The Phidget Architecture: Rapid Development of Physical User Interfaces

Chester Fitchett and Saul Greenberg
Department of Computer Science
University of Calgary
Calgary, Alberta, Canada T2N 1N4
Tel: +1 403 220 608, E-mail: saul@cpsc.ucalgary.ca

Many ubiquitous computing environments rely on special purpose physical devices to for input (sensors, etc.) and output (motors, lights, etc.) Yet everyday programmers interested in such environments face considerable hurdles creating, developing and combining physical devices and interfacing them to conventional programming languages. They usually do not have adequate know-how; appropriate off-the-shelf devices and construction kits are rarely available; commercial hardware is at the wrong level of abstraction or do not have an easily-programmable API. Consequently, programmers either give up or find themselves immersed in a quagmire of tediousness: selecting and purchasing electrical components and hobby kits, circuit board design, microprocessor programming, wire protocol development, and so on.

As a consequence of these problems, we made a concerted effort to think about how we could package physical devices and their software for easy development of physical user interfaces. Our approach was to develop physical widgets, or *phidgets*, which are almost direct analogs to how graphical user interface (GUI) widgets are packaged and 'dropped into' software applications. Our primary belief is:

> … just as widgets make GUIs easy to develop, so could phidgets make the new generation of physical user interfaces easy to develop.

In a previous paper[1], we described the general idea of phidgets and how we built them. We also showed that everyday programmers could develop physical user interface easily and with minimal training. Here, we detail the phidget architecture and its design considerations.

Requirements for a phidget architecture

Before delving into details, we need to consider what a phidget architecture has to support.

1. *Insertable components.* Like graphical widgets, a phidget should be presented to the programmer as an easily used component that can be inserted into an application. The component should supply an abstracted and well-defined interface to manage its physical entity. It should hide details of how the entity is implemented.
2. *Connection manager.* Physical devices may appear and disappear. For example, during run time a device may come on-line or go off-line, or it may have intermittent connectivity (especially if it is wireless). The job of a connection manager is to monitor and communicate with attached devices, to inform the application program about the appearance and disappearance of particular devices, and to give the programmer a 'handle' to devices as they appear.
3. *Identification.* There must be a way to link a software phidget with its physical counterpart. While not a problem when there are only a few well-known devices attached to a single computer, device identification can become an issue when several devices of the same type (but perhaps with different end uses) are attached to the computer, or where the types and numbers of devices are not known ahead of time. A clear identification scheme is required.
4. *Simulation mode.* For software development purposes, the same phidget code should work in a simulation mode. That is, the software designer should be able to program, debug and test the system even if the actual physical device that comprises part of the phidget is absent. This could include an extended API to set the simulation characteristics of the device, and a graphical representation that allows a person to see and optionally interact with the device state.

Example Phidgets

We have completed several types of phidgets that support the features listed above. Other phidgets are in progress.

GlabServo lets a programmer control a device containing several servo motors. The position of each motor can be set programmatically (Figure 2);

GlabPowerBar resembles a standard 120-volt power bar with several outlets. The programmer can programmatically and rapidly turn individual outlets on and off (not shown);

GlabInterfaceKit is a general-purpose 'construction' kit,

Fitchett, C. and Greenberg, S. (2001) **The Phidget Architecture: Rapid Development of Physical User Interfaces**. in *UbiTools'01 Workshop on Application Models and Programming Tools for Ubiquitous Computing*. Held as part of UBICOMP '2001.

http://www.cpsc.ucalgary.ca/grouplab/papers/index.html.

[1] Greenberg, S. and Fitchet, C. (2001) Phidgets: Easy Development of Physical Interfaces through Physical Widgets. *Proc UIST 2001*, ACM.

where one can plug in a combination of off-the-shelf switches, LEDs, solenoids, sensors and so on (Figure 3). Specifically, a programmer can control up to 8 digital output devices (e.g., LEDs or solenoids), can retrieve the state of up to 8 digital input devices (e.g., various types of switches); and can inspect the state of various analog sensors that can be connected to it (e.g., heat, force and light sensors.

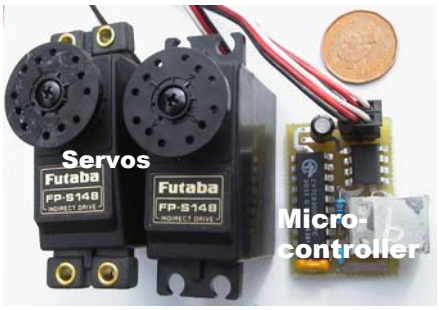

Figure 2: GlabServo and its motors

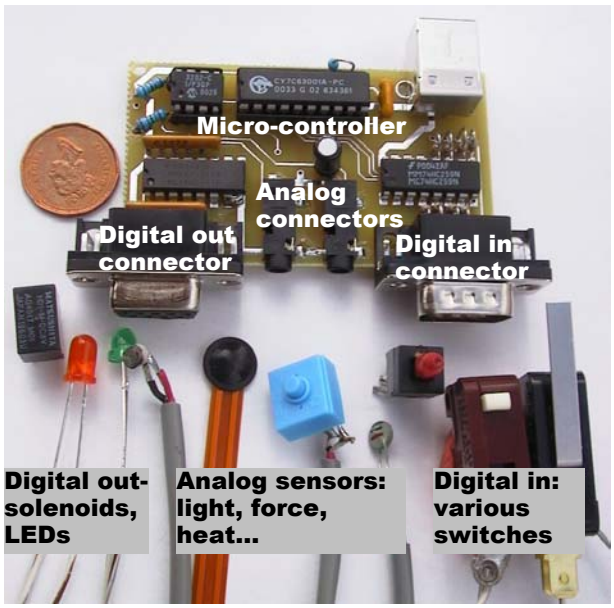

Figure 3 GlabInterfaceKit and a host of sensors, switches, LEDs and solenoids that can be connected to it.

Architecture

Our phidgets abstract out into the following architectural units, illustrated in Figure 4. We will use the GlabServo phidget to illustrate particular details.

The ***Physical Device*** is the packaged physical unit given to the physical designer, who may then use it in whatever way she wishes to create a physical interface that would be given to the end user (Figure 4 left side). The physical device includes the primitive input and output device components (sensors, motors, switches, etc), a circuit board with micro-controller, and a communications layer. For example, the primitive device components of our GlabServo are the actual Servo motors, while for the GlabInterfaceKit it would be the various sensors and switches that can be plugged into it. Most our phidgets are built around a circuit board using a USB micro-controller to control the on-board electronics. This communication layer is based upon the USB communication standard, and it is the USB micro-controller's responsibility to handle the communication protocol with the host computer. Finally, device packaging depends on the device, as illustrated in Figures 2-4 The GlabServo is delivered as a small circuit board (~1.5 $cm^{2)}$ as illustrated in Figure 2, and device designers can optionally attach one or two servo motors to it. In contrast the GlabPowerBar is packaged as a full-size power bar (we actually adapt a commercial one) with the electronics hidden inside.

The ***Wire Protocol*** is the communication protocol between the physical device and the host computer (we use MS Windows 2000). It is not visible to end programmers. As mentioned, our current phidget set communicates using standard USB protocol[2], where we wrote low-level software for both the micro-controller and Windows 2000 to set up and manage basic communication. When our physical devices are plugged in, Windows sees them as USB devices. Atop this protocol, every device knows and can transmit its phidget type (e.g., a GlabServo transmits

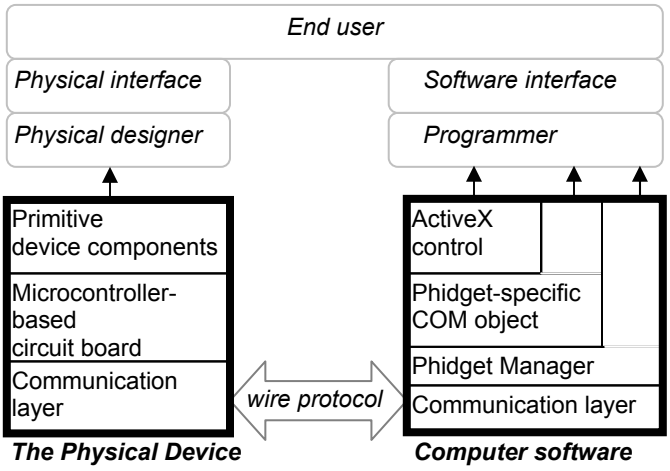

Figure 4 Phidget Architecture

the string "GlabServo"), and an identification number that is unique for a phidget instance of that type (Point 3 in requirements). Each device also transmits information specific to its type e.g., particular events that indicate the device state. Similarly, the host computer can transmit device-specific requests. For example, a host can tell the GlabServo device to set a motor's position to a given angle.

The ***PhidgetManager*** is a COM[3] object. It includes an event-based API available to end-programmers for connection management (Point 2 in requirements). Its major API elements include:

```
Properties:
   Count as Integer
   Item (Index as Integer)
Events:
```

[2] The current architecture is not limited to it. Because the Phidget Manager abstracts the communication layer for all phidgets, this is the only architectural component that would have to be extended to support other communication links, such as X10, Ethernet, wireless, or RF.

3 COM objects are Microsoft's standard way of packaging, distributing and including software modules. COM APIs are accessible from a variety of programming languages.

```
OnAttach (Phidget as IGlabPhidget)
OnDetach (Phidget as IGlabPhidget)
```

Programmers use this API to discover all attached devices. Specifically, the `OnAttach` and `OnDetach` events are automatically generated whenever a physical device is connected or disconnected to or from the computer. These events return a reference to an `IGlabPhidget` interface that the programmer can use to identify the device (see below). For example, if an end user plugged in a GlabServo, the `OnAttach` event would automatically fire and return a reference that the programmer can use to discover that it is a GlabServo. Alternatively, the programmer can find out how many phidgets are currently attached via the `Count` property, and enumerate through them via the `Item(Index)` property.

Internally, the PhidgetManager is implemented as a layer of abstraction built atop the USB communications layer (Figure 4 right). It monitors USB devices on the system to see if they are phidgets: if they are, it creates a phidget-specific COM object (Figure 4), and passes back an `IGlabPhidget` interface to this object through the `OnAttach` event. Behind the scenes, the Phidget Manager also serves as a transport layer: it mediates all communications between all upper layers and the USB layer. This is not visible to the programmer.

Phidget-specific COM objects are created by the Phidget Manager whenever a device is seen (Figure 4, right). These objects correspond directly to physical devices e.g., a GlabServo physical device corresponds to a GlabServo COM object, a GlabPowerBar device to a GlabPowerBar object, and so on. Internally, all phidget-specific COM objects communicate to its matching physical device through the PhidgetManager.

Because these objects have to be created at run time when a device is plugged in, there are two interfaces (or APIs) to this object: the generic IGlabPhidget interface, and the specialized phidget-specific interface, as described below.

IGlabPhidget interface is a required interface provided by all phidget-specific COM objects. Through this generic API, end-programmers can identify basic properties of any phidget-specific COM object returned by the Phidget Manager whenever a device is seen.

```
Properties:
   DeviceType As String
   IsAttached As Boolean
   SerialNumber As Long
```

Through this `IGlabPhidget` interface, the programmer can discover what kind of device it references (and thus assign it to an interface specific to the object, as described shortly), its serial number, and whether the physical device is still attached. For example, the programmer could test if an attached device is of the `DeviceType` "GlabServo", optionally check its `SerialNumber` to discriminate between multiple instances of attached GlabServos, and then assign this object to its more specialized GlabServo phidget interface (see below).

The ***phidget-specific interface*** is a superset of IGlabPhidget in that it also exposes an API specialized to the particular phidget-specific COM object (Point 1 in requirements). For example, the GlabServo COM object API also includes properties and events to handle its various motors:

```
Properties:
   MotorPosition(Index) as Integer
   NumMotors as Integer
Events:
   OnPositionChanged (Index as Integer
                      Position as Integer)
```

Thus the programmer can find out how many motors are available using the `NumMotors` property, can set a particular motor's position through the `MotorPosition(Index)` property, and will receive the `OnPositionChanged` event whenever a motor is repositioned. Of course, other phidget-specific COM objects will have their own device-specific API. Essentially, specialized interfaces such as these allow a programmer to directly control the device and get feedback of its state.

Phidget ActiveX Controls[4] wrap our various phidget-specific COM objects to give each of them an on-screen interface and a simulation capability (Figure 4 right). Programmers have the choice of using either these visible ActiveX controls with simulation capability or its simpler phidget-specific COM counterpart as appropriate.

Unlike the phidget-specific COM object, the control provides a visual interface to the device, where it displays its real or simulated state as well as the optional means for an end user to interact with its on-screen representation. Programmers can easily drop its visual representation into an interface builder (e.g., Visual Basic). Each control can optionally operate in a simulated mode when no actual physical device is connected to it (Points 1+4 in requirements). Here, the software mimics the device's behavior. Finally, the control includes extensions to the phidget-specific API for managing these new features.

For example, the GlabServo ActiveX Control (Figure 5) has two graphical motors. Users can interactively rotate the motors to new positions by dragging the motor platter, which will also reposition the actual motors if the device is attached. Some examples of its extended API include:

```
Properties:
   BackColor as OLE_COLOR
   FillColor as OLE_COLOR
   Enabled as Boolean
   SimulateWhenDetached as Boolean
```

Here we see a few properties for setting the colors in the control (BackColor and FillColor), whether the control is interactive (Enabled), and whether the control should simulate a physical device if one is not attached (SimulateWhenDetached).

[4] ActiveX controls, specialized COM objects, are Microsoft's standard way of packaging graphical widgets containing a visual region that can be displayed on-screen.

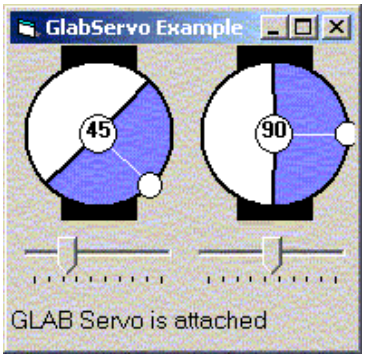

Figure 5 A screen snapshot of the example program

```
Private WithEvents PM As GlabPhidgetManager          'The phidget manager

Private Sub Form_Load()                              'Initialization
  Set PM = New GlabPhidgetManager                    'Start the phidget manager
  Servo.SimulateWhenDetached = True                  'Simulate Servo if needed
  Servo.MotorPosition(1) = 45                         'Position motor1 to 45 degrees
  Servo.MotorPosition(2) = 90                         'and motor2 to 90 degree
  label.Caption = "Simulated: no device attached"    'On-screen feedback
End Sub

'Event handler: Connect to the servo device when it is attached (or plugged in).
Private Sub PM_OnAttach(ByVal Phidget As GLABPHIDGET.IGlabPhidget)
  If Phidget.DeviceType = "GLAB Servo" Then          'A servo device has appeared
    Set Servo.ServoPhidget = Phidget                 'We link it to the servo phidget
    label.Caption = Phidget.DeviceType & " attached" 'On-screen feedback
  End If
End Sub

'Event handler: When the Servo phidget is disconnected, it automatically continues to simulate it.
Private Sub PM_OnDetach(ByVal Phidget As GLABPHIDGET.IGlabPhidget)
  If Phidget.DeviceType = "GLAB Servo" Then
    Set Servo.ServoPhidget = Nothing
    label.Caption = "Simulated: no device attached"   'On-screen feedback
  End If
End Sub

'Event handler: The servo generates an event every time its position is changed.
'We use this to reset the position of the sliders
Private Sub Servo_OnPositionChange(Index As Integer,Position As Integer)
    Slider(Index).Value = Position
End Sub

'Event handler: As the user moves a slider, rotate the corresponding servo to the position indicated
Private Sub Slider_Scroll(Index As Integer)
    Servo.MotorPosition(Index) = Slider(Index).Value
End Sub
```

Figure 6 A complete Visual Basic program for interacting with two servo motors

Example Program

While sounding complex, this architecture is surprisingly easy to use in practice. The Visual Basic (VB) program in Figure 6, shown running in Figure 5, illustrates the complete source for a toy application controlling two servo motors. The programmer used VB's interface builder to drop in a GlabServo ActiveX control and three conventional widgets: a label for displaying a text message and two sliders set to return values between 0 and 180. This takes seconds to do. The label provides textual feedback on whether the GlabServo is being simulated or if the device is actually connected. The individual sliders are used to position the motors.

The programmer sets the motor positions and the simulation option in the `Form_Load` initialization routine. He connects and disconnects to a physical servo device using the `OnAttach` and `OnDetach` event handlers. Because the end user can also set a motor's position by directly rotating its image of the motor platter, the programmer must update the slider's position when notified by the `OnPositionChanged` event handler that the motor position is changed. The exectuable program works in both simulated and non-simulated mode. If no servo is plugged in, its behavior is simulated on screen and the end user can still interact with it. As soon as a GlabServo device is plugged in, the physical motors will automatically rotate to the current simulated motor settings.

Our other phidgets are programmed just as easily. For example a GlabPowerbar phidget would be detected the same way, and a particular outlet could be turned on by a line of code resembling: `PB.OutletState(2)=True`. A GlabInterfaceKit is slightly more complex as it has both input and output values. Typically, changes to input values (such as those generated by sensors) are returned via an event. For example, this event handler would detect and print out changes to values generated by a light sensor:

```
'Report a sensor's value whenever it changes
Private Sub PS_OnSensorChange(_
      Index As Integer, SensorValue As Integer)
  Print "Sensor: " & Index & ":" & SensorValue
End Sub
```

Architecture extensions

One more software component enriches the kinds of applications we can build: a shared dictionary. Any distributed process can publish key/value pairs into this dictionary. Similarly, any process can subscribe via pattern matching to particular keys: by doing so, they are automatically informed of changes to these key/value pairs via events. It then becomes very simple to program groupware based on physical devices. For example, an application can capture a person's presence using a well-positioned GlabProximitySensor and publish that into the shared dictionary. Other applications can subscribe to this information and use it to activate other physical devices e.g., by turning a lamp on and off with the GlabPowerBar. We could create context-aware widgets similar by combining, abstracting and publishing contextual readings from various phidgets into the shared dictionary; other applications can then use these values to monitor and react to contextual changes.