Shared Phidgets: A Toolkit for Rapidly Prototyping Distributed Physical User Interfaces

Nicolai Marquardt
Faculty of Media
Bauhaus-University Weimar
Bauhausstr. 11, 99423 Weimar, Germany
nicolai.marquardt@medien.uni-weimar.de

Saul Greenberg
Department of Computer Science
University of Calgary
Calgary, Alberta CANADA T2N 1N4
saul.greenberg@ucalgary.ca

ABSTRACT
Many physical user interfaces are best viewed as an interacting collection of remotely-located distributed hardware and software components. The problem is that current physical user interface toolkits do not normally offer distributed systems capabilities, leaving developers with extra burdens such as device discovery and management, low-level hardware access, and networking. Our solution is *Shared Phidgets*, a toolkit for rapidly prototyping distributed physical interfaces. This toolkit offers programmers several ways to easily access remotely located hardware components, including a powerful distributed model-view-controller object model. Network communication and low-level access to the device hardware are transparently handled, regardless of device location. The programmer can also create new abstract devices by transforming and aggregating low level hardware device capabilities.

ACM Classification: H.5.2 [Information Interfaces]: User interfaces – input devices and strategies, interaction styles, prototyping, user-centered design

General terms: Design, Human Factors

Keywords: Distributed physical user interfaces, toolkits, prototyping, hardware-software integration, Phidgets.

INTRODUCTION
Physical user interfaces are an increasingly important part of many emerging visions of human computer interaction: ubiquitous computing and calm technologies [24], pervasive computing [2], tangible user interfaces [18], information appliances [21], ubiquitous media and reactive environments [5], interactive art, ambient displays [7], and context-aware computing [8]. In most of these visions, physical user interfaces comprise some type of appliance constructed from simple hardware devices – sensors, switches, actuators, displays, motors, RFID tags and readers – that developers package in some manner, connect to, monitor, and control via software. These appliances are either carried by people or deployed at meaningful locations within the end user's everyday environment, with the idea that they work within (rather than apart from) the everyday practices of people [9].

While some physical user interfaces can act as a standalone 'disconnected' appliance, they are usually envisaged as components interacting within a network of other appliances and computers. As Dey et. al. state in their discussion of context-aware applications: "…the devices used to sense context most likely are not attached to the same computer running an application that will react to that context" [8]. That is, they become a *distributed physical user interface* comprising a variety of hardware devices connected to *different computers* over *multiple locations*, all networked together, e.g., [13,21,8].

From this perspective, a distributed physical user interface is best viewed as hardware nodes on a distributed system. This viewpoint reveals that developers of such systems face two considerable challenges.

1. They must program, communicate with and control low-level hardware devices.
2. They must assume the additional programming burden inherent in most distributed systems: resource discovery, network communication and protocol development, contention control, managing failures due to connectivity problems and latency, debugging intricacies of distributed systems, and so on.

Toolkits are now available that simplify device programming, or that simplify distributed systems development. Yet none do both (see §Related Work). Consequently, we set ourselves the goal of designing a toolkit that would allow a programmer easy access to the many distributed devices that comprised a network of physical user interfaces. In particular, we wanted a toolkit that:

- automatically discovers devices connected to a myriad of different computers;
- manages all network aspects so that no network programming is required;
- uses the same API to control a device, regardless of whether it is attached to a local or a remote computer;
- uses a distributed Model-View-Controller (dMVC) design pattern to represent every device so that data associated with the model is easily queried and manipulated;

Cite as:
Marquardt, N. and Greenberg, S. (2006) **Shared Phidgets: A Toolkit for Rapidly Prototyping Distributed Physical User Interfaces.** Report 2006-829-22, Department of Computer Science, University of Calgary, Calgary, Alberta, Canada, T2N 1N4. April.

- generates notifications across the network whenever device state is changed;
- offers graphical 'skins' that let a person view local and distant device state and control them via a conventional GUI;
- offers the means to create 'abstract devices' that transform and aggregate low level hardware device capabilities into higher level abstractions; and
- provides a set of high-level tools to: manage all the necessary hardware and network connections, control and observe devices, examine the state of the MVC model, and emulate devices for testing purposes.

Our solution is the Shared Phidgets toolkit, a significant software extension of the commercial Phidgets platform [22]. To forewarn the reader, this paper concentrates on the technical infrastructure underlying Shared Phidgets. This reflects our belief that the toolkit offerings and its architectural makeup are substantial contributions that tremendously simplify the development of distributed physical user interfaces. After briefly summarizing related work, we provide a step by step scenario showing exactly how a programmer would use Shared Phidgets to create a simple distributed awareness appliance. We then detail the Shared Phidgets architecture, and describe how this architecture is used by developers.

RELATED WORK

Until recently, physical user interface development was restricted to the few programmers who knew about hardware and circuit design, and who were willing to do a huge amount of back-end programming: firmware, networking and protocol development to link hardware and computers, and the application software that monitors and uses the device. Fortunately, the advent of various toolkits has made rapid physical user interface development reachable by average programmers. This includes research products (e.g., iStuff [3], Input Configurator [10], d.tools [17], Calder [19]) and well-developed commercial offerings (e.g., Phidgets, Inc. [22,14], and MakingThings [20]). Although offerings vary, they typically provide hardware devices with well-defined functionality, easy connectivity (including wireless) between the device and a single traditional computer, and a software API that lets a programmer access the hardware functionality. Consequently, developers can focus on high level design of physical user interfaces rather than on low level implementation details (see also [1,12]).

Perhaps the most widely used of these toolkits is Phidgets, first created as a research system [14], and then commercialized by Phidgets Inc [22]. Phidgets include USB-based hardware boards controlling various input sensors (e.g., temperature, movement, proximity, light intensity, RFID tags) and output actuators (e.g., servo motors, LED indicators, LC text displays). Phidgets also provide a comprehensive architecture and API to discover, control and observe all phidgets connected to a single computer.

While all the above toolkits simplify hardware programming, they do nothing to help one manage hardware as a distributed system. Dey et. al.'s Context Toolkit [8] is the exception. Their toolkit has several important components. C*ontext widgets* encapsulate and abstract the actual (possibly distributed) devices and software used to collect contextual information. *Interpreters* transform this low level information into high level abstractions. *Aggregators* collect, group and logically relate multiple pieces of information. *Services* use any of the above input components to control something, i.e., to perform an output. *Discoverers* maintain a registry of all the above components and their capabilities. Under the covers, all components can communicate as a distributed system by using a subscription-based event system built atop of TCP [8].

Yet the Context Toolkit does not actually ease how programmers compose low level hardware devices. Our understanding is that a context widget's connection to actual hardware (including hardware control) has to be custom coded; the toolkit itself supplies no support for this difficult step. That is, the toolkit begins with the abstracted 'context widget' but does not explicitly support how these are created from hardware. Thus there is a significant gap between how one actually accesses the hardware (as provided by the previously mentioned toolkits) *vs.* how one leverages this hardware in a distributed setting (as in the Context Toolkit).

This gap is the 'sweet spot' that our Shared Phidgets toolkit addresses. As we will see, we extend the existing Phidgets architecture so that programmers can access low-level hardware devices located anywhere on the network, and yet compose them in ways that provides capabilites somewhat comparable to the Context Toolkit's offerings.

PROGRAMMING SCENARIO

To set the scene, we illustrate how 'Jim' uses Shared Phidgets to create an awareness appliance that lets a person at home know if his working spouse is present, around, or absent from her office. The appliance comprises three linked devices distributed across two locations: the home and the office. While simple, this appliance implements previously published ideas, including the Door Mouse [5], Physical but Digital Surrogates [15], and Aggregates [8].

Description

The office part (not illustrated) comprises two off-the-shelf sensors attached to a Phidget InterfaceKit circuit board [22] plugged into the 'office computer. A proximity sensor detects if someone is seated at the desk, while a force sensor detects if the office door is closed. Software aggregates these two sensor values into a new 'availability' value:

- *present:* door open, someone seated;
- *around:* door open, no one is seated;
- *absent:* door closed, seated state ignored.

The home part, illustrated unadorned in Figure 1, contains a Phidget TextLCD display, and a figurine glued to a Phidget Servo. Both are plugged into the 'office computer. It also contains a graphical user interface (Figure 1, middle) mirroring the state of these devices. The LCD display contents and the figurine's position depend on availability state:

- *present:* faces forward (0^o), says 'Present'
- *around:* faces sideways (90^o), says 'Somewhere Around'
- *absent:* faces the wall (180^o), says 'Unavailable'.

Implementation

The steps below assume that a server is up and running, located at (say) tcp://demo.ca:test, and that the special *Connector* tool (described later) is running on each computer. Both come with the toolkit, and Jim starts them in seconds.

Jim first works on the office sensors. He positions the proximity sensor in front of the desk chair, tapes the force sensor to the inside of the door jamb, and plugs it into the Interface Kit. This takes a few minutes (the most difficult job is hiding the wires). The Connector detects the InterfaceKit as soon as it is plugged into the office computer, and immediately publishes its sensor data to the server. This data is now available to other software that connects to this server.

Jim then builds the home part of this device illustrated in Figure 1, which just requires a bit of gluing. He then writes the small program (Figure 1 bottom) that monitors the two distant sensors and uses its values to determine the servo position and the LCD display contents. The steps below outline this process; Jim does the first three with an interface builder, and writes code only in the last step.

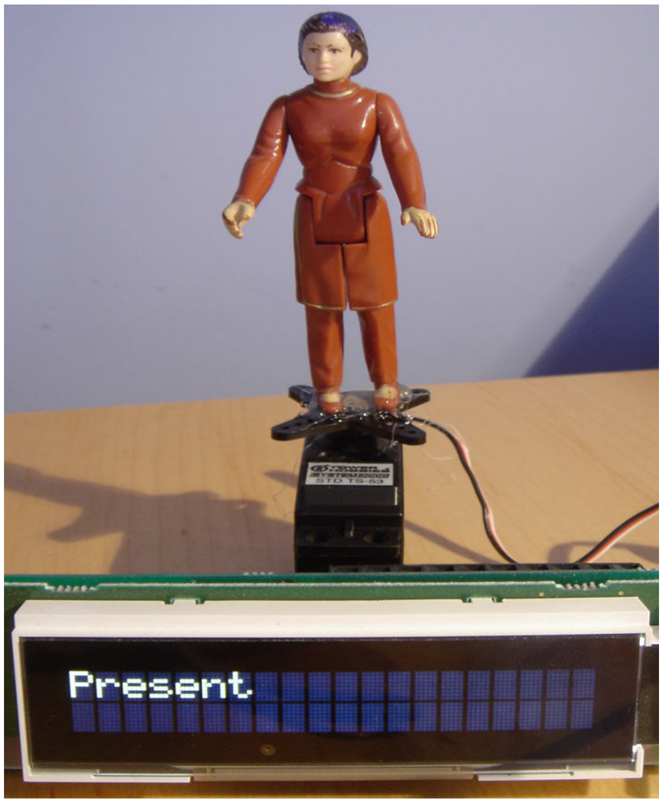
Connect to the server. Drag and drop the Shared Phidgets ConnectionManager object onto the window form, and set its SharedDictionary property to "tcp://demo.ca:test. This object automatically connects to the central server.

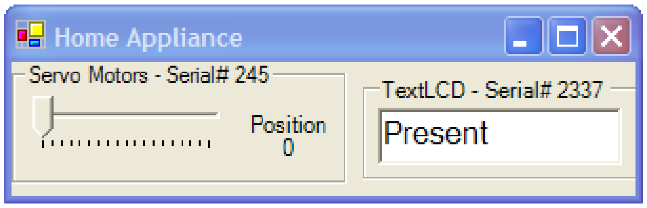
Create an object connected to the distant InterfaceKit. Drag and drop an InterfaceKit object onto the form. Set its FilterLocations property to "Office". This act automatically queries the 'Locations' metadata field associated with all attached InterfaceKits, finds the one in the office, and links the software with the distant hardware.

Create objects that control and graphical display the local appliance. Drag, drop and link a Servo and ServoSkin object, and a TextLCD and TextLCDSkin object into the window form. Set their serial numbers to match the local hardware. These 'Skins' are graphical interfaces that reveals the state of the servo and the display hardware; while not strictly necessary, this graphical view is included for illustrative purposes.

Monitor the sensor values to control the local appliance. Create an event handler for the interfaceKit's SensorChange event to monitor the current values of these sensors. Jim also creates a utility method Aggregate to aggregate the sensor values, and uses this aggregate to control the home appliance. Figure 1 shows this code.

He compiles and runs this program on the home computer. It automatically connects to the server over the internet, and raises events as sensor data collected from the office




Home Appliance

Servo Motors - Serial# 245

Position
0

TextLCD - Serial# 2337

Present

```
// Enumerate three availability states; labels for LCD
enum Availability {Present = 0, Around = 1, Absent = 2 };
string [] labels = new string [3] {
      "Present", "Around", "Absent"};
//New sensor values received; update the appliance
private void iK_SensorChange(…){
  int status = Aggregate(iK.Sensors[0].Value,
                         iK.Sensors[1].Value);
  textLCD.Display = labels [status];
  servo.Motors[0].Position = 90 *  status;
}
// Aggregate the sensor values into an availability state
private int Aggregate (int sensor0, int sensor1) {
  bool door = (sensor0 < 50);   // closed, if force < 50
  bool seated = (sensor1 < 300);// seated, proximity < 300
  if (door && seated) return (Availability.Present);
  else if (door && !seated) return (Availability.Around);
  else return (Availability.Absent);}
```

Figure 1. The home appliance and its graphical interface

computer changes. The event handler code in Figure 1 is invoked and the home appliance is adjusted accordingly.

Discussion

The above example is notable in that its programming complexity (or simplicity) is almost identical to that of the original non-distributed Phidgets [14]. Aside from starting the server and the Connector programs, the only coding difference is that the programmer had to include a ConnectionManager and an address to the server. All distributed systems aspects were otherwise hidden. We stress that this example only shows the most basic use of SharedPhidgets; much more sophisticated and nuanced appliance designs

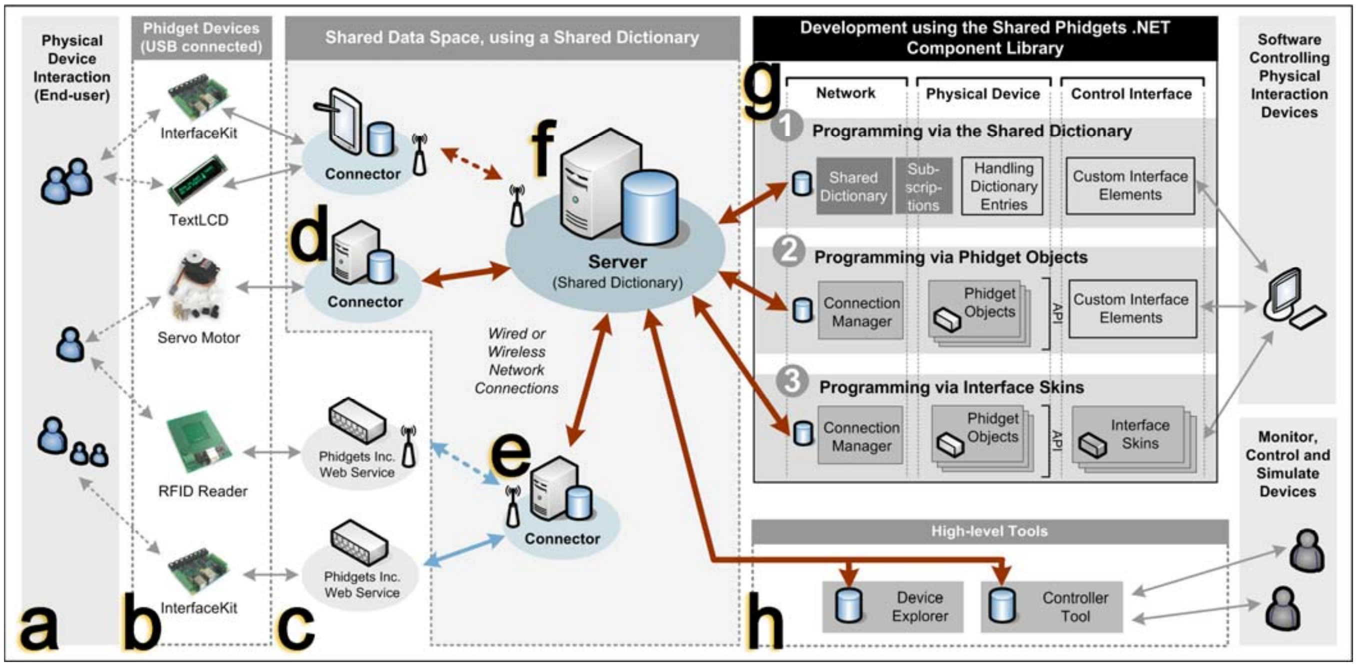

Physical Device Interaction (End-user)
Phidget Devices (USB connected)
Shared Data Space, using a Shared Dictionary
Development using the Shared Phidgets .NET Component Library
Software Controlling Physical Interaction Devices
InterfaceKit
TextLCD
Servo Motor
RFID Reader
InterfaceKit
Connector
Connector
Server (Shared Dictionary)
Wired or Wireless Network Connections
Phidgets Inc. Web Service
Phidgets Inc. Web Service
Connector
Network
Physical Device
Control Interface
1 Programming via the Shared Dictionary
Shared Dictionary
Sub-scrip-tions
Handling Dictionary Entries
Custom Interface Elements
2 Programming via Phidget Objects
Connection Manager
Phidget Objects
API
Custom Interface Elements
3 Programming via Interface Skins
Connection Manager
Phidget Objects
API
Interface Skins
Monitor, Control and Simulate Devices
High-level Tools
Device Explorer
Controller Tool
a
b
c
d
e
f
g
h

Figure 2. An overview of the Shared Phidgets Architecture

are possible, as illustrated in later sections. What is especially important is that this example reveals that programmers can not only access devices as in other hardware toolkits, but can trivially treat them as higher level 'context widgets' as is done in the Context Toolkit [8].

SHARED PHIDGETS ARCHITECTURE

The Shared Phidget architecture and its other offerings are illustrated in Figure 2. This section concentrates on what happens 'under the covers': its primary hardware and software components and the interactions between them, Subsequent sections describe what application developers actually see and the tools they use to facilitate their programming process.

Phidget Hardware and Devices

Phidget devices [14,22] are the combined hardware/circuit board building blocks exploited by our infrastructure (Figure 2b). These devices are used by developers to create the physical portion of their interface, which in turn defines the end user's interaction with the physical device (Figure 2a). Various hardware components are now available from Phidgets Inc. These include *input sensing* (motion, touch, temperature, proximity, magnetic force, light), *input controls* manually activated by people (switches, dials, sliders, joy sticks, key fobs, accelerometers, RFID readers), *output actuators* **(**motors, servos, solenoids, valves) and output *displays* (lights, text and graphics displays).

Computer Communication to Phidget Devices

Phidget devices interact with a controlling host computer, and thus need to be connected to them. Currently, all Phidget devices are connected to a host computer via USB.

Phidget Inc. supplies two rudimentary interfaces to let programmers communicate with these devices. First, a dynamic link library offers an API to access all locally attached Phidget devices. Second, an (inappropriately named) 'web service', currently in beta, provides a socket-based interface to the local machine's phidgets (Figure 2c). While this second form can be exploited as a crude network service, it was actually developed as a platform-independent interface that simplifies access to Phidget device capabilities across different programming languages and operating systems.

Because multiple devices can be plugged into a single computer, the controlling computer needs a way to differentiate between them. To do this, each Phidget device returns its type (e.g., phidgetServo, phidgetRfid), and each device of a particular type returns a unique serial number. This means that the type / name combination uniquely identifies each connected device.

So far, we have described the offerings of Phidget Inc., which closely matches the original Phidget architecture [14]. Remaining sections depart radically from Phidget Inc. offerings. As we will see, our new distributed architecture provides a *shared distributed data space* that contains information about all Phidget devices regardless of their location, and a *connector* mechanism that hooks devices on the local computer to this shared data structure.

Shared Distributed Data Space

A fundamental component of the Shared Phidgets architecture is a shared data space implemented as a distributed data structure. We use the *shared dictionary* provided by the Grouplab .NETWORKING toolkit [4].

.NETWORKING allows client processes read and write access to a collection of data objects maintained within the shared dictionary server (shown at Figure 2f). This also works as a *notification server*, where clients are immediately notified of any changes to data they are subscribed to, regardless of who made these changes. .NETWORKING also automatically manages all runtime networking housekeep-

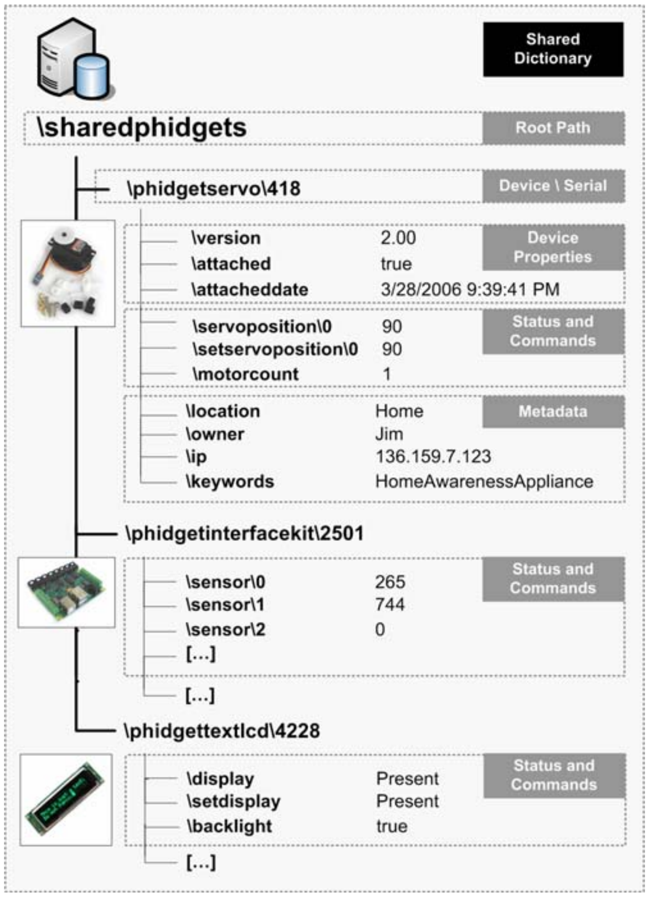


Figure 3. Path structure in the Shared Dictionary

ing tasks, i.e., socket creation/teardown, wire protocol, data marshalling, parsing, etc.

Data in the shared dictionary is structured as hierarchical key/value pairs. Values can be primitive data such as integers and strings, binary and multimedia data, or complex data such as lists and structures. A key is expressed as a path hierarchy. A rich set of operators allow programmers to subscribe and iterate over data held in particular subpaths of this hierarchy. Further details about how this shared dictionary works can be found in [4].

In Shared Phidgets, all participating machines access this data via a *distributed Model-View-Controller* (dMVC) pattern [16]. The *model* is the abstract data stored on the shared dictionary. There are multiple *controllers* – the client machines – that can change values in the shared data model. Similarly, when multiple clients receive notifications of changes of that data they can each generate their own *view* of it.

Every phidget device has an entry in the shared dictionary that completely defines its state. Using our office/home appliance example, Figure 3 shows the partial dictionary entry for its three devices: the Phidget Servo, InterfaceKit and TextLCD. To illustrate the dictionary's hierarchical nature, the key defining the position of servo motor 0 is \sharedphidgets\phidgetservo\418\servoposition\0. This key path specifies the root (shared phidgets), the device type (phidgetservo), its serial number (418), and the position and motor number attribute. Thus given a serial number of a Phidget Servo, it is easy to search for it, and to modify or iterate through all its properties and values. For the InterfaceKit, we see the values of sensor 0 (the door pressure sensor) and sensor 1 (the proximity sensor). We also see that the textLCD is currently displaying the text "Present".

As a person triggers the proximity sensor by sitting down at their office chair, the controller generates a new value for that sensor in the underlying model (...\sensor\1 in Figure 3). A notification is generated, and as a consequence the home client updates the view of the appliance. This actually transforms that notification into two other controller actions. First, it resets ...\setservoposition\0 to the new value, which in turn changes the servo view in the graphical user interface and performs the actual rotation. Second, it resets the ...\setdisplay to the new text, which then appears on both the physical textLCD display and its GUI counterpart.

While we could go on at length about the details of this part of the architecture, the critical point is that the Shared Phidgets architecture is realized primarily as a dMVC pattern over a distributed client / server shared data model with updates indicated via a notification server. This greatly simplifies the internals of distributed data management and, as we will see later, the way we can create appropriate APIs for the programmer.

Connectors and Phidget Proxies

The shared data space maintains a runtime model of all the shared phidgets, but does not define how the local machine (and consequently its local phidgets) connects to it. This is the job of the *Connector*, illustrated at Figures 2d and 2e. The Connector runs quietly in the background on each local client machine. It notices any locally-connected Phidgets that are plugged in over time. As it finds a new device, the Connector dynamically adds an appropriate *phidget proxy object* to handle it.

This proxy object has two responsibilities. First, it observes and controls its specific Phidget device features, e.g. an interface kit object observes all sensor values generated by it; a servo object controls a servo's position; a text display object controls changes to the displayed message. Second, it serves as an intermediary between the Phidget device and the shared dictionary model. For each connected Phidget device, the object creates key/value pairs matching the Phidget's attributes in the shared dictionary. It then mediates between the two to make sure that both data and device reflect the same state. Acting as a controller, it monitors the Phidget device for any changes, and updates the shared dictionary model to reflect those. For example, a reported sensor value will update its corresponding data model entry. At the same time it acts as a view, where it monitors the shared dictionary model (via notifications) for any data updates, and adjusts the Phidget device to reflect that new value. For example, a change in a servo position data will translate to the servo motor actually rotating to that position.

Revisiting Figure 3, we now see that it illustrates how three connected Phidget devices have matching entries in the shared dictionary; excluding timing delays, these represent the properties and current state of the hardware. All paths begin with the \sharedphidgets\ root element, followed by the device name (e.g. phidgetservo\ and phidgetinterfacekit\) and its serial number (e.g., 418 for the servo, 2501 for the interfaceKit). The next part of the path corresponds to the particular attributes of the unique Phidget device. These attributes can be divided into the following three categories.

General device properties provide information common to all phidgets: the version number, whether this device is currently attached to the local computer, and date stamp.

Current device status represent the available input and output functions of the Phidget. Each sensor or control input involves a separate dictionary entry, and the Connector tool is responsible for controlling (updating) these values as the hardware triggers updates to them as events. An example is illustrated by the values shown for the sensor inputs of the InterfaceKit. Each actuator or display output is represented by two entries: one with the current value of the output, and one for submitting requests to change the value (e.g., the servo's servoposition vs. setservoposition in Figure 3). These two entries are important: the set entry is what the output should be, and this is in turn used by the Connector to direct the hardware until that value is achieved in the corresponding entry (depending on the device, this could be near-instantaneous or take several seconds). Multiple working threads ensure the rapid forwarding of all these controlling commands, and execute the time-sensitive changes without blocking the system.

Metadata entries contain additional information describing each device. This includes the IP of the actual computer that is connected to the Phidget device, its physical location, its owner, and keywords. The actual metadata is specified through the local machine's Connector interface. This information is stored in an XML file and is automatically added to the shared dictionary entries of each connected Phidget. For example, we see that the servo in Figure 3 is located at 'Home', that it is owned by 'Jim', and that it is part of an information appliance called 'HomeAwarenessAppliance'. In our original example, we used the 'Location' metadata to find the InterfaceKit in the distant office. While some metadata fields are always provided, the end user can fill in a table within the Connector to add one or more custom fields and values and associate these with a particular Phidget device. These custom metadata entries are then stored within that phidget's data model in the shared dictionary. For example, the three phidgets comprising our example appliance may be viewed as a single appliance by creating a metadata key 'ApplianceType' and setting its value to 'Jims Awareness Appliance'. Another metadata key 'Where' can indicate if it is the office or home side by setting its value to 'Office' or 'Home'. Because metadata information originates in the local machine's Connector, the metadata information is updated if and when users move devices between computers (e.g., new location, owner…). That is, the metadata information can offer context-dependant information that can be exploited by the programmer.

Internally, the Connector accesses either the Phidgets Inc. dll (as at Figure 2d), or it opens connections to one or more Phidgets Inc. web services (as at Figure 2e). Through this web service connection, our Connector can serve as an intermediary to other computers hosting standard Phidget devices but not running our SharedPhidgets software (Figure 2e). This is important for it gives platform-independence: while SharedPhidgets currently runs only on Windows, we can still connect to Apple MacIntosh and Linux boxes hosting phidgets.

In summary, the Connector mediates between the data *model* and Phidgets discovered on the local machine or through web services. As a *view* onto the model, the connector commands physical hardware to reflect data state changes made from other client applications. As a *controller*, it transforms state information of the physical widget into changes to the data. As a local data store, it adds *metadata* that identifies particular features of that device as it relates to its local installation.

Finally, all this happens without any user or programmer intervention. In practice, the Connector is automatically started on login, and runs in the background. If desired, end users can raise a *Connector GUI*, through which they can monitor the status of all shared devices and their properties, and where they can add and/or alter the metadata.

DEVELOPER TOOLKIT LIBRARY

The dMVC model considerably simplifies the job of sharing and manipulating Phidget devices across a network and across multiple machines. Yet we recognize that the dMVC pattern may be unfamiliar to average programmers. Consequently, our architecture offers a three-tier programming model, illustrated at Figure 2g and separately described below, that lets developers choose (and intermix) three different development strategies that balances power and simplicity. Regardless of the strategy, standard development environments and programming languages are used. We stress that each strategy trivializes hardware access and network communication, thus letting developers focus on their physical interface design ideas.

Programming via the Shared Dictionary

Programmers can develop Phidget applications by addressing the shared dictionary directly (Figure 2g, row 1). While this adds power, programmers need to know the API to the .NETWORKING shared dictionary (e.g., subscription management and data organization), regular expressions (for pattern matching) and be familiar with the dMVC pattern. With subscription objects, developers receive notification events of changes to entries in a sub-tree of the dictionary hierarchy. They would also write custom code as event handlers (callbacks) that take action on changes to these dictionary entries.

In spite of the extra work, the shared dictionary programming model is very powerful as developers profit from pattern matching via path wildcards. That is, they can easily iterate through and control many devices at the same time. For example, the single line below returns all motors on all Phidget devices on all computers, where the '?' is a regular expression that generates matches for a single hierarchical level.

```
/sharedphidgets/phidgetservo/?/setservoposition/?/
```

By embedding this line in a foreach statement, a programmer can iterate over this expression to (say) reset all motors of all Phidget Servos to 0 degrees.

The dictionary can also be used to create new *'abstract devices'* that monitor, transform, and aggregating low level hardware device values into new data entries. As an example, reconsider our office/home appliance of Figure 1. In that system, the home component created and used the Availability aggregate. In practice, this should be done by the office component; this would let us (say) add new sensors to more accurately infer presence without changing the home side of the appliance. Similar to the code in Figure 1, the office side could calculate availability state. It then stores this aggregated sensor value into a new shared dictionary SD entry that models an abstract device called 'officepresence:

```
SD ("/sharedphidgets/officepresence/1/availability") = "Present"
```

The home appliance can then subscribe to this abstract device instead of the InterfaceKit, where it monitor this key's value to control the figurine position and LCD display.

Programming via Phidget Objects

In cases where the power of the Shared Dictionary is not required, the programmer can develop Phidget applications by a simpler object-oriented API that completely encapsulates individual Phidget device capabilities. This is what was done for coding the home part of the appliance shown in Figure 1. As illustrated in Figure 2g row 2, and as previously mentioned, the Connection Manager provides a *phidget proxy object* matching each Phidget device. This proxy reads and modifies that Phidget's entries in the shared dictionary according to a particular device, and provides an object-oriented properties/methods API for the developer. A programmer controls a device by altering its properties and/or invoking its methods, and monitors changes to device status by adding event handlers. Networking and distributed data sharing aspects are completely hidden. Not only is this simpler than directly accessing shared dictionary, but this familiar programming model requires little extra learning as it matches conventional GUI object-oriented programming and the original Phidgets programming paradigm [14]. Of course, the actual Phidget devices being controlled may now be located anywhere on the network. This is not a problem, as the programmer can quickly link the Phidget object to the actual Phidget device by filling in two properties: the hosting computer's metadata location (as used in our original example) and/or its serial number. Of course, some power is sacrificed. For example, the 'abstract device' that reflects an aggregate of different sensor values cannot be created through this object-oriented API. However, using the shared dictionary the programmer can create a new Phidget proxy object that encapsulates the abstract device, and then make that new object available to other programmers.

Programming using Metadata and Device Discovery

As a brief but important aside, this section reveals how programmers can exploit metadata and discover devices.

Metadata is accessed via the shared dictionary or the Phidget object in a manner similar to other phidgets properties. For example, one can access the servo's location and keywords (Figure 3) through the Servo Object's properties:

```
String location    = servo.PhidgetDevice.Location;
String keywords = servo.PhidgetDevice.Keywords;
```

Programmers can also discover devices across the network by a variety of means. The easiest is filtering: filter terms are added to a Phidget object, and the device whose metadata matches those terms are attached to it. For example, the following code will discover the InterfaceKit used in our example and in Figure 3 by matching its Location and its Owner properties (other filter properties include serial numbers and IP addresses):

```
interfaceKit.FilterLocations.Add("Home");
interfaceKit.FilterOwner.Add("Jim");
```

Custom metadata can be accessed, exploited and even reset via a hashtable associated with a Phidget Object. Programmers can iterate over all its metadata entries to look for matches, or see if a particular one exists through a ContainsKey() method, or get a particular value through the GetMetadata("key") method. A programmer can even add metadata at runtime. For example, if a programmer had access to GPS location, that can be added as a metadata on the fly:

```
servo.PhidgetDevice.AddMetadata("GPSPosition",
                                    "N-51-02-11:W-114-01-10");
```

These user-defined metadata entries become immediately visible in Connector tool, and other connected applications can explore this information by iterating over the metadata collection.

Programming via Interface Skins

Some physical user interfaces also contain a GUI counterpart that allows people to monitor and control Phidget devices, perhaps from remote locations. While these can be programmed from scratch atop of Phidget Objects, programmers will typically use *interface skins*.

We already illustrated interface skins in our home appliance. For example, the skin for the Phidget Servo illustrated in Figure 1 shows that the servo phidget is attached, and the position of its single motor. The end user can also use its slider to reposition this motor. Figure 1 also shows the skin for the TextLCD, while Figure 5 shows the skin for the InterfaceKit.

Interface skins are wrapper objects around the Phidget proxy object API. They normally provide GUI representations for every sensor or actuator functionality of the corresponding Phidget device (Figure 2g, 3rd row). Using an interface builder, programmers simply drag and drop these skins into a window, as done with our home appliance. This 'plug and play' approach requires minimal programming. In our experiences, skins provide an intuitive starting point for developers who have never dealt with hardware, where it entices them to experiment with the variety of physical interaction devices. It also serves as a very effective debugging mechanism.

We call them 'skins' as a single Phidget device can be represented by multiple skins (this differs from the strategy first described in [14]). For example, the Interface Kit can have a skin that displays all sensor values in a textual table, as animated sliders, or as a graph that shows changes over time. If they wish, programmers can create their own custom skins that visualizes a phidget in different ways, or that acts as a GUI to an abstract phidget that aggregates properties collected from several phidget devices (e.g., as described in the shared dictionary subsection). A programmer can even create multiple views onto phidget devices by connecting multiple skins to one or more phidgets.

Extensibility

Shared Phidgets is an extendable architecture. It is straightforward to include new Phidget Devices as they are made available by Phidgets, Inc. New interface skins can be created for existing phidgets, for abstract phidgets, or for new phidgets devices. The toolkit offers abstract base classes as the building block for these Phidget proxy classes and the interface skins; all implemented device objects and skins are derived from these. The base classes provide the main API and implementation of the Phidget discovery methods, shared dictionary connection, and subscription objects. Devices from other vendors can also be included, albeit with some more effort.

Discussion

The power of the Shared Phidgets architecture can be considered at three levels.

First, our distributed Model View Controller approach adds considerably to how programmers interact with distributed hardware devices. The hardware toolkits mentioned in the related work only access devices within a localized setting. Shared Phidgets remove this limitation, and allows programmers to easily and transparently access devices regardless of where they are located across the Internet.

Second, the ability to add metadata to devices means that programmers can now leverage semantic meaning associated with each device and location. Expected uses include using this metadata programmatically to decide upon location and context-dependant actions, and to discover device groupings and functions so they can be collectively considered as an appliance.

Third, and perhaps most importantly, we stress that the ability to create new abstract devices from a combination of hardware building blocks is extremely powerful. This is where Shared Phidgets reflect the offerings of the Context Toolkit. Letting people define abstract devices is the key to how we bridge between traditional physical user interfaces and Dey's context widget. As a dictionary entry, this new abstract device can encapsulate and abstract values from one or more actual devices. It can also interpret, transform and aggregate them into higher level abstractions. Because the programmer can also create new API's and Interface Skins that further wrap this abstract device, these devices can be presented as meaningful information appliances with APIs that reflect the semantics of that appliance.

HIGH LEVEL TOOLS

SharedPhidgets also comes with two important tools, each constructed atop the programming environment, that lets a developer or end-user monitor, control and even simulate all Phidgets distributed across the network (Figure 2h). Both tools are useful for administrating and/or debugging all connected devices regardless of what computer they are actually attached to. Due to space constraints, images of these tools are not shown, but are available on line (see §Software Availability).

The *Controller Tool* provides a direct view into the data model held by the shared dictionary, and is invaluable for debugging. By default, all Phidget devices and their data attributes (including metadata) are listed by their key paths and value pairs. However, the user can apply *filters* to display only key entries of one Phidget type (e.g., all Phidget Interface Kits), or of a specified serial number, or of specific data groups (e.g., metadata). As well, the user can select a field from a single Phidget and use that as a filter, where only that Phidget's attributes are shown. The user can also use this tool to create a simulated Phidget device, and populate its values. Under the covers, this creates the needed set of shared dictionary entries according to the device type. This simulation is extremely powerful, for it means that the end user can simulate a complete SharedPhidget network without actually attaching any hardware – this simulation can then be used to test other software. For example, a programmer could have simulated the InterfaceKit in the office part of our awareness appliance (Figure 1), and adjust its values to test the home part of it.

The *Device Explorer* provides a person with a compact and more readable overview of all devices across the network, including their serial number, attached status, metadata such as location, owner, etc. The user also has the option of interacting with any device by simply selecting it from the list; the interface skin for that device is created dynamically, through which the user can monitor and/or adjust its values. For example, a distant user can connect to the textLCD in our scenario example (revealing a skin similar to that shown in Figure 1), observe what it is displaying, and can change the text by typing into the skin.

Both programs are simply applications built atop the SharedPhidgets programming model.

EXAMPLES

Space does not allow us to describe the many example distributed physical user interfaces that can be constructed through the Shared Phidgets toolkit. However, a few are listed below just to give a feel for its power.

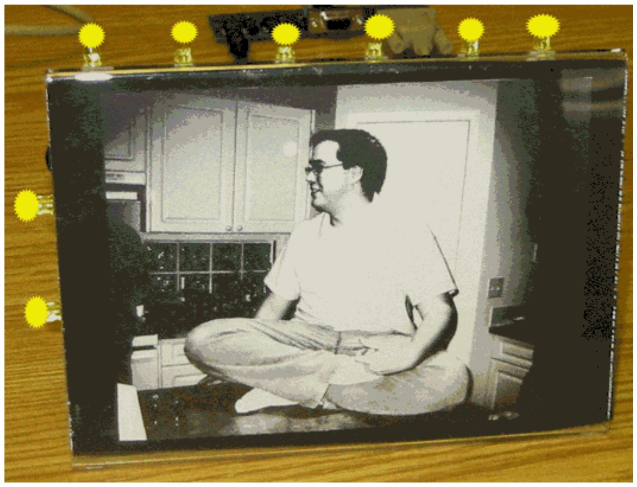


Figure 4. LumiTouch replication, by Kathryn Elliot

LumiTouch Replication. LumiTouch, built by MIT's Tangible Media Group, comprises a pair of interactive picture frames [6]. When a person touches one frame, the other frame lights up. Student Kathryn Elliot trivially rebuilt this (Figure 4) with two InterfaceKits, one per frame, each with attached LEDs and capacitive sensors. The lights on one frame are controlled by simply monitoring the capacitive sensor detecting touch on its distant partner frame.

Tablet Viewer. Want et al. [23] described a system where a person users a tablet computer to explore information attached to everyday objects. An RFID tag attached to the physical object associates that object with digital information; the tablet reads the tag and brings up that information. We replicated this by attaching an RFID tag to every Phidget device and/or sensor in a distributed space and taping an RFID tag reader to a Tablet PC. We then created an application where mobile users could scan and quickly associate an RFID tag with a corresponding phidget or sensor type. When the user rescans that tag, it would either bring up that device's skin (allowing immediate monitoring or control) or, for sensors, it would show the current sensor value and a graph of its previous values. This is an invaluable tool for debugging the state of devices within a physical environment.

However, its real value appears after a programmer creates abstract devices and skins that represent a set of phidgets as a true information appliance. As in Want et al., [23] the tablet becomes a 'window' into the appliance, revealing information and facilitating interaction with it in a way that was formerly possible only through extensive custom coding. For example, the LumiTouch replication in Figure 4 could be repackaged as an abstract device that permits a remote person to not only activate it by touch, but that lets them attach text messages to it. If the local person notices the lights flashing in a certain pattern, she can pass the tablet PC over it to display the message within that appliance's skin.

Location-based messaging. Previous studies show that home inhabitants exploit knowledge of each other's routines by leaving messages for them at particular locations

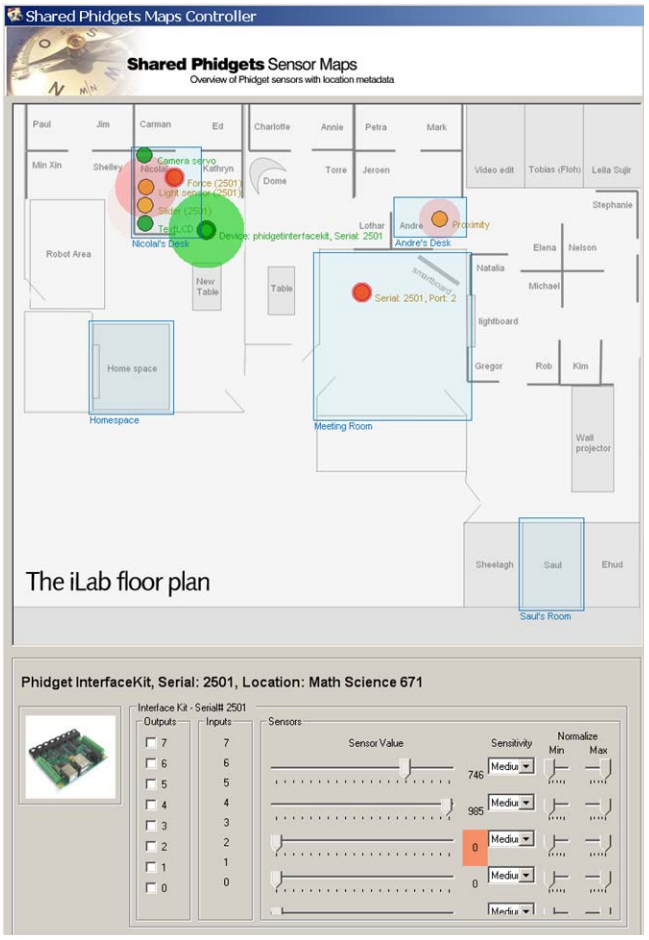


Figure 5. The Sensor Map Application.

[11]. Based on this, Elliot et. al. designed a system that allowed people to send notes to devices at particular locations within the home [11]. Her work used the standard Phidgets library, so much coding was needed to configure these distributed devices. In contrast, this could be easily rebuilt in Shared Phidgets. For example, each device (e.g., a Phidget LCDDisplay) is already tagged with a location metadata attribute, e.g., 'Kitchen', 'TV stand', 'Hallway', etc. If one wants to send a note to (say) the 'Hallway', one can quickly find the device with that metadata value and set its text display attribute.

Sensor Maps, illustrated in Figure 5, creates a map overview of all available Phidget hardware devices and sensors (any existing map image can be loaded into the application). Attached devices are displayed as small circles atop the map. When a device generates an event (e.g., sensor value change), the circle expands and fades (the radius depends on the current sensor value being tracked). Figure 5 illustrates one example used to show sensors distributed across a 3 room laboratory. As in previous examples, users can click on the device circle to display its interface skin, as seen in Figure 5, bottom. Regions on the map can be marked with additional location information, and this data can be added to all devices bounded by these regions. As before, its real use would be reflected by abstract devices. For example, the map could let a person see and control the state of all appliances located within a smart home.

CONCLUSIONS

Shared Phidgets is a new generation physical user interface toolkit that bridges the gap between current hardware-oriented toolkits and Dey's Context Toolkit. It recognizes that many physical user interfaces will comprise interacting distributed components, and that these components will be remixed and abstracted in a variety of ways.

In the past, programmers were responsible for all distributed systems aspects. Through its robust dMVC architecture, Shared Phidgets takes over this chore. Perhaps surprisingly, there is almost no extra programming penalty (unless one wants the extra power offered by accessing the shared dictionary directly); distributed phidgets can be coded in a manner very similar to the original non-distributed Phidgets system [14].

Another feature includes Interface Skins, which serves as both views and controllers into the distributed model; new skins can be created that are customized to the actual physical user interface. Through metadata, people can add appliance-specific, location and context-dependant attributes to the physical user interfaces they create.

Finally, the capabilities of phidgets can be interpreted, recombined, aggregated, and abstracted into abstract devices, which bridge into the powers offered by the Context Toolkit [8]. These abstract devices can be further abstracted as programmable objects and skins.

ACKNOWLEDGMENTS

Thanks to Phidgets, Inc., who provides a wonderful platform for us to extend. Research was partially funded by the NECTAR NSERC research network and its industrial sponsors: Smart Technologies Inc and Microsoft.

Software availability. Shared Phidgets software, documentation, tools, tutorials and examples are all freely available at http://grouplab.cpsc.ucalgary.ca/cookbook/.

REFERENCES

1. Abowd, G. D. Software engineering issues for ubiquitous computing. *Proc Int'l Conf. Software Engineering.* IEEE Press, 1999. 75-84.
2. Ark. W. and Selker, T. A look at human interaction with pervasive computers. *IBM Systems Journal* 38(4), 1999.
3. Ballagas, R., Ringel, M., Stone, M., and Borchers, J. iStuff: a physical user interface toolkit for ubiquitous computing environments. *Proc ACM CHI,* 2003.
4. Boyle, M. and Greenberg, S. Rapidly Prototyping Multimedia Groupware. *Proc DMS Distributed Multimedia Systems*, Knowledge Systems Institute, IL, USA, 2005.
5. Buxton, W. Living in augmented reality: Ubiquitous media and reactive environments. In K. Finn, A. Sellen & S. Wilber (Eds.). *Video Mediated Communication*, Erlbaum, 1997. 363-384.
6. Chang, A., Resner, B., Koerner, B., Wang, X, and Ishii, H. LumiTouch: An emotional communication device. *ACM CHI Extended Abstracts,* 2001. 313-314.
7. Dahley, A., Wisneski, C. and Ishii, H. Water Lamp and Pinwheels: Ambient projection of digital information into architectural space. *Summary ACM CHI*, 1998.
8. 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.
9. Dourish, P. *Where the action is: The foundation of embodied interaction*. MIT Press. 2001.
10. Dragicevic, P. and Fekete, J. The Input Configurator toolkit: towards high input adaptability in interactive applications. *Proc. AVI,* ACM Press, 2004, 244-247.
11. Elliot, K., Neustaedter, C. and Greenberg, S. Sticky Spots and Flower Pots: Two case studies in location-based home technology design. Technical report, Dept. Computer Science, University of Calgary. April 2006.
12. Greenberg, S. (In Press) Toolkits and Interface Creativity, *J. Multimedia Tools and Applications*, Kluwer.
13. Greenberg, S. Collaborative physical user interfaces. In K. Okada, T. Hoshi and T. Inoue (Eds) *Communication and Collaboration Support Systems*. IOS Press, 2005.
14. Greenberg, S. and Fitchett, C. Phidgets: Easy development of physical interfaces through physical widgets. *Proc ACM UIST*, 2001, 209-218.
15. Greenberg, S. and Kuzuoka, H. Using digital but physical surrogates to mediate awareness, communication and privacy in media Spaces. *Personal Technologies*, 4 (1), Elsevier, January 2001.
16. Greenberg, S. and Roseman, M. Groupware toolkits for synchronous work. In M. Beaudouin-Lafon (Ed), *Computer-Supported Cooperative Work (Trends in Software 7),* Chapter 6, Wiley & Sons, 1999.
17. Hartmann, B., Klemmer, S.R., and Bernstein, M. 2005. d.tools: Integrated prototyping for physical interaction design. *IEEE Pervasive Computing*, Oct-Dec 2005.
18. Ishii, H. and Ullmer, B. Tangible bits: Towards seamless interfaces between people, bits and atoms. *Proc. ACM CHI*, 1997. 234-241.
19. Lee, J., Avrahami, D., Hudson, S., Forlizzi, J., Dietz, P. and Leigh, D. The Calder toolkit: Wired and wireless components for rapidly prototyping interactive devices. *Proc ACM DIS*, 2004.
20. Making Things. www.makingthings.com. April. 2006.
21. Norman, D. *The Invisible Computer.* MIT Press, 1998.
22. Phidgets, Inc. www.phidgets.com. April. 2006.
23. Want, R., Fishkin, K., Gujar, A. and Harrison, B. Bridging physical and virtual worlds with electronic tags. *Proc. ACM CHI*, 1999.
24. Weiser, M. and Brown, J. Designing calm technology, *Powergrid Journal,* v1.01, July, 1996.