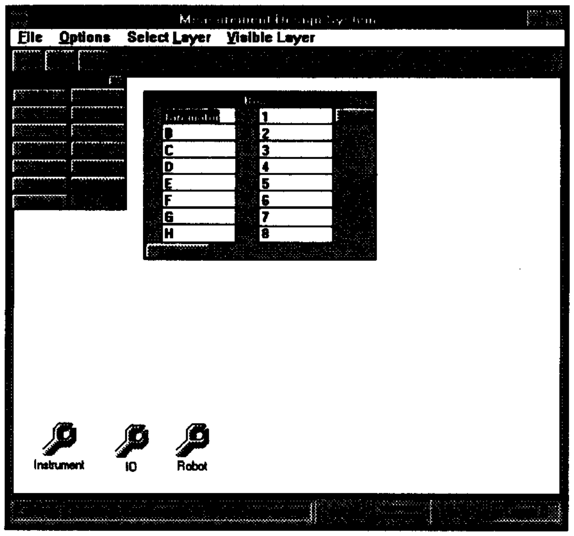
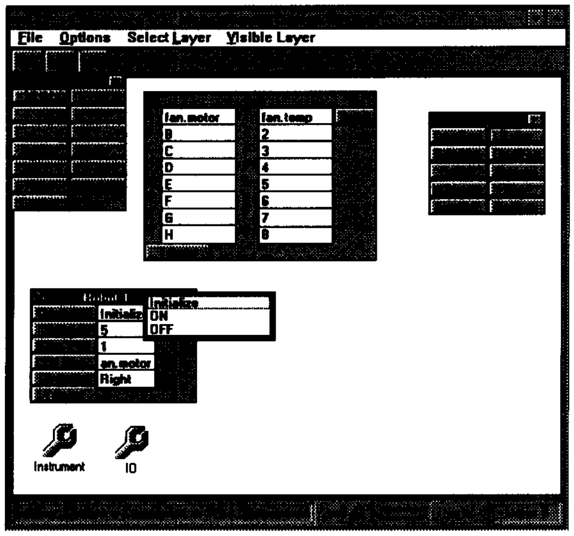


Figure 7: Typical Wave panel.

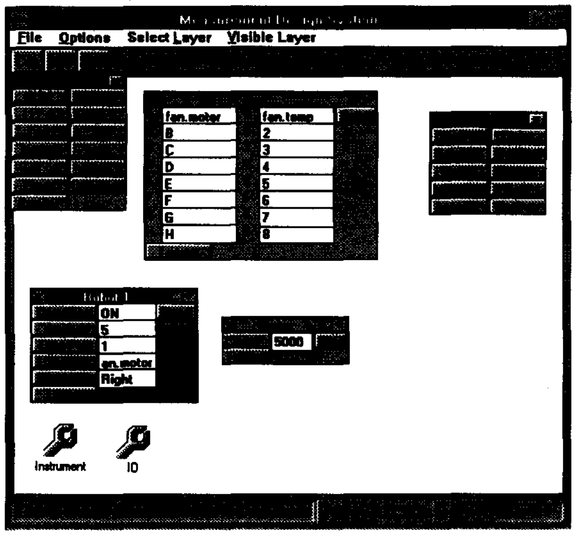
The callouts in Figure 7 highlight several of the important features of the Wave panel. The first is the palette of components. Wave has several palettes, typically grouped to have common functionality (like the components that interact with the Control Lab). When the user clicks the palette button associated with a component, an instance of that component is created and placed on the main layer. The component may then be manipulated via editing, dragging, and iconification. Each component usually has input fields, input ports, and output ports. The input fields contain local state (like the speed of a motor) that may be set by filling in a specific value or by selecting it from a menu of choices. The input fields may also be wired into the computation so they receive their values dynamically. Input ports accept wired connection of values, perform the component's operation, and then send the results to the appropriate set of output ports. Wires enable data and control flow between components and are permitted to have arbitrary fan-in or fan-out. Loops are supported by the programmer specifying a path into the loop flow that is not in the circular part of the loop (i.e. the loop head is triggered by some external data and subsequently triggered by the loop end). We support multiple layers as a means of handling hierarchy and managing complexity of the diagram. Any component can appear on any layer, with the wired communication between components starting on one layer and finishing on a different one. We use thumbnails to show other related layers. These thumbnails are resizable and may be visible on any layer, depending on what makes the most sense in terms of clarity of the program fragment. The basic concept of our layers is to group a part of the computation together into a logical phase of the program operation (a layer is similar to an integrated circuit layer without the restriction of only vertically passing data between the layers - the communication between layers can take any form).



(a)



(b)

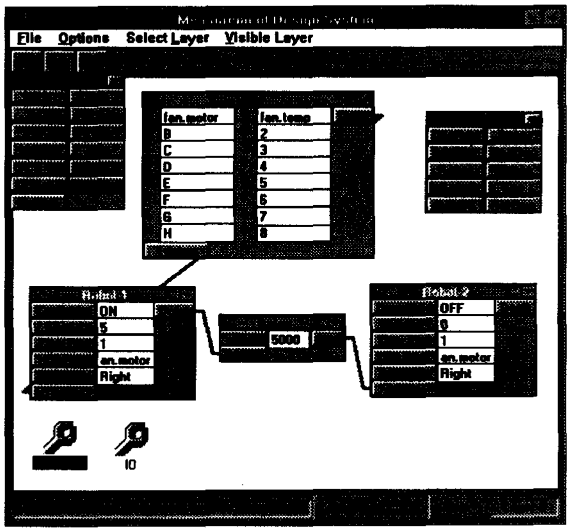


fan.motor
B
C
D
E
F
G
H

fan.temp
2
3
4
5
6
7
8

ON
5
1
an.motor
Right

5000



fan.motor
B
C
D
E
F
G
H

fan.temp
2
3
4
5
6
7
8

ON
5
1
an.motor
Right

5000

OFF
0
1
an.motor
Right

(c) (d)

Figure 8: Designing a fan control using Wave.

Figure 8 shows an example of designing a simple fan control program with Wave. It turns the fan motor on for 5 seconds. Upon startup, the user is given an empty layer and several palettes of components. In Figure 8a, we first select the box component which allows us to assign names to the Control Lab devices. Once the names are assigned, they can then be accessed using the object-oriented or VB style of prefixing the name with "`box.`" as "`box.fan_motor`" in this example. In Figure 8b, we have selected a "robot" control component which allows us to send commands to an output device and are currently choosing the type of command from the pop-up menu. Notice that several of the input fields have been filled in with values, including the control device ("`box.fan`") which references the fan motor device connection as defined in the box component. Figure 8c shows adding a delay component and setting the delay time to 5000 milliseconds (5 seconds). Finally, Figure 8d has the all of the components including wires. At this point, pressing the **Run** button would allow the program to execute. The first robot component opens a connection to the Box Server, and sends it the "turn on" command. Once that command has been issued, the program waits five seconds and then sends a "stop" command. Without anything else to do, the program will then stop. During the execution, each component's top menu bar will highlight while that component is executing.

5. The Design Of Wave

We were inspired in our work by trying to combine the openness and ease of programming in the Visual Basic/PC environment, with the intuitive click-and-paste-and-wire interface that we saw in other instrument control systems such as Hewlett-Packard's VEE or National Instrument's LabView. We also got useful ideas about palette management and drawing manipulation from programs like Shapeware's Visio. In this section, we will describe and show some of the more interesting implementation and design decisions that we made in Wave. From the beginning, we knew that we wanted to use the click-and-wire model to create an environment that was very conducive to software reuse. That is, the components should be flexible and easy to design and integrate into the system. Another given was that coming from object-oriented programming backgrounds, we wanted to "think" object-oriented as much as possible (naturally within the context of the event-driven model of VB).

Our first key decision was what exactly should a component be? We toyed with several ideas ranging from a simple graphical box, to a control, to a full-fledged form. Using a form was very appealing because one could design a form as a "class" and then clone that form via VB's `new` expression. Plus, a form seemed very "object-oriented" in that one could have private data, and private operations (methods in Lisp or Smalltalk terminology or member functions in C++). So, we settled on a form being our basic component. When the user clicks on a palette button, a "`new`" instance of the associated form is created and displayed for the user. Another advantage of this approach is that it fits naturally with a "file-based" reuse strategy in that each separate form file represents a distinct component. In addition, by saving each form file in text format, we were able to create a wizard that allows one to easily generate new components (see Section 5.7 for a description of the wizard).

Naturally, once one makes a fundamental implementation decision like our equating components with objects, we were faced with several challenges. These and other aspects of the Wave design are described in the following sections.

5.1. Calling Procedures Defined in Each Component

In the most basic part of the system, we need the ability for one component to call procedures defined in another component. For example, when a component computes a value, it must pass that value out its output port and on to the connected input port. Passing data through an input field was easy since the control representing that field is visible outside of the component. However, passing an array of values into an input port is problematic. How does the source component provide the destination component with those values? Once it passes the values along, how does the destination component start to execute its internal computation? There are many other cases where we need the ability to execute some code associated with a particular component.

One possible solution is to try and make everything global. The procedures for propagating values could be global (with a special name to indicate that it is associated with a particular form). Likewise, internal data could be made global. This is much more difficult, since each instance of a form must have its own internal set of data. So, code would have to be defined to manage arrays of internal data structures for each instance of a form. This is complicated by the fact that the size of the array of data coming in an input port is not restricted. For example, the + component can add individual numbers or vectors of numbers of any size. Since VB does not permit redim'able arrays within types or arrays of redim'able arrays, the implementation is extremely difficult.

Our solution was to develop a general purpose mechanism for calling local procedures within a form. The external interface is through the dispatch global procedure.

```
Option Explicit

' These define the operations that we can do to any component.
Global Const METHOD_Print_loader = 0
Global Const METHOD_Set_values = 1
Global Const METHOD_start_form = 2
Global Const METHOD_delete_output_links = 3
Global Const METHOD_delete_input_links = 4
Global Const METHOD_retrieve_properties = 5
Global Const METHOD_delete_one_link = 6
Global Const METHOD_new_colors = 7
Global Const METHOD_compute = 8

Sub dispatch (f As Form, routine As Integer)
  ' Call the routine indicated by the routine number for component f.
   f.Dispatcher.Tag = routine
   f.Dispatcher = True
End Sub
```

Each component includes a command button control called dispatcher. Typical code for a dispatcher follows (this comes from the sensor component):

```
Sub Dispatcher_Click ()
    Dim method_num As Integer
    method_num = dispatcher.Tag
    Select Case method_num
    Case METHOD_compute
        compute
    Case METHOD_new_colors
        disp.BackColor = new_backcolor
        disp.ForeColor = new_forecolor
    Case Else
        dispatched Me, method_num, connections, my_controls(), link_out(), link_in(), props()
    End Select
End Sub
```

If the component wishes to handle any operations itself, those are first selected and then performed. In this case, it performs the compute and new_colors operations. For any operations that are to be handled generically, dispatched is called passing all relevant internal data structures. Thus, adding a generic method to any component is easy; we simply assign it an index and then place a call to the appropriate code into dispatched as follows.

```
Sub dispatched (f As Form, routine As Integer, connections As ListBox, my() As Control,
link_out() As Control, link_in() As Control, props())
  ' This subroutine is responsible for calling the actual code.
  Select Case routine
   Case METHOD_Print loader
      loader_dump_parm f, my()
   Case METHOD_Set_values
      loader_read_parm f, my()
   Case METHOD_start_form ' Attempt to Start Control
      start_form f, link_in()
   Case METHOD_delete_output_links ' delete all output links to form in arg one
      delete_link form_to_delete, connections, link_in(), link_out(), 0
   Case METHOD_delete_input_links ' delete all input links to form in arg one
      delete_link form_to_delete, connections, link_out(), link_in(), 1
   Case METHOD_retrieve_properties
      get_props_aux f, props()
   Case METHOD_delete_one_link ' Delete an output link to a control in arg one
      delete_one_link out_ctl_to_delete, in_ctl_to_delete, connections, link_out(), link_in()
   Case METHOD_new_colors ' Color anything internal to match the parent.
      ' new_backcolor and new_forecolor have the arguments.
      ' For now, we don't need to do anything here.
   Case METHOD_compute
      ' Compute is handled by each component itself.
   End Select
End Sub
```

5.2. *Access to Component-Local Data*

The problem of accessing component-local data was solved with the same `dispatch` mechanism as described in the previous section. Global procedures can have access to local data by passing that data into the global procedure as shown previously in `Dispatcher_click`. As an example, the following code illustrates a call to `dispatch` for when the user decides to delete a component. What this code does is scan through its list of all output connections, find the form that it is attached to and tells that form to delete any input connections that it might have to the form that is being deleted (in this case, `f`). All of our connections are doubly-linked (the source knows about the destination and the destination knows about the source), but the data structures that keep this linkage information are maintained inside of each component. So, it is easy to write a generic procedure to do the deletion, but it must be called with the right data. `Dispatch` allows us to do this without adding any new code to each component. We simply create a new operation number and make sure that we pass the linkage data to the global procedure. As you might guess, the main disadvantage of this model is that if some other internal data structure must be made accessible to generic routines, then all calls, across all components must have their calls to `dispatched` updated to take new arguments.

```
    ...
    For i = 1 To UBound(link_out)
        If link_out(i).Parent Is f Then
            form_to_delete = f.Tag
            dispatch link_in(i).Parent, METHOD_delete_input_links
        End If
    Next I
    ...
```

Note, this illustrates one other disadvantage of this model: because we cannot pass any arguments to a control (other than what we might package up in the tag field), we must resort to using global variables as temporary holders of the arguments.

5.3. *Execution Model*

One important decision was how to actually handle the wired execution model. Our solution uses a combination of the `dispatch` mechanism and a timer with a stack. Each component has a local procedure called compute, which performs the computation that is defined for the component. Dispatch is used to call the component and tell it to compute. When the computation is completed, the component takes the results and propagates them to its output components. Then it pushes the output components onto a global stack of runnable components. A timer in the top-level form wakes up and checks to see if there are any runnable components. If the stack is empty, it goes back to sleep. If there are components, then it invokes dispatch with compute for that component and the cycle continues until there is nothing more to execute. The following code is the loop within the stack_timer for handling the wired computation model.

```
Sub stack_timer_Timer ()
    Dim f As form
    While stack_has_elements()
        If Not stack_timer.Enabled Or mode = mode_editing Then Exit Sub
        If map_focus >= 0 Then map_deactivate frm_save(map_focus)
        Set f = stack(pop())
        map_activate f ' Highlight the form that is about to compute.
        map_focus = f.Tag ' Globally remember which component is executing.
        update_space ' Print GDI/USR information on status bar.
        dispatch f, METHOD_compute
    Wend
End Sub
```

5.4. Wires, Hierarchy, Forms, and the MDI

From the beginning, we chose to implement Wave as an MDI-based application. The reasoning was that components as forms would exist in the context of the parent MDI program (Wave). Windows would then manage much of the headache for us in terms of allowing forms to grow outside of the boundaries of the parent form (by automatically adding scroll bars) and properly maintaining the correct physical relationship between the components. This became more challenging once we wanted to add wires. Given the standard VB documentation, we found out that it is impossible to draw on the MDI form itself.[7] So, our plan was to create a form that would contain the wires and position itself behind all of the component forms. It was tricky coding, but we finally got it working in concert with the MDI form to allow us to show the wires connecting components.

This worked fine until we decided to add hierarchy. Suddenly, we needed the ability to have some forms be included on the current layer, while others needed to be hidden on their own layer. MDI does not allow child forms to be hidden, so we had to experiment with several other methods (like moving them off the visible screen - which didn't work because the MDI parent would create a huge scrollable area that eventually allowed you to get to the other forms). Our current solution was to carefully control the Zorder of the forms and keep all forms that are not

[7] We recently learned it is indeed possible to draw on the MDI form by using Windows API calls. We discovered this in a package called mdidemo that we found on the Internet. This same package showed that it was possible to solve another problem we had, which was the ability to hide MDI child forms.

on the current layer behind the wiring form. So, the wiring form looks like it is the background of the screen, but it is actually hiding those forms that are on other layers. The thumbnail forms that you have seen in the various Wave screen dumps are actually the layer's wiring form, scaled, with its associated forms drawn as boxes. This technique works just fine, but was very tricky code to make sure that the forms on the other layers did not "peek" out at inappropriate times. Again, as mentioned in [7] using the Windows API allows you to hide an MDI form.

One thing that we learned from this is that if you are going to write anything more than simple data entry forms, then you need to make sure that you understand the Windows API and how to use it. The best book that we have seen on this is Daniel Appleman's "Visual Basic Programmer's Guide to the Windows API", 1993, Ziff-Davis Press.

5.5. OLE Automation

We used OLE automation to allow us to interact with Excel. We created a chart component that takes up to four input vectors and plots their values. The component has an embedded chart, with support code to interact with the chart by providing it the data. The providing of data was a perfect case for OLE automation. Plus there are added benefits. Using an Excel-based component, the data could be automatically graphed as a chart, and then since the chart is Excel, the user can use the built-in Excel functionality to modify the look and feel of the chart to suit his or her needs.

The chart component was created by selecting the OLE 2.0 control and specifying that we wanted an embedded Excel object. Still at component design time, we created a sample set of data in the spreadsheet, created a graph of that data, and finally set the view of the OLE object to be the chart. Having created that infrastructure, the code would then take each element out of each array, copy it into the embedded spreadsheet, and then let Excel produce the graph. This was fine in theory, but proved to be too costly in practice. It was just too slow to copy all those array elements over via OLE automation, one element at a time. The overhead of doing this for large vectors was prohibitive. Our solution was to do bulk copying via the clipboard. The OLE interface code takes each vector and writes a copy to the clipboard.[8] Then, via OLE automation, we get Excel to paste that data directly into a spreadsheet column. This improved the speed immensely, making it very usable. The attached code is the `compute` portion of the chart component and the `paste_array` subroutine which copies a vector into the clipboard. In order to illustrate the salient parts of this code, we have eliminated the code that handled multiple input vectors.

```
Sub compute ()
    If mode <> mode_running Then Exit Sub
    If inp_init(0) = 1 Then
        ' ole1 is the control.  inp0 is the first of the input vectors.
```

[8] One could also use a file.

```
        Dim excel_chart As Object
        Dim app As Object ' The application.
        Dim wb As Object ' The workbook.
        Dim sheet As Object ' The actual worksheet.
        Dim max_ubound As Integer ' Size of largest input vector.
        Dim plot_count As Integer ' Keeps track of which plot this is.

        ole1.Action = 7 ' Activate the embedded chart.
        ' Get a handle on the sheet by wandering the OLE object hierarchy.
        Set excel_chart = ole1.Object
        Set app = excel_chart.application
        Set wb = app.workbooks(1)
        Set sheet = wb.sheets(2)

        ' Determine the size of the largest input vector.
        plot_count = 1
        If inp_init(0) = 1 Then
            max_ubound = max(max_ubound, UBound(inp0))
            plot_count = plot_count + 1
        End If
        ' *** Similar code deleted to handle other input vectors.

        ' Select a range in the sheet.
          excel_chart.chartwizard sheet.range(sheet.cells(1, 1), sheet.cells(max_ubound + 1,
plot_count))

        ' Paste the input vector into the sheet.
        plot_count = 1
        If inp_init(0) = 1 Then
            paste_array inp0(), max_ubound, plot_count, sheet
            plot_count = plot_count + 1
        End If
        ' *** Similar code deleted to handle other input vectors.

        ' Propagate the values.
        If inp_init(0) = 1 Then
            propagate_array inp0(), 0, link_in(), link_out(), Me
        End if
        ' *** Similar code deleted to handle other input vectors.
    End If
End Sub

Sub paste_array (inp(), max_ubound, column, sheet As Object)
  ' Paste the first argument into the clipboard.  Note, we always paste
  ' max_ubound elements, so if the vector is too small, we paste extra zeros.
    Dim tc, i As Integer
    tc = Chr$(10)
    Dim clip As String
    clip = ""
    For i = 0 To UBound(inp)
        clip = clip & inp(i) & tc
    Next i
    For i = UBound(inp) + 1 To max_ubound
        clip = clip & " " & tc
    Next i
    ' sheet.cells(i + 1, 1).value = inp0(i)
    clipboard.SetText clip
    sheet.paste sheet.range(sheet.cells(1, column), sheet.cells(max_ubound + 1, column))
End Sub
```

We made one other change. When running our sample application, we noticed that the first time that we activated the chart object, it took about 25 seconds. This happened during the computation and resulted in an annoying pause while we waited for the data to be plotted. We solved this problem by performing the activate at program load time. Right after loading the chart object, we simply execute its `ole1.Action = 7` line, thus allowing that pause to happen only at load time instead of during the potentially time-critical program execution time.

5.6. Windows NT

Part way through our development, we switched from using Windows for Workgroups (WFWG) to Windows NT. This was one of the most important changes that we did. The robustness of NT and its ability to withstand crashes in VB or the VB program was essential. Not one time did we lose any work due to a crash in the program under NT. The OLE automation also seemed much more robust under NT. Often under WFWG, the Excel application would "hang around," eventually consuming resources and either crashing the system or forcing a reboot. This rarely happened under NT. Finally, NT allowed us to handle larger programs due to its smaller consumption of GDI and USR resources. This does have a downside. We were able to write programs under NT that would not run under Windows because they consumed too many resources and Windows starts out with a much larger percentage of the resources used by the operating system.

5.7. The Wizard

As we developed more components, it became clear that we needed a way to assist the process. Most of the components are very similar. They have zero or more input ports which receive a vector of data, zero or more input fields, and one or more output ports. They all have similar menu bars for moving the forms around, buttons to control deleting and minimizing the component, etc. If we had an inheritance mechanism, we could have constructed a parent class that specifies all of the common features and operational characteristics. Each individual component would then specify those parts that were different (how it performed its computation, what specific input fields did it need, etc.). We solved this problem by creating a component wizard. Using a structure similar to the wizards popularized in the various Microsoft Office tools, the user fills in several screens specifying the necessary characteristics of the component. On the last screen, they write the VB code that defines what the component does.

The wizard works from a template of a generic component (an original .FRM file modified with template markers). It processes the template file to drive the creation of the various "fill-in-the-field" wizard pages. When the component designer is finished, the wizard takes the new data and the template file and generates a .FRM file for the new component. This can then be added directly to the Wave program project using the **File->Add File...** command and then used to generate a new Wave executable. The ability to save .FRM files as text was an essential enabling technology to make the wizard possible. The wizard was written using a small toolkit which makes it easy to write similar wizards that read INI and template files, process parameters and create one or more output files.

6. What We Learned

We have completed our initial prototype and are preparing to move into the next phase. We are now focusing more on the educational aspects of the system and how best to use it as the basis of the senior Software Engineering Laboratory sequence in the Department of Computer Science at the University of Utah. In parallel, we are investigating object-oriented design techniques and how to adapt them for reuse and also how they could be applied to an environment like Visual Basic and Wave.

6.1. Observations

Now we end with a list of some of our observations:

- Visual Basic is a very nice environment. It has its quirks and warts, but on the whole, it was quite pleasant. Our previously limited exposure to event-based programming meant that often our initial ideas on how to solve a problem were incorrect, so we had to retrain our development processes to always consider events and how to appropriately apply them.

- Configuration management was a real pain. We resorted to using the RCS - Revision Control System on a network connected UNIX system. This at least gave us the ability to keep track of entire versions of the system, but did little more. We are currently testing a new product called "SourceWorks/VB" from ViewPoint Technologies, which so far seems to have a very nice set of features.

- The inability to have structured access to form-local procedures and data forced us to build mechanisms to provide the same functionality. If Visual Basic allowed you to distinguish between `private` and `visible` form procedures, there would be a significant improvement in how one writes and structures VB programs.

- Inconsistencies in the language were a continuing source of irritation. For example, you `new` a form to create a new instance, but you `load` a control to create a new control. Other examples include: a function is restricted to only return one of the seven basic data types; redimensionable arrays and object types are not permitted inside structured data types.

- "Form as object" still seems like a good design decision. However, we are currently running into memory limits due to the large number of forms present. We are currently investigating lighter-weight options. Some of those include: reduce the cost of each component to easily permit hundreds of the forms to be present; optimize so only those forms on the current layer actually exist, the others are handled as "objects" with state and associated procedures; make the "face" of each form be a picture or image, creating the input controls at runtime (have one version of each kind of form, with the other instances being just a picture); or eliminate the form as component mapping and just use a scaleable box as the component (again, creating the input controls dynamically).

- As previously mentioned, OLE automation was slow when passing one element at a time to Excel. We did some initial experiments with the 32 bit version of Excel and it was ***much*** faster. So, this concern may be mitigated by the latest 32 bit offerings from Microsoft.

- We would prefer to use the OLE/COM procedure call model throughout the system and look forward to future VB offerings that will make that possible.

- Finally, we would like the ability to have the Wave design environment be part of the VB design environment (eliminating the two step process required to add a new component: 1] add the component to the Wave project; 2] create the Wave executable to simply adding the new component to the current VB design environment).

As VB becomes more like the Visual Basic for Applications (VBA) language that is used as the extension language in the various Microsoft word processing and spreadsheet products, we are certain that most of these "wish list" items will actually become part of VB. That will help to make VB programmers even more productive and allow them to write code that is even easier to maintain.

7. Conclusion

An important observation of our work has been that working with "real-world/analog" devices is very different and very challenging. In the "non-real world" that many of us who write software see, these issues don't ever come up. But when trying to make sure that a real time device is handled properly, that sometimes does not send over the correct data, it is indeed very challenging. Next on our agenda is to investigate the features of Borland's Delphi and Visual Basic 4.0 to see how the benefits of objects can be combined with component-oriented software.[9]

Wave needs more tuning for performance, or conversion of some parts into C/C++. We wish Visual Basic was more Object-Oriented so mechanisms like our dispatch code was automatically handled, and we had a reasonable inheritance mechanism to help sharing of common parts of components. We do not need a full object-oriented language, but moving in that direction would be a significant improvement.

[9] For more extensive discussion, see Deborah Kurata's, "Architecting Solid Software," VBPJ, April 1995 and Jon Udel's, "Component Software," Byte, May 1994.

Finally, we still have considerable work left on the software reuse issues. We believe that we have created a foundation that fits our concepts of a kit, but have not yet demonstrated any reuse by retargetting the system for a slightly different domain. This is left for our next version.

8. Acknowledgments

The authors are grateful to the HP staff members who reviewed earlier drafts of this report, including Randy Coverstone, Regan Fuller, Jon Gustafson, and Marc Tischler. We want to especially thank Keith Moore for his comments, as they helped to crystallize many of the fuzzy parts and also point out areas that should be seriously investigated for improvement in future versions of the software.

9. Biographies

Dr. Martin L. Griss is a senior Laboratory Scientist at Hewlett-Packard Laboratories, Palo Alto, where he researches object-oriented reuse and measurement system kits. As HP's *reuse rabbi*, he led work on software reuse process, tools, software factories and the systematic introduction of software reuse into HP's divisions. He is an Adjunct Professor at the University of Utah working with Professor Kessler on reuse-based software engineering education. He was previously director of the Software Technology Laboratory and has over 20 years in software engineering research. He has authored numerous papers and reports on reuse, writes a reuse column for the Object Magazine and is active on several reuse program committees. He was until 1982 an Associate Professor of Computer Science at the University of Utah, where he worked on portable Lisp compilers and interpreters, and on computer algebra systems.

Dr. Robert R. Kessler has been on the faculty of the University of Utah since 1983, currently holding the position of Associate Professor of Computer Science. Prior to going on sabbatical at Hewlett-Packard Research Labs, Professor Kessler was director of the Center for Software Science, a research group working in nearly all aspects of system software for sequential and parallel/distributed computers. He has interests in parallel and distributed systems, portable Lisp systems, architectural description driven compilers, software engineering, and general systems software. He has experience with practical business software as well as academic software development and most recently is working on component-oriented software development using Visual Basic and OLE. Professor Kessler completed his first textbook in 1988, *Lisp, Objects, and Symbolic Programming*. He is currently Co-Editor-in-Chief of the International Journal on Lisp and Symbolic Computation.

Appendix A - Communication Interface Between The Clients And Box Server

This appendix describes the architecture of the Box Server and clients as shown in Figure 9 and the methods used to communicate between them. The main architectural features are the Control

Lab Interface Handler, which handles the serial line and low-level interface to the Control Lab; the mapper that takes each High-level Command and turns it into a sequence of Low-level Commands; and finally, the part used to interface the clients and server. Each of these parts is described in more detail in the following sections.

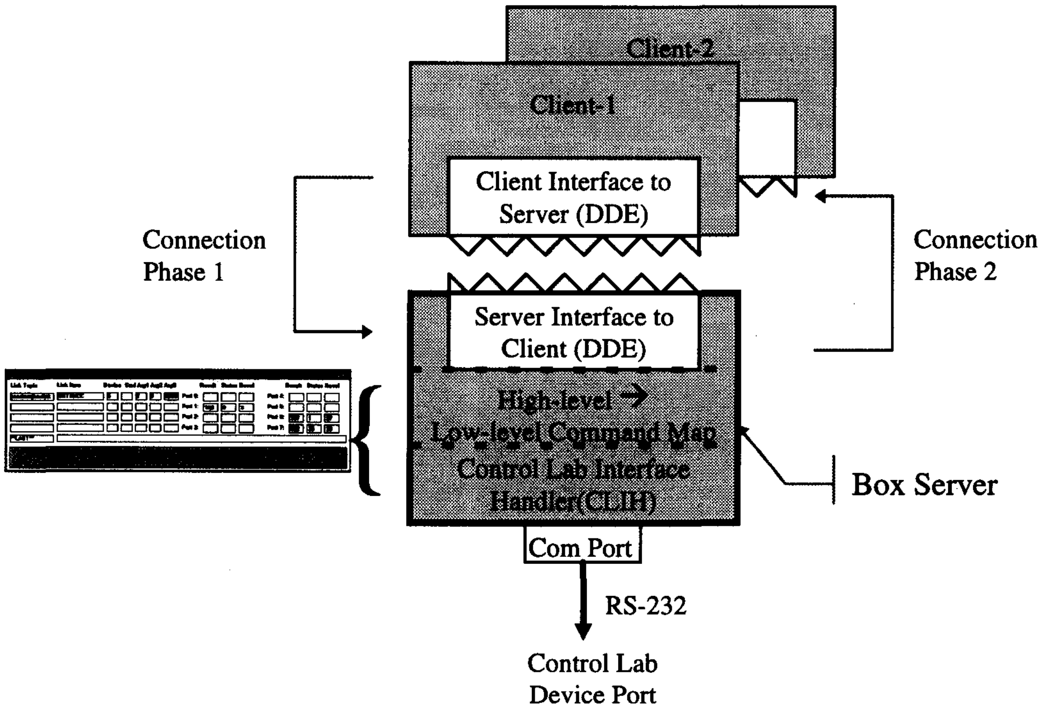

Client-2
Client-1
Client Interface to Server (DDE)
Connection Phase 1
Connection Phase 2
Server Interface to Client (DDE)
High-level → Low-level Command Map
Control Lab Interface Handler(CLIH)
Box Server
Com Port
RS-232
Control Lab Device Port

Figure 9: Box Server architecture.

A.1 Control Lab Interface Handler

The Control Lab Interface Handler (CLIH) was implemented independently of the Box Server and is used in several other application programs (for example, one of the programs provides a Command Button interface to turn on and off the various output devices connected to the Control Lab). The CLIH uses a hidden form that includes the MSComm control for the serial interface, a `keep_alive` timer, an `input_timer`, and a command button control named `empty`. The serial interface and `keep_alive` timer code is straightforward. Processing in the `input_timer` is more interesting due to the callback mechanism. The `input_timer` wakes up every 15 milliseconds and looks to see if a 19 byte sensor frame is present. When a frame is found, the timer code then checks to see if the sensor data has changed. If it has changed, it checks to see if anyone has registered to monitor the sensor. If so, it notifies that code that there is new sensor data. A callback mechanism is used to handle this notification.

The callback mechanism is handled by any code that wishes to be notified when a change happens first creating a command button that is used as the interface between itself and the

CLIH. The callback model is that the code in the command button's change event will be executed whenever the sensor data changes. The first step in the process is registration of the command button by calling the global register procedure, passing the command button and the sensor device number (a value between 0 and 7). The corresponding unregister procedure is used to disconnect. The CLIH uses a global array named callback to store the command buttons. The empty command button on the hidden form is used to indicate that no one is interested in a particular sensor. As you might expect, the eight elements in the callback array correspond to the eight input sensors. This means that only one command button at a time can be monitoring a particular sensor.

```
' Callback is a list of controls in other forms that
' we will notify when that input value changes in its
' analog value.
Global callback(7) As Control

Sub register (ctrl As Control, device As Integer)
  ' This procedure registers a control as being
  ' interested in the changing values of the argument
  ' device.
    Set callback(device) = ctrl
End Sub

Sub unregister (ctrl As Control, device As Integer)
  ' This procedure unregisters a control as being
  ' interested in the changing values of the argument device.
    Set callback(device) = legoform.Empty
End Sub
```

Initialization of callback is handled in the form_load code for the hidden CLIH form:

```
Sub Form_Load ()
    Dim dummy
    Dim i As Integer
    For i = 0 To 7
        Set callback(i) = empty
        ' Irrelevant code deleted
    Next i
    ' Irrelevant code deleted
    ' Initialize the Control Lab interface.
    dummy = Init_box(1)
    ' Turn on the timer and start collecting input data
    Input_Timer.Interval = 15
End Sub
```

The internal code in the input_timer that notifies when a change happens is in the following fragment. Once it is determined that code is registered for the sensor, that associated command button's tag is set to the sensor number (allowing the code associated with a command button to be used for many different sensors). Then the command button's change event is forced to fire by setting its value to 1 Finally, the input_timer waits until the change event code is finished executing, which is signaled by the command button's value changing back to 0.

```
      ...
        If Not (callback(sensor_num) Is empty) Then
        callback(sensor_num).tag = sensor_num ' Tell which sensor changed
        callback(sensor_num) = 1 ' Force the associated control change event
        Do
        Loop Until callback(sensor_num) = 0
     End If
     ...
```

Typical change event code for the interface command button follows. Note, the three global arrays `analog`, `status`, and `revolutions` have the current sensor values and are typically accessed within the body of the code.

```
Sub GW_Change ()
    If gw = 1 Then
        dim sensor as integer
        sensor = tag
        ' Process the sensor values.  The sensor data is in three variables:
        ' analog(sensor), status(sensor), and revolutions(sensor).
        gw = 0
    End If
End Sub
```

This type of interface works very well and is efficient enough to properly handle the Control Lab. A few observations:

1. More modular code would result if the three sensor values could be passed in as arguments to the gateway instead of storing the values in global variables. This is not directly possible using a command button as the interface, so the only other viable solution is to use known variables that can contain the argument values (like `arg1`, `arg2`, `arg3`). Then the procedure could pick up its arguments from those globals. In this case, we decided that since the current sensor values were already stored in the globals, we might as well save the time required to copy into the other globals and simply permit external access to those values. Note, this whole mechanism is not safe in an environment that allows multiple threads of execution. One could easily have problems of contention and indeterminate access to the values. However, the current Windows system does not allow multiple threads of execution within a single VB program.

2. Since the `input_timer` loop is busy waiting until the change event code is finished, it is imperative that the gateway not do very much processing. In the current system, we have only one routine which must do a large amount of processing (it has to copy an array of sensor data into a file), so it schedules a timer to perform the copy when the input loop is not busy.

3. Most of our applications that call the library have a command button control array for each sensor. This means that the call is already indexed by the sensor device number, so, the code to dimension and set the `sensor` variable could be eliminated and the typical `index` argument used instead.

4. Finally, this mechanism would not be needed if VB allowed one to call a non-local procedure (which may be internal to a form) by its name or with a pointer to the code.

A.2 The Mapper

The command mapping part of the Box Server is responsible for turning high-level commands into an appropriate sequence of low-level commands. The CLIH defines a set of simple procedures that perform the basic operations in the Control Lab. The output routines send byte strings to the Control Lab to perform operations such as "turn on," "set speed," "turn off," and "set direction." The CLIH input routines translate the values in the sensor data structures from their analog values into more meaningful data such as "Fahrenheit temperature," "light sensor value," and "rotational count." The high-level commands of the Box Server have five components: the command number, the device number, and three argument values. The mapper takes these commands and calls the CLIH routines. It also sets up sensor monitoring command buttons and `change` event code for handling any input sensor data requested by the command.

A.3 The Server Interface to the Client

Establishing communication between a Client and the Box Server is complicated by two factors. The first factor is that the server does not know what clients may be using its services. This results in a complicated registration process. The second factor is that we must establish communication in both directions. The clients send commands and the Box Server returns results with the added complication of request/acknowledge protocols between the two. The following illustrates solving this problem using DDE as our transport mechanism. Note, we had considered using an OLE interface between the client and server, but it is currently not possible to use OLE for communicating between two vanilla VB 3.0 programs (except possibly by using a third party C or C++ DLL/VBX).

The easiest way to think about DDE communication is to consider shared memory between two processes. If two independent processes have some shared memory, then when one process writes into the shared memory, the other process would be able to immediately read that value. In a truly shared memory model, this would also work in the reverse direction. In DDE, shared memory is represented by two controls that are shared between the two processes. The normal way of using DDE is that it works in only one direction from a source to a destination. When the source is changed, then the destination sees that change. In DDE, you can communicate the other direction using the `LINKPOKE` method, but this is forcing the communication in the wrong direction and requires special handling.

Given the above DDE model, our system sets up commands so the source is in the client and destination is in the Box Server, with a separate link set up in the opposite direction for the flow of results back from the Box Server to the client. An important complicating factor is that linkage between a source and destination must be initiated on the destination side. That is, the destination must say that it wants to receive information from the source. So, when the Box

Server wants to receive information from the client, it is responsible for establishing that connection.

Given this background, the difficult part of getting these two processes to communicate is handling the command and argument processing. Since we want the ability for the client to issue the command and the server to automatically pick it up, the Box Server must establish communication with the client. However, the Box Server does not know who the client is, so the client must send the server the necessary information that allows it to establish the link. Communication in the other direction is easier because the client can simply find one of the several result data collections and then establish the communication to the Box Server itself.

As shown in Figure 9, establishing the link requires two phases. The first step, "Connection Phase 1," involves the client sending the server information about the client. In "Connection Phase 2," the server establishes the DDE link with the command and arguments in the client. The part where the client wants result data is not shown in the figure, but involves similar processing as Phase 2, except it is initiated by the client.

Connection Phase 1 requires the following steps:

1. Find the right Box Server process.

2. Determine if the Box Server can handle any more clients and if so get the next available communication location.

3. Tell the Box Server who the client is so the Box Server can establish the communication in the reverse direction.

4. Finally, tell the Box Server the names of the controls that the client will use for sending commands and arguments to the Box Server.

Each of these steps is detailed in the following paragraphs.

Since Client and Box Server communication via DDE is with controls, we need to set up the controls in each process. Every client has a hidden form called "`boxlink`" which has the controls used for its side of the communication. The Box Server uses the seven columns of controls on the left side of the Box Server panel as shown in Figure 5. Thus establishing communication is accomplished by setting up the linkage between the `boxlink` and the Box Server forms.

When any DDE conversation is initiated, one must specify both an application name (`boxsvr`) and a uniquely named topic for communication, both separated by a vertical bar (|). In our case, there is the possibility of multiple Box Server programs running simultaneously (one for each COM port). So we needed a way of distinguishing the right program instance. Without doing something special, each program instance would end up with the same exact link topic (since the program is the same program). The trick that we use is that when the Box Server begins execution on a COM port port_num, it changes the name of its linktopic to include the COM port number (it concatenates "COM" & port_num to produce the entire client linktopic of "boxsvr|COM1"). This linkage guarantees that we are talking to the Box Server attached to the correct COM port.

Since the Box Server can communicate with several clients, there can be multiple, simultaneous command and argument sequences being processed. So, the client must arrange with the Box Server which command/argument sequence it should use. In our implementation, the Box Server has an array of commands and arguments, with the designation that each array index is used for one client communication. Therefore, the client must determine if any array indices are currently available and if so, take control of that index. This is accomplished by the client scanning the server's "topic" array, looking for an empty value. Once an empty index is found, the client reserves it and is free to use that index for the rest of the communication.[10]

At this point, the client has established communication into the server, so the client then needs to tell the server who the client is and how to communicate back. The first step is to write into the linked data structure the linktopic string that will be used to talk back to the client. Just like when we established the linkage to "boxsvr|COM1" we must tell the Box Server the client application name and link topic. The hidden boxlink form in the client uses the name "boxlink" as its topic name, so we write the client's app name (e.g. wave) and boxlink into the DDE link, poking the link to get the Box Server to recognize that it has a new value. Note, the current code assumes that each client will have a unique name.

Finally, the Box Server needs one more piece of information. It needs to know the names of the controls that are on the boxlink form that are used by DDE to communicate the command and argument values. Instead of sending over the name of each and every control, we have established a protocol that the five values (representing the command, the device number, and three additional arguments) are all named with the same common prefix followed by "_cmd", "_device", "_arg1", "_arg2", and "_arg3". So, all the client has to do is send over the common prefix (in our case called "boxsvr").

Once all of this is done, the client waits to make sure that the Box Server has finished all of its processing and established the appropriate linkages. So, the client code waits for the server to signal a "DONE". The procedure open_COM_port does all of these steps.

[10] An obvious problem with this is what happens if two clients attempt to access the same index value at the same time. They both may realize that the entry is empty and blindly establish communication with it. We decided that the chances of this were small and ignored it in our prototype. A better solution is for the Box Server to be responsible for finding and reserving the array index itself, then communicating the index result back to the client.

```
Sub open_COM_port (port_num)
  ' Port_num is the COM port number, a value from 1 to 4
    Dim dum ' Dum is ignored result of shell and doevents.
    Dim winname As String
    Dim boxlink_client As Integer
    ' ** Step 1 - find the right Box Server process.
    winname = "Control Box Interface - Port " & port_num
    If FindWindow(0&, winname) = 0 Then
        ' boxsvr_exe is the name of the boxserver executable.
        dum = Shell(boxsvr_exe & " " & port_num, 1)
    End If
    dum = DoEvents()
    ' Set up the link with the VB source.
    boxlink.boxsvr.LinkTopic = "boxsvr|COM" & port_num
    ' ** Step 2 - find an available communication slot. Set the text value to something
    ' other than "", so the while loop gets a chance to execute at least once.
    boxlink.boxsvr.Text = "dummy"
    boxlink_client = 0
    While boxlink.boxsvr.Text <> ""
        ' Reset the linkmode to NONE to wait until we set a new linkitem.
        boxlink.boxsvr.LinkMode = NONE
        ' Establish link to the next text box on the server.
        boxlink.boxsvr.LinkItem = "topic(" & boxlink_client & ")"
        boxlink.boxsvr.LinkMode = AUTOMATIC
        ' Link is established so first Test to see if we have exceeded the number of clients.
        If boxlink.boxsvr.Text = boxsvr_max_keystring Then
            MsgBox "Cannot connect to server: No more clients", 16
            End
        End If
        ' See if this topic is currently available.
        If boxlink.boxsvr.Text <> "" Then boxlink_client = boxlink_client + 1
    Wend
    ' ** Step 3 - tell the Box Server who we are. First clear out the command field so
    ' we don't mistakenly send a command.
    boxlink.boxsvr_cmd = ""
    ' The topic field will not have our app name and form name.
    boxlink.boxsvr.Text = LCase(app.Title) & "|boxlink" ' Set up the call back
    boxlink.boxsvr.LinkPoke
    boxlink.boxsvr.LinkMode = NONE          ' reset connection
    ' ** Step 4 - tell the Box Server the prefix name of our command/arg controls
    boxlink.boxsvr.LinkItem = "item(" & boxlink_client & ")"
    boxlink.boxsvr.LinkMode = AUTOMATIC
    boxlink.boxsvr.Text = "boxsvr"
    boxlink.boxsvr.LinkPoke
    ' Now wait until the boxsvr is finished with its processing.
    wait_for_ack
End Sub

Sub wait_for_ack ()
  ' Wait for an acknowledgement to come back.

    ' Wait until the box is finished processing.
    While boxlink.boxsvr <> "DONE"
        DoEvents
    Wend
    ' Tell the Box Server that we received the DONE signal.
    boxlink.boxsvr = "GOTACK"
    boxlink.boxsvr.LinkPoke
    DoEvents
    ' Clear out the current command, so if we happen to issue the same
    ' command two times in a row, the change event will guarantee to fire.
    boxlink.boxsvr_cmd = ""
End Sub
```

Sidebar: One of the tricky issues in client/server computing is naming. How does the client find the server? In sophisticated applications with many different clients and servers, servers registers their application under a particular known name with a naming server (e.g. I am "boxserver" and my server name is "the_box"). A client that desires to set up communication with a particular server would ask the naming server for the server location (e.g. in this case, asking for "the_box" and receiving back "boxserver" as the name of the server) and then proceed with direct communication once the server has been found and the link has been established. In our application, we have designated that the server has a known name, which any client must use directly. Although this saves the overhead of setting up a naming server and that initial establishment of communication, it is less flexible because all clients must be updated if the name of the server program changes (e.g. if we wanted an experimental version of the boxserver, we would have to save the old version of boxserver under a different name and then call the new version boxserver instead of registering some new name like "expbox" as "the_box" with the naming server). Note - an interesting side issue is how to find the naming server? This is usually done by searching for a known Name or by broadcasting for the Name Service.

A.4 The Client Interface to the Server

When the Box Server starts, it sets its `linktopic` to include the COM port number (which is passed as a command argument to the executable program). It also changes its own `app.title` to include the COM port number so that when the program is viewed in the task list, we can easily distinguish the target COM port of each particular program instance.

```
Sub Form_Load ()
    ' Get the command line args, which includes the COM port number.
    Dim args As String
    args = Command$
    lego_COM = Val(args)
    If lego_COM = 0 Then
        MsgBox "Error, Server must be started with COM Port Number", 16
        End
    End If
    ' Set the name of the caption which is used to find us in the client.
    form1.Caption = "Control Box Interface - Port " & lego_COM
    ' Change my name and link topic name to identify the port.
    app.Title = "BOXSVR" & lego_COM
    form1.LinkTopic = "COM" & lego_COM
    ' Load the hidden form of the CLIH.
    Load legoform
End Sub
```

Now, the server waits for Connection Phase 2 to start. This happens when it gets values for both its `topic` and `item` fields. Recall that `topic` contains the `linktopic` to communicate with the client (`app_name|topic_name`) and the `item` control contains the prefix name of the command and argument controls in the client. Once both of these controls have values, the establish_links subroutine is called. This routine sets up the linkages between the local command, device, and argument controls with the remote counterparts.

```
Sub Topic_Change (Index As Integer)
    If topic(Index) = "" Then Exit Sub
    If item(Index) <> "" Then establish_links Index
End Sub

Sub Item_Change (Index As Integer)
    keep_alive
    Dim ii as string
    ii = item(Index)
    ' DONE and GOTACK are used for request/acknowledge communication.
    If ii = "" Or ii = "DONE" Or ii = "GOTACK" Then Exit Sub
    If topic(Index).Caption <> "" Then establish_links Index
End Sub

Sub establish_links (Index As Integer)
    cmd(Index).LinkMode = NONE
    cmd(Index).LinkTopic = topic(Index).Caption
    cmd(Index).LinkItem = item(Index).Caption & "_cmd"
    cmd(Index).LinkMode = AUTOMATIC
    keep_alive
    device(Index).LinkMode = NONE
    device(Index).LinkTopic = topic(Index).Caption
    device(Index).LinkItem = item(Index).Caption & "_device"
    device(Index).LinkMode = AUTOMATIC
    keep_alive
    arg1(Index).LinkMode = NONE
    arg1(Index).LinkTopic = topic(Index).Caption
    arg1(Index).LinkItem = item(Index).Caption & "_arg1"
    arg1(Index).LinkMode = AUTOMATIC
    keep_alive
    arg2(Index).LinkMode = NONE
    arg2(Index).LinkTopic = topic(Index).Caption
    arg2(Index).LinkItem = item(Index).Caption & "_arg2"
    arg2(Index).LinkMode = AUTOMATIC
    keep_alive
    arg3(Index).LinkMode = NONE
    arg3(Index).LinkTopic = topic(Index).Caption
    arg3(Index).LinkItem = item(Index).Caption & "_arg3"
    arg3(Index).LinkMode = AUTOMATIC
    keep_alive
    ack_done Index
End Sub
```

The last step of establishing the linkage is for the server to acknowledge that it is finished. This is accomplished by letting the item control serve double duty. This is possible because once the command and argument linkage is established, the item value is no longer needed. So, the code uses the item control array element to communicate acknowledgments. The server writes the key phrase "DONE" into the item and then goes back to sleep waiting for something else to do. Note, one downside of using the same control for multiple purposes is that we had to specially code the change event for the item control to ignore DONE (and any of the values that are used in the request/acknowledgement protocol).

```
Sub ack_done (Index)
    item(Index) = "DONE"
    keep_alive ' Send keep alive byte to box.
End Sub
```

Finally, we wanted to mention an interesting heuristic that we discovered relating to the problems of handling real sensor data and attempting to transmit that back to the client. For some of the sensors, their values change in every frame. This means that the DDE link must be able to handle each new value every 1/50th of a second. We discovered that on our hardware configuration[11], this overloaded the system and made it impossible for the Box Server inner loop to actually get sufficient cycles to keep up with the sensor frames. Our solution was to throttle the changes by only updating the DDE return value every 10 times that it actually changed. This is sufficient when monitoring a few sensors and is a fairly typical solution to the problem of reduction of sensor duty rate. Our next version of the system will allow the specification of the stability rate as a part of the client registration process (i.e. "please inform me when the data is stable for 'n' seconds.").

[11] Our hardware is an HP Vectra 486/66XM with 24 MB of RAM and running Windows NT 3.5.