Phidgets: Easy Development of Physical Interfaces through Physical Widgets

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

ABSTRACT
Physical widgets or *phidgets* are to physical user interfaces what widgets are to graphical user interfaces. Similar to widgets, phidgets abstract and package input and output devices: they hide implementation and construction details, they expose functionality through a well-defined API, and they have an (optional) on-screen interactive interface for displaying and controlling device state. Unlike widgets, phidgets also require: a connection manager to track how devices appear on-line; a way to link a software phidget with its physical counterpart; and a simulation mode to allow the programmer to develop, debug and test a physical interface even when no physical device is present. Our evaluation shows that everyday programmers using phidgets can rapidly develop physical interfaces.

INTRODUCTION
In the last decade, various movements embraced human-computer interface designs that include physical user interfaces augmented by computing power. These include *ubiquitous computing* and *calm technology* [16], *pervasive computing* [1], *tangible user interfaces* [7], *information appliances* [13] and *context-aware computing* [3].

Researchers in these areas have demonstrated many simple but exciting examples of physical user interfaces. Ishii and his Tangible Media group developed several elegant ambient fixtures to communicate information at the periphery of human perception. These include pinwheels that rotate under program control [2], bottles that play sounds when they are opened [8], touch counters that track and display the use of physical objects [17], and water lamps that project water ripples onto a surface [2]. Heiner, Hudson and Tanaka built the information percolator, where scrolling displays are created by bubbles rising through tubes of water controlled through the release of air [6]. Greenberg and Kuzuoka [4] illustrated how devices can serve as digital but physical surrogates of remote people: they can present the remote person's status, serve as a communication channel, and react appropriately to people's implicit and explicit actions. Kaminsky et. al. [9] detailed how Microsoft Actimates could display notifications, indicate numeric values, and respond to user actions. In a more abstract vein, many media artists have created interactive installations where a combination of computational and physical devices respond to how people move within a space (e.g., see SIGGRAPH Art Galleries).

Cite as:
Greenberg. S. and Fitchett, C. Phidgets: Easy development of physical interfaces through physical widgets. Report 2001-686-09, Deptment Computer Science, University of Calgary. www.cpsc.ucalgary.ca/grouplab/papers/

While an exciting new area, everyday programmers now face considerable hurdles if they wish to create even simple physical user interfaces. Perhaps the biggest—but we believe easily solved—obstacle is the sheer difficulty of developing and combining physical devices and interfacing them to conventional programming languages. Several specific problems are listed below.

1. Even simple devices made out of cheap and readily available electrical components (switches, sensors, solenoids, motors) are hard to build unless one has a background in hobby electronics, circuit design or electrical engineering. Sadly, most computer scientists and human computer interaction specialists lack this background.
2. Commercially available devices may have no published application-programming interface (API). As a result, an outsider cannot program them unless the device is 'hacked' or reverse-engineered. Examples include Microsoft's Actimates hacked by Kaminsky, Dourish and Edwards [9]; Fujitsu's email notification figurine hacked by Greenberg and Kuzuoka [4]); Lego Mindstorms RCX documentation by Knudsen [10].
3. Alternatively, commercial devices designed for particular application settings typically have a configuration and/or API at a level of abstraction that is not well suited for building the kinds physical devices we want. For example, the X10 protocol developed for controlling 'Smart Home' and security appliances is too high-level and limited for general device development (but see [5]). At the other extreme, we have programmable logic controllers (PLCs) for constructing control devices used in manufacturing: these are abstracted at a very low level, where designers may require extensive training and have to do difficult programming to do even the simplest things.

4. Developers may not have these devices readily available at early stages of their programming effort, perhaps due to expense, shipping delays, cost factors, etc. While a program can be written without a device, they are difficult to test and debug.

MOTIVATION: OUR FRUSTRATING 1st EXPERIENCES

Our own first experiences echoed these problems. We were designing a reactive media space environment built around several simple interoperating devices [4]. The devices illustrated in Figure 1 were built upon proximity sensors, servo motors and light sensors, as well as switched cameras, microphones, speakers, and small video displays. While our overall focus was on media space design (details in [4]), we found ourselves immersed in a quagmire of tediousness: selecting and purchasing small electrical components and hobby kits, circuit board design, microprocessor programming, wire protocol development, and so on. Fortunately for the computer scientists in our group, visiting collaborator Hideaki Kuzuoka was proficient in hardware/microprocessor work and assumed this burden. Still, we expended considerable time (months) and effort developing and debugging these devices and their related low-level software. Although successful as a stand-alone project, the tale ended poorly: after Kuzuoka left, the software and the devices themselves became almost impossible to maintain or extend. The problem was that we had built a working prototype, but had not really considered how individual devices and its software could be maintained, modified and reused in different ways. Eventually, the devices visible in Figure 1 ended up as a disassembled mess.

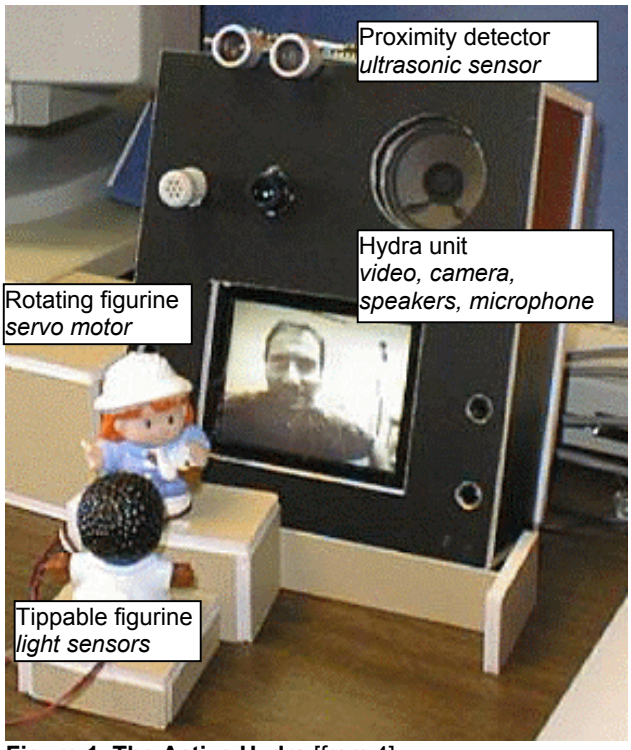


Figure 1. The Active Hydra [from 4]

Our experiences are echoed in the many discussions we have since had with other researchers and artists who have developed physical user interfaces around physical devices. Most commented that they either had to invest considerable time learning basic electronics, microprocessor programming and device-building, or that they had to bring in specialists e.g., electrical engineers or knowledgeable hobbyists. No common devices were easily obtainable, nor were there any high-level development platforms: consequently, people either developed 'one-off' devices, or evolved some limited form of reusable hardware/software modules for in-house use.

RESEARCH AGENDA: THE PHIDGET CONCEPT

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 goals were to create devices:

- simple enough so that developers can concentrate on the overall use, modification and recombination of devices into a physical user interface instead of low-level device construction and implementation;
- easy enough for the average programmer to program and extend.

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[1]. 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.

As we will see, a phidget comprises a device, a software architecture for communication and connection management, a well-defined software API for programming the device, a simulation capability, and an optional on-screen component for interacting with the device.

Why GUI Widgets are so successful

GUI widgets have greatly simplified the programmer's development of interactive software. They abstract and package well-designed standard and non-standard input and output controls. They hide often-difficult implementation details, while exposing functionality through a well-defined API. Through relatively simple programming, they can be interconnected so they can work in concert with one another. As a toolkit set, widgets give the programmer a good repertoire of graphical components that can be used to assemble an interface[2] [12]. The result

[1] Phidgets differ from Phicons [15]. Phicons are input instruments, whereas phidgets are programmable components representing physical devices.

[2] Myers [12] argues that there is a disparity in many GUI toolkits, where building control panels of widgets are extremely easy, but composing non-widget graphics is hard.

is that programmers using widgets can concentrate on GUI interface design rather than low-level graphical programming.

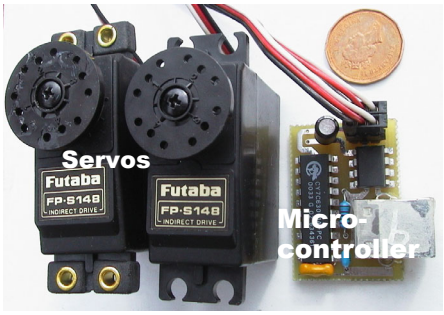


Figure 2: GlabServo and its motors

Phidget requirements

As with conventional GUI widgets, the important idea of a phidget is that it presents the programmer with an easily used entity that can be inserted into an application. Both provide an abstracted and well-defined interface: widgets to a graphical interactive entity, phidgets to a physical entity. Both hide details of how the entity is implemented.

Unlike widgets, phidgets have a few more requirements.

1. *Connection manager.* Whereas GUI widgets are always available to the application at run time, 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.
2. *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.
3. *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.

WHAT WE BUILT

We designed and built a software and hardware architecture, and have completed several types of phidgets that support the features listed above.

Example phidgets

Phidgets we have completed are listed below. Phidget names are prefixed with 'Glab', an abbreviation of our Grouplab research laboratory name.

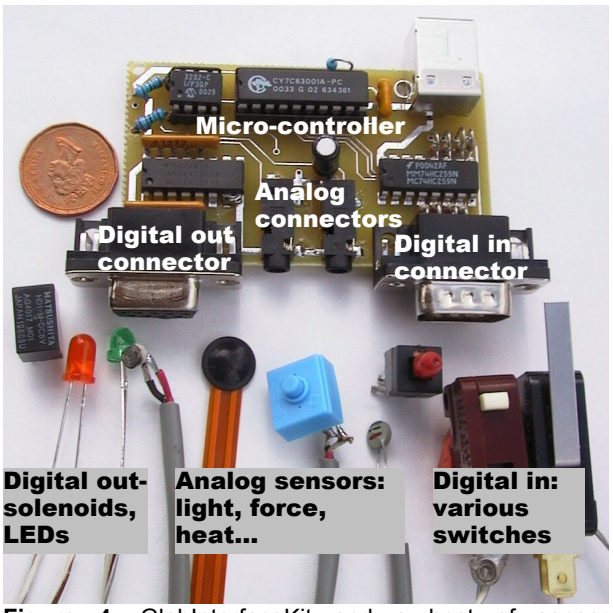


Figure 4: GlabInterfaceKit and a host of sensors, switches, LEDs and solenoids that can be connected to it.

- *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 (see Figure 3);
- *GlabInterfaceKit* is a general-purpose 'construction' kit, where one can plug in a combination of off-the-shelf switches, LEDs, solenoids, sensors and so on (Figure 4). 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.

Other earlier phidgets we built include motion detectors, proximity sensors, and animated figurines. Several are being updated to our current architecture, and we have many other new phidgets in progress.

Software and hardware architecture

Our phidgets abstract out into the following architectural units, illustrated in Figure 5. We will use the GlabServo

Figure 3: GlabPowerBar. Note the USB connection and the circuit board just visible inside

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 5, 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 CY7C63000 USB micro-controller from Cypress Semiconductor to control the on-board electronics. Our 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[3]. 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 (Figure 3).

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, 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 the string "GlabServo"), and an identification number that is unique for a phidget instance of that type (see Point 2 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 the position of one of its motors to a particular angle.

The ***PhidgetManager*** is a COM[4] object. It includes an event-based API available to end-programmers for

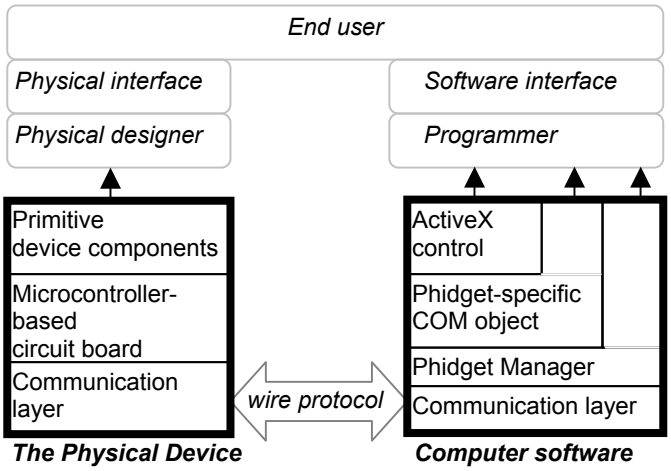


Figure 5. Phidget Architecture

connection management (see point 1 in requirements). Its major API elements include:

```
Properties:
   Count as Integer
   Item (Index as Integer)
Events:
   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 5, right side). It monitors USB devices on the system to see if they are phidgets: if they are, it creates a phidget-specific COM object (Figure 5, right side), 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 5, right side). These object 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.

---

[3] Other communication standards are possible. We previously built phidgets atop the 16F84 micro-controller from Microchip Inc. which connected to the RS-232 serial port. We are now experimenting with wireless protocols e.g., Bluetooth.

4 COM objects are Microsoft's standard way of packaging, distributing and including software modules. They have a well-defined binary interface (seen as an API) so they can be accessed from a variety of programming languages.

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. 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[5] wrap our various phidget-specific COM objects to give each of them an on-screen interface and a simulation capability (Figure 5, right). Programmers have the choice of using either these visible ActiveX controls with simulation capability or its simpler phidget-specific COM counterpart as appropriate.

[5] ActiveX controls correspond to graphical widgets, and are Microsoft's standard way of packagaing widgets by wrapping them as specialized COM objects containing a visual region that can be displayed on-screen.

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 there is no actual physical device connected to it (see Point 3 in requirements). In this case, 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 is illustrated in Figure 6, where we see 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).

While the above may sound complex, this architecture is surprisingly easy to use in practice. The next section illustrates this by example.

Example Program

The Visual Basic (VB) program in Figure 7 and shown running in Figure 6 illustrates the complete source for a toy application that controls two servo motors. The programmer used VB's interface builder to develop the interface. He dropped 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. In this example, 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.

In the code, we see how the programmer sets the motor positions and the simulation option in the `Form_Load` initialization routine. We also see how 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

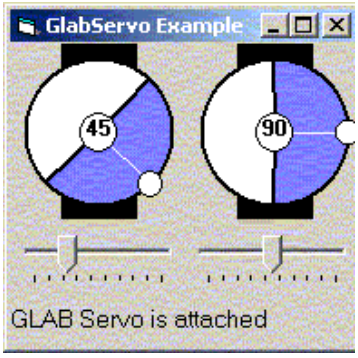


Figure 6. A screen snapshot of the example program

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
```

Other example programs of only modest complexity can let people replicate previous device-based interfaces. Natalie Jeremijenko's pioneering *dangling string*—an 8 foot plastic string that vibrates to indicate the amount of local Ethernet traffic [16]—is easily recreated using the GlabServo with a program similar to the one illustrated in Figure 7: the 'hard' part is the non-phidget code for retrieving Ethernet traffic readings. Dahley, Wisneski, and Ishii's *Pinwheels* [2]—a motorized toy fan used to broadcast events— can be quickly built atop the GlabPowerBar (to control motors that spin the pinwheels). Similarly, Heiner, Hudson and Tanaka's *information percolator*—water-filled tubes that can display patterns as bubbles [6]—can be built using the GlabPowerBar to rapidly switch the aerator pumps on and off: this is similar to how their original version was built.

```
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 7. A complete Visual Basic program for interacting with two servo motors

Architecture extensions

We can enrich the kinds of applications we build by including one more software component into our architecture: a notification server [14, 4]. Our version of the notification server implements 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 [4]. 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. For example, the application can use a GlabServo to rotate a figurine as shown in Figure 1, or turn a lamp on and off [5] with the GlabPowerBar. Our example programs for controlling and interconnecting these devices are surprisingly short and easy to code. Similarly, we could create context-aware widgets similar to [3] 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.

Finally, while the current architecture only handles USB, it 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, or even Bluetooth.

EVALUATION

As mentioned earlier, our goals behind phidgets are to provide programmers with physical devices that are:

- simple enough so that developers can concentrate on the overall use, modification and recombination of devices into a physical user interface instead of low-level device construction and implementation;
- easy enough for the average programmer to program and extend.

To evaluate if our phidgets design achieved these goals, we gave the phidgets hardware and software to computer science undergraduates taking a second course in Human Computer Interaction. Students had no prior experience building physical devices. All 16 students were given the exercise paraphrased below, worth 10% of their final grade.

> A Computer Science professor has designed a variety of phidgets that he plans to demonstrate at a conference. To make this demonstration more interesting, he would like to show how these phidgets could be used in practice. Consequently, he wants you to design an imaginative 'out of the box' interface using these phidgets. The interface you create may be practical, artistic, or fun. It could be geared towards office workers, people at home, children, or whomever you wish.

We had no formal evaluation metric except to see what students designed and whether they found it difficult to program with phidgets.

Overall results

All students successfully completed the projects. Student reported spending modest time doing their project, ranging from a few hours to a few days. All reported that most of their effort was spent in physical construction, that is, of building an interaction device or display around the phidgets (see examples below). In comparison, they reported relatively little time working on the software, and that software development was easy. Only a few students using the GlabInterfaceKit had to do some trivial electronics, where they soldered their chosen sensors or switches to connectors.

Students demonstrated their projects to the course instructor, the teaching assistant, to each other, and to several HCI graduate students. All participants were impressed by the high quality of the work and the creativity shown: demonstrations were frequently accompanied by positive exclamations ('wow', 'that is so cool', etc.) and by clapping. To illustrate what students were able to do, a few example projects are described below. These were not necessarily the best projects, but were chosen because they can be described in a print medium.

Illustrated examples

Flower in Bloom (Susannah McPhail) is a floral arrangement made out of artificial flowers The central large flower can bloom under program control from a continuum ranging from closed to fully bloomed (Figure 8). At its heart is a servo motor which controls a guideline that retracts the flower while pulling the leaves around it.

Power Dimmer / Power Lamp (Brant LeClerq). Brant wanted the ability to vary the voltage supplied to outlets, but the GlabPowerBar did not provide this. Instead, he built a 'Power Dimmer': by screwing two servo motors into two off-the-shelf rotary dimmer switches, he could rotate them under program control to vary the amount of power going

closed | partially bloomed | fully bloomed

Figure 8: Flower in Bloom

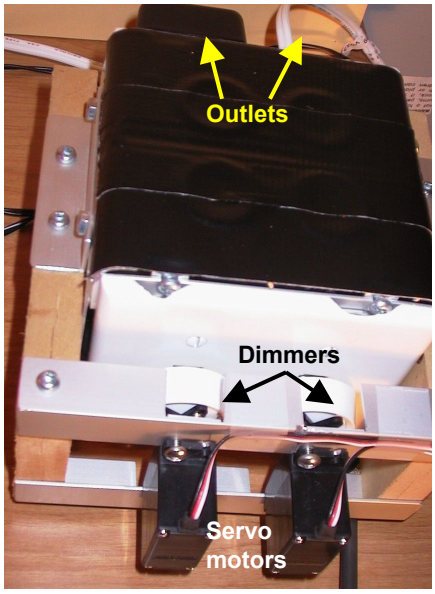


Figure 9a: Power dimmer

Figure 9b: Power lamp

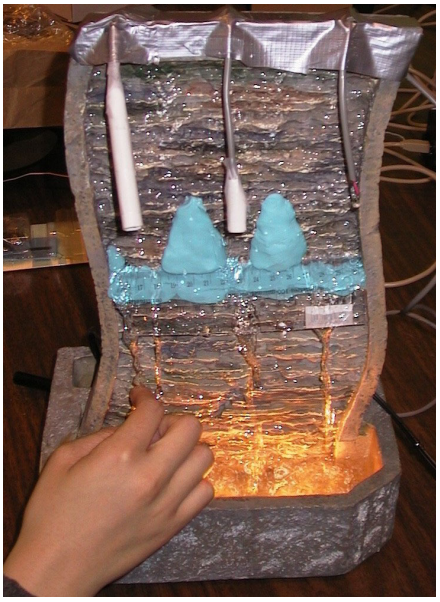


Figure 10. The Waterfall harp

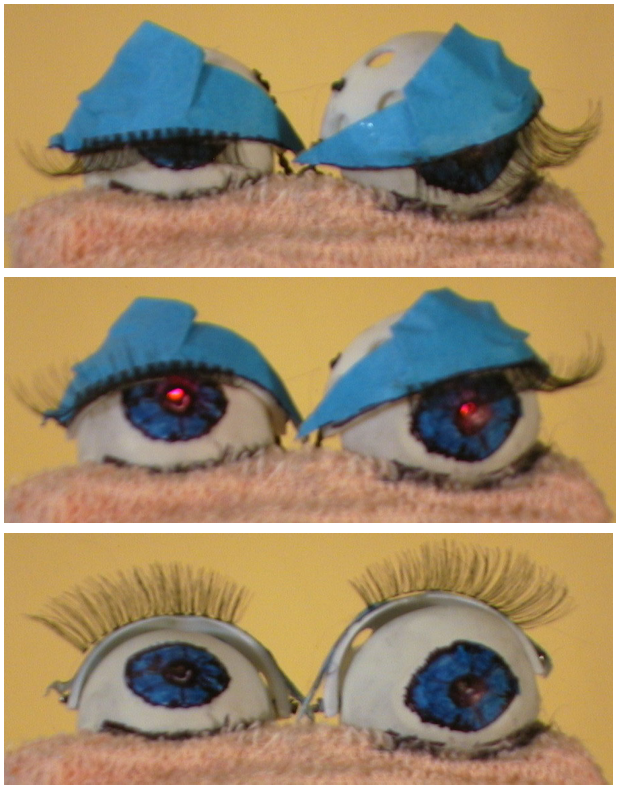
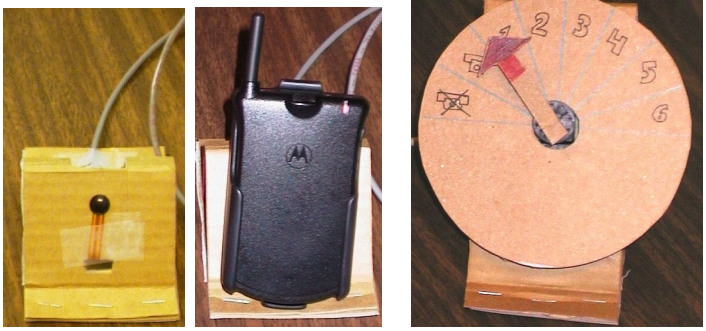


Figure 11: Phidget Eyes: closed, open & lit, fully open



a) cradle b)cradle & phone c) missed calls dial

Figure 12. Missed Calls

to two attached 120 volt outlets (Figure 9a). He then created a Power Lamp: he plugged in a fan (cannibalized from an old PC), and mounted it beneath a children's lamp containing a rotating light-shade (Figure 9b). By rotating the servos, he could adjust the intensity of the light and the fan speed, which in turn affected the rotation of the lamp shade.

Waterfall Harp (Olive Au). Olive began with a commercial appliance: it circulated water (via a small pump) to make a continuous waterfall, and had a light in its base to backlight the water. She added three light sensors (connected to a GlabInterfaceKit), and used clay to funneled water so that each sensor had a water rivulet beneath it (Figure 10). She then programmed the backend software so that a particular musical chord would play whenever a particular light sensor was blocked for a small time duration. The effect was to produce a 'waterfall harp': as people 'strummed' the water by moving their fingers through the rivulets (thus blocking the sensors), musical chords corresponding to each rivulet would play.

Phidget Eyes (Debbie Mazurek) is constructed out of ping-pong balls, fake eyelashes, string and glue (Figure 11). The eyes can open and close in any position (controlled by a servo motor), and its pupils can also light up (controlled by two LEDs connected to an interface kit).

Missed Calls (Raul Nemes). Raul wanted a device that could tell him how many calls he missed on his cell phone when he did not carry it with him. He made a cardboard cradle for his cell phone, which contained a force sensor connected to a GlabInterfaceKit (Figure 12a+b). By monitoring the readings from this senor, his software

program could detect when the phone was placed in or out of the cradle, and when the phone vibrated as it rang. His software displayed the phone status in two ways: as an icon on a Windows 2000 task bar, and as a physical cardboard dial where the dial position was controlled by a servo motor (Figure 12c). The dial showed whether the phone was in the cradle or not, and how many times a call had come in without being answered. When a person took the phone off the cradle, it would automatically reset itself.

Other examples

Several other project example are included below, but are not illustrated with figures due to lack of space. The first two use rely on the information published in the shared dictionary to track the on-line status of remote people.

Water lamp (Euan Forrester) projects patterns of light onto a ceiling to show the on-line status of up to four people. Light projects through a water tray, where 4 servo motors in different corners selectively disturbs the water. The rate of disturbance depends on online activity values of particular people. This appliance is somewhat similar to [2].

Bird in a birdcage (Shane Bill) is a cage containing a bird that would wiggle whenever a remote person tried to contact the bird's owner. Placing a cloth over the birdcage would set its owner's on-line status in the shared dictionary to 'away'. The bird was controlled by a servo, and a light sensor detected sudden darkness.

Weather-woman (Martin Fuhrer) contains two cardboard vertical gauges that indicate temperature and humidity information for any major city (automatically extracted from existing internet services). Each gauge uses a servo motor to manually lift and lower a string holding a nail, which points to the correct position on the gauge. Software includes the ability to calibrate this physical gauge.

Bubbler (David Miller) uses two cleverly combined servo motors to dip a children's bubble stick into a vat of bubble fluid, and then rotates the stick to position it in front of a fan connected to a GlabPowerBar. Bubbles then come out

Nerf Emailer (Carmen Neustaedter). Whenever email arrives, a cardboard letterbox positioned on a desk rotates to face the user's monitor: a round mail disk (made out of soft sponge) then shoots out of the letterbox opening, hits the monitor, and falls into the user's lap. One servo motor rotates the letterbox, while another pulls the trigger of a children's Nerf gun hidden inside the box.

Evaluation Summary

What should be clear from the creativity and scope of these projects is that our two goals behind our phidget design were met. We see in these examples that students did concentrate on the overall interface design instead of low-level electronic device construction, and that they were able to program, combine and extend our fairly simple phidgets in quite imaginative ways. All this re-enforces our previously stated belief: just as widgets make GUIs easy to develop, so could phidgets make the new generation of physical interfaces easy to develop.

RELATED WORK

In the introduction, we listed a few examples of physical user interfaces, yet most are built using custom one-off devices and software. While the idea of wrapping physical devices to make them easy to program and replicate is an obvious one, what is surprising to us is that we have not been able to find any systematic work on phidget design.

There are isolated instances of devices that could be characterized as phidgets. For example, the commercial Winnov Videum video camera and board (www.winnov.com) includes an SDK that wraps access to the physical camera as an easy to program device. Similarly, Kaminsky et. al.'s hacked version of Microsoft's Actimates repackages it as a graphical entity with a well-defined API [9].

A variety of hardware boards have been developed to allow programmers to experiment with robotics and embedded control applications. An excellent example is the Handy Board [11]. This is a hand-held, battery-powered micro controller that came out of MIT laboratories. Through it, a programmer can control a variety of raw devices e.g., 9 digital inputs, 7 analog inputs, IR output and indicators, an LCD screen, a piezo beeper, 4 DC motor outputs, and so on. While resembling some of our physical devices, the Handy Board is not a phidget. Its target is personal and educational robotics projects. Typically, its programmers develop a program that can be downloaded to the micro-processor, after which the board runs autonomously. That is, it does not behave as a phidget because the right side of the phidget architecture in Figure 5 is missing. However, we suspect that the Handy Board (or some version of it) could be fashioned into a supercharged equivalent to the GlabInterfaceKit device: this should be straightforward when a USB version of a Handy Board becomes available.

The Context Toolkit [3] leverages the notion of a widget to create *context widgets*. Context widgets gather contextual information from several sources, then abstracts and makes this information available to the programmer. One source of contextual information may come from actual physical devices (which they call generators). However, these are internally defined and not really exposed as part of the API. For example, their Activity widget API contains attributes giving location, timestamp, and an abstracted activity level, as well as a callback whenever activity changes [3]. Its actual generator is based on a microphone, but could have been implemented with other information generators, such as infrared sensors, video image analysis, etc. A context widget may also contain several generators, and it may combine and abstract the information collected from these generators. However, the context toolkit does not facilitate building these generators. In comparison, our phidgets are at this generator level, and could (at least in principle) be

used to simplify how actual devices are incorporated within a context widget.

In essence, some previous work is at a lower level of abstraction than ours (e.g., the Handy Board), and some of it is at a higher level (e.g., the context widgets). While isolated examples resembling phidgets do occur, there has been no systematic description of phidget architectures or design rationales.

CONCLUSIONS

Our main message is that packaging devices as physical widgets or *phidgets* greatly simplifies programming these devices, which in turn allows designers to concentrate on how physical user interfaces can be crafted *vs.* low-level implementation details. Of course, this is not a revolutionary idea: we suspect that existing practitioners have already packaged their own devices for internal reuse. We are surprised, however, that there has been no real push to publish, standardize and even to commercialize devices as phidgets. Yet there is a real need for this: almost all the people we have talked to who developed systems based on physical devices—researchers, developers, artists—had to start from scratch.

There is much left to do. We need to evolve a *de facto* standard phidget set. This already exists for GUI toolkits; for example, virtually all sets include various buttons, list boxes, menus, text boxes and so on. However, it is unclear what phidgets would be included in a standard phidget set. Likely candidates include the ones we built and those suggested by devices included in other physical user interfaces, but there are likely many more. As with GUI widgets, this phidget set must provide the programmer with conceptual building blocks that are not only individually useful, but can be assembled in a way that lets the designer build a rich physical interface.

Software and hardware availability. Phidget software is available from www.cpsc.ucalgary.ca/grouplab/. We are still deciding on how we will distribute hardware and schematics (e.g., commercialization, licensing or freeware). Contact the authors for further information.

Acknowledgements. We are indebted to the creative students in our undergraduate course. They graciously allowed their projects to be included as examples of what could be done with phidgets. Michael Boyle contributed greatly to our discussions about the phidget architecture. We thank our collaborator Professor Hideaki Kuzuoka (University Tsukuba), whose visit started all this going. The Collaboration Group at Microsoft Research, the National Sciences and Engineering Research Council of Canada (NSERC), and the Alberta Software Engineering Research Consortium (ASERC) partially funded this work.

REFERENCES

1. Ark. W. and Selker, T. A look at human interaction with pervasive computers. *IBM Systems Journal* 38(4), 1999.
2. Dahley, A., Wisneski, C. and Ishii, H. Water Lamp and Pinwheels: Ambient projection of digital information into architectural space. *Summary of CHI '98*, 269-270, 1998.
3. Dey, A. K., Salber, D., and Abowd, G. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction*, Vol 16, 2001.
4. Greenberg, S. and Kuzuoka, H. Using digital but physical surrogates to mediate awareness, communication and privacy in media spaces. *Personal Technologies* 4(1), January, Elsevier. 2000.
5. Gruen, D., Rohall, S., Petigara, N. and Lam, D. "In your space" displays for casual awareness. Demonstration at *ACM CSCW*, 2000.
6. Heiner, J., Hudson, S. and Tanaka, K. The information percolator: Ambient information display in a decorative object. *Proc. Symposium on User Interface Software and Technology (ACM UIST'99),* 141-148, 1999.
7. Ishii, H. and Ullmer, B. Tangible bits: Towards seamless interfaces between people, bits and atoms. *Proc. Human Factors in Computing Systems (ACM CHI'97),* 234-241, 1997.
8. Ishii, H., Mazalek, A., Lee, J. Bottles as a minimal interface to access digital information. *Extended Abstracts of Conference on Human Factors in Computing Systems (ACM CHI '2001)*, 2001.
9. Kaminsky, M., Dourish, P., Edwards, K. LaMarca, A., Salisbury, M. and Smith, I. SWEETPEA: Software tools for programmable embodied agents. *Proc. Human Factors in Computing System (ACM CHI 99),* 144-151, 1999.
10. Knudsen, J. (1999) *The Unofficial Guide to LEGO Mindstorms Robots*. O'Reilly Press.
11. Martin, F. *The Handy Board Technical Reference.* www.handyboard.com, 2000.
12. Myers, B. State of the art in user interface software tools. In Baecker, R., Grudin, J. Buxton, W. and Greenberg, S. *Reading in Human Computer Interaction: Towards the Year 2000.* Morgan Kaufmann. 1995.
13. Norman, D.A. *The Invisible Computer*. MIT Press, 1998.
14. Patterson, J., Day, M. and Kucan, J. Notification servers for synchronous groupware. *Proc. Computer-Supported Cooperative Work (ACM CSCW'96)*, 122-129. 1996.
15. Ullmer, B., Ishii, H. and Glas, D. mediaBlocks: Physical containers, transports, and controls for online media. *Proc. 25th Annual Conference on Computer Graphics.* 1999.
16. Weiser, M. and Brown, J. Designing calm technology, *Powergrid Journal,* v1.01, July, 1996.
17. Yarin, P., and Ishii, H., TouchCounters: Designing interactive electronic labels for physical containers. *Proc. Human Factors in Computing Systems (ACM CHI '99),* 362-369, 1999.