Volume
1

SAUL GREENBERG AND CHESTER FITCHETT

Phidgets in C#

A Tutorial

SAUL GREENBERG AND CHESTER FITCHETT

Phidgets in C# - A Tutorial

© Saul Greenberg and Chester Fitchett
Department of Computer Science • University of Calgary
Calgary, AB Canada T2N 1N4
Phone 403,220,6087 • Fax 403.284.4707
saul@cpsc.ucalgary.ca

Draft of February 28, 2005

Part 1

Introducing Phidgets

The first three chapters introduce Phidgets. Chapter 1 briefly explains what Phidgets are. Chapter 2 gives several 'getting started' examples that illustrate how easy it is to program Phidgets. Chapter 3 introduces concepts that are common to all phidgets. Chapter 4 shows you how to use Phidget Skins, which are a graphical interface to phidgets that you can use in a program with no coding.

Read these chapters closely and do the exercises. To avoid excessive repetition, the remaining parts of the book assume you have performed the exercises in Part 1 and understand them.

Chapter

1

What are Phidgets?

Phidgets are easy to program hardware devices that make it very easy for the average program to design and implement physical user interfaces .

Designing 'out of the box'

We are just now beginning to see a wave of new system designs that include 'out of the box' physical devices augmented by computing power. Instead of the conventional view of computers as boxes with large monitors, designers are now realizing that 'computers' can be just about anything: small displays, sensor networks, ambient displays, 'smart' furniture and appliances, and so on. Various research movements have embraced this approach: ubiquitous computing and calm technology, pervasive computing, tangible user interfaces, information appliances, and context-aware computing .

While exciting in concept and rife with opportunity, everyday programmers now face considerable hurdles if they wish to create even simple hardware-dependent applications. Even simple devices made out of cheap and readily available components (switches, sensors, motors) are hard to build unless one has a background in hobby electronics, circuit design or electrical engineering.

Perhaps the biggest—but most easily solved—obstacle is the sheer difficulty of developing and combining physical devices and interfacing them within the application software. This is where Phidgets come in.

The idea of Phidgets

Our original goal was to create hardware devices that were easy to program, test, debug, and extend. Most importantly, we wanted these devices to be so simple that developers could concentrate on the overall product design instead of low-level device construction and implementation. Our approach was to develop 'physical widgets', or *Phidgets*, that are almost direct analogs to how graphical user interface widgets are packaged and 'dropped into' applications. As we will see, all Phidgets comprise four parts:

- a hardware device (usually a board and perhaps some things you plug into it),
- a USB cable (to connect to the host computer),

- a programmer's API accessed by any .NET programming language and packaged as an object (a .NET component), and

- an (optional) graphical skin – also an object – that lets the end user see the device state and control it.

Phidgets are hardware that present themselves as a 'black box' to programmers via its API, which is itself encapsulated as an object. All phidgets have well-defined functions, accessed by the familiar programming notions of properties and events. A few of the many Phidgets are listed below. Other phidgets are described in Part II of this book, or on the Phidgets web site www.phidgets.com.

- **Phidget Servo** contains one or more servo motors, where they programmer can easily rotate a motor to an angle between 0 – 180 degrees by setting its position property.

- **Phidget RFID** is an RFID reader that returns the unique identity of an RFID tag as it is passed over the reader as an event.

- **Phidget InterfaceKit** is a general purpose device that returns (as events) changing values of various sensors and switches that are plugged into it, and that lets you turn low-power things (such as LEDs) on and off by setting a property.

- **Phidget WeightSensor** is a scale that returns the weight of the item placed atop of it as an event

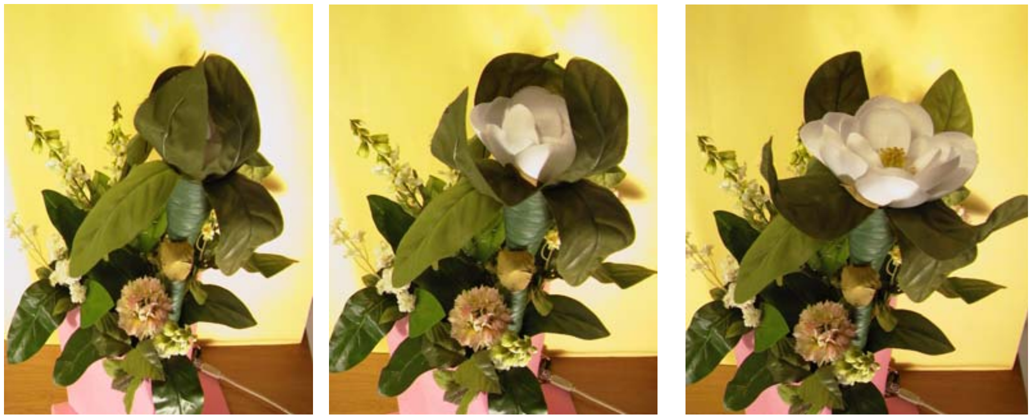
- **Phidget Encoder** is a rotary dial that returns as events the changes in position of the dial as a person rotates it, and whether or not its shaft is pressed as a button.

To use Phidgets, software people only have to leverage skills that they already know, i.e., how to program. They do *not* have to know anything about hardware, circuit design, firmware, wire protocols, low-level device managers, and so on.

Flower in Bloom

Several examples illustrate how phidgets can be used. Susannah McPhail, a student in a course using Phidgets at the University of Calgary, wanted to create a flower that bloomed under computer control. The final result is shown below. To build this flower, she used a servo motor, where the position of the motor could be rotated between 0 to 180 degrees. She attached a shaft to the head of the motor, so that motor rotation was translated into pushing and pulling a guide wire. This guide wire operated the leaves that wrapped itself around the flower.

The primary effort in this project was creating the linkage between the physical flower and the motor. The flower itself was modified from a store-bought plastic flower. Programming this operation was totally trivial. The initial version used the graphical skin (which required almost no coding), while a later version included a graphical control that was customized to opening and closing this flower (about 10 minutes of coding and screen design).


Flower in Bloom: closed partially bloomed fully bloomed

At this point, many things could be done with the flower. For example, using standard programming techniques and knowledge of Windows objects, one could (say) instrument the flower so that it reflects the on-line/away/off-line state of an Instant Messenger contact. Or the flower could be 'watered' every time someone visits your home page; with lots of visits, the flower is healthy and blooming. As visits wane, the bloom begins to close. Again, this can be done through basic programming, i.e., by monitoring hits to your web site, by incrementing the 'bloom' (i.e., motor position) every time a hit is observed, and by decaying the 'bloom' over time (e.g., through a timer control).

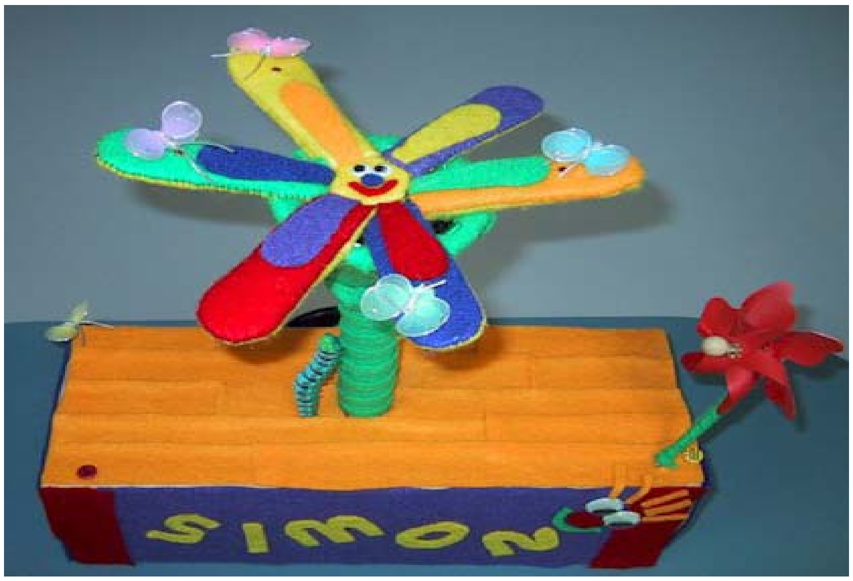
Picture Frames

Inspired by the MIT Tangible Media Group's Lumitouch frame, Kathryn Elliot implemented the Picture Frames system shown below. Picture Frames was designed as a messaging system to show that one person is thinking about another, and that this thought is reciprocated. These are two or more picture frames, each attached to a different person's computer, that interact with one another as an ambient display. Co-workers, friends or loved ones separated by distance use the Picture Frame as a way to make others aware of their presence in a non-intrusive manner and/or to communicate emotional content. Exploiting the tangible property of the device, touching one frame causes lights to blink on its partner's frame, and the remote person can respond in turn. Any 5"x7" picture may be inserted into the Picture Frame, which transforms the device into a fairly concrete embodiment of the remote person.

Picture Frames.

Simon the Sunflower

Simon the Sun Flower, developed by Nancy Lopez, recreates Hasbro's Simon game for children as a flower with glow-worms located on its petals (see below). The flower creates a visual and audio pattern by playing a sequence of sounds while lighting up glow worms on particular petals (via LEDs). The child whose turn it is repeats the pattern by squeezing the bugs on the correct leaves (each contains a pressure sensor); if they make a mistake, the sound changes and it is the next child's turn. As with the previous example, it was built using LEDs connected to a Phidget InterfaceKit, as well as pressure sensors. The programming challenge was not on controlling these LEDs or acquiring sensor data, but on developing and playing pleasing sounds. The construction challenge was sewing and gluing the various parts of the flower together.



Simon the Sunflower

Other examples.

A myriad of example projects developed using Phidgets are viewable as both pictures and videos at http://grouplab.cpsc.ucalgary.ca/phidgets/ and selecting the Gallery link. Almost all items shown were developed in a very short amount of time (a few weeks) by students as their very first Phidget project.

Are you ready for Phidgets?

Average programmers can learn to use and program Phidgets with ease. I know this from experience. I have taught many students – all 'average' programmers - how to program phidgets. They needed very little technical instruction to get going. Indeed, their only difficulty was relearning the skills acquired in Grades 1 to 7: cutting paper, using glue, building things with clay, sewing, building with Lego, and so on.

This book is here to help you through the easiest part of that process: learning how to program Phidgets. Don't be intimidated by the fact that there are many pages in this book.. The book is this thick only because there are quite a few different types of Phidgets, and each phidgets has a corresponding chapter. In practice, you will likely find yourself able to program Phidgets after reading the three chapters in part 1, and the chapter(s) dedicated to the particular Phidget you want to program.

Chapter

2

Getting Your Hands Dirty

This chapter provides several introductory examples of how you can program and run several phidgets.

This chapter offers three simple examples that illustrate step by step how you can program and run phidgets. No extra details are provided; the examples are just intended to whet your appetite. You can choose to program one or all of these examples, although your choice will depend somewhat on the actual phidgets you have on hand. Full details for each phidget are described in later chapters. If your programs run as described, you will also know that everything is installed correctly on your system.

Before trying these examples, make sure that you have installed the phidget software, and that your phidgets software components are visible in your C# Visual Studio .NET environment. Appendix 1 describes how to do this.

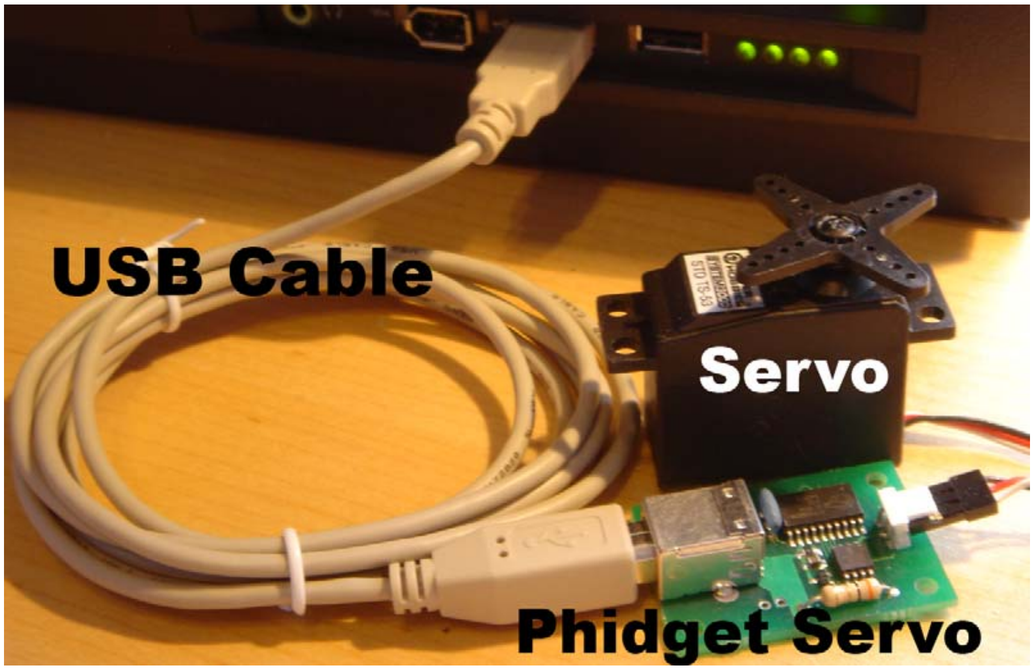
Making a Servo Motor move

You will create a program that rotates a servo motor to a random position between 0 and 180 degrees whenever the Phidget Servo is plugged into the computer.

What you need

Our first example uses a 1-motor Phidget Servo kit comprising a servo motor, the Phidget Servo hardware board, and a USB cable. As shown in the figure, plug the connector leading from the servo motor to the connector on the Phidget Servo hardware board; the white, red and black wires on the connector should match the 'w r b' letters on the board. Then use a USB cable to plug the hardware board into a USB port on your computer. You may get a message on your computer saying "New Hardware found", but no further action needs to be taken by you (if it says to restart the computer, just ignore that message).

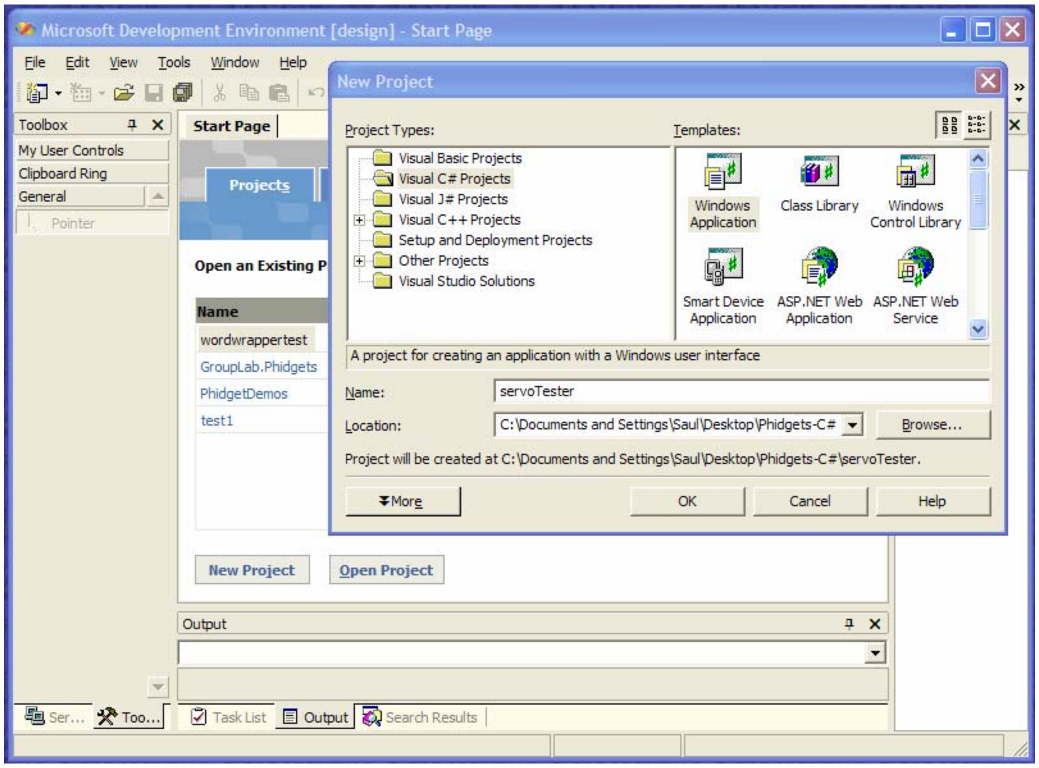
This exercise will also work with a 4-motor Phidget Servo. Just make sure a motor is plugged into the first connector on the board. Since that connector is powered by the USB cable, you do not need to use the power adaptor.



A 1-servo phidget plugged into a computer; the servo motor is also connected to the hardware board.

What you have to do

Start Visual Studio .NET and create a new C# Windows Application project called 'servoTester'.


Microsoft Development Environment [design] - Start Page

New Project

Project Types:
- Visual Basic Projects
- Visual C# Projects
- Visual J# Projects
- Visual C++ Projects
- Setup and Deployment Projects
- Other Projects
- Visual Studio Solutions

Templates: Windows Application, Class Library, Windows Control Library, Smart Device Application, ASP.NET Web Application, ASP.NET Web Service

A project for creating an application with a Windows user interface

Name: servoTester

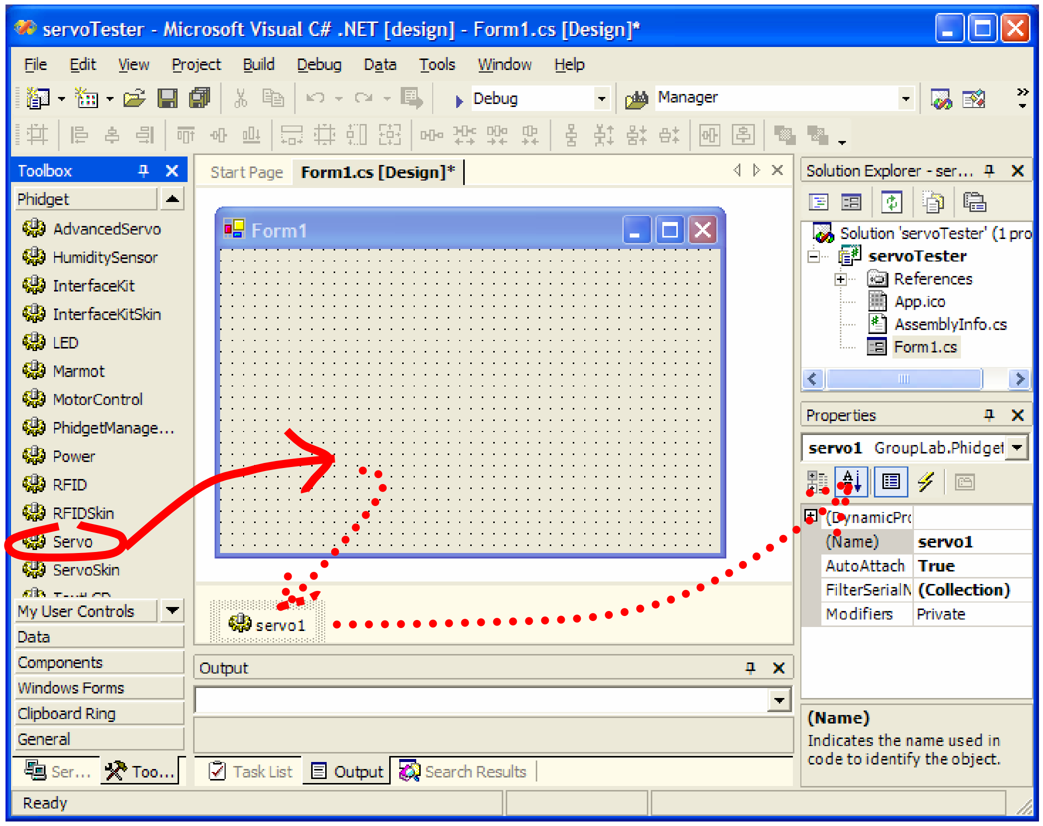
Location: C:\Documents and Settings\Saul\Desktop\Phidgets-C#

Project will be created at C:\Documents and Settings\Saul\Desktop\Phidgets-C#\servoTester.

Creating a new C# project titled 'servoTester'.

Open the Phidget Tab in the Toolbox and find the component labeled 'Servo' in the list. This is illustrated in the following figure.

Drag and drop the Servo component from the Toolbox onto the Form1 window. After you do so, you will see a **servo1** icon appear below the form. This too is illustrated in the following figure. If you click on **servo1** icon, it will raise the properties of the servo1 object in the properties window.

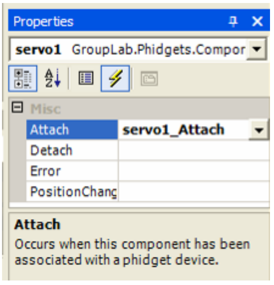


(DynamicPro	
(Name)	servo1
AutoAttach	True
FilterSerialN	(Collection)
Modifiers	Private

(Name)
Indicates the name used in code to identify the object.

Drag and drop a Servo component into the form, and view its properties.

Create an event handler for servo1's Attach event. This is done in the usual .NET way. That is, click on the servo1 component to see its properties in the properties window, then click on the event button (the lightning bolt) to see the list of events, then click in the empty field next to the Attach event and hit return. A method called `servo1_Attach` will be automatically created



Attach	servo1_Attach
Detach	
Error	
PositionChang	

Attach
Occurs when this component has been associated with a phidget device.

Insert the following code in the `servo1_Attach` event handler. The `Attach` event is raised when the servo device is seen by the system (i.e., when you start the program if the servo is plugged in, or if you plug in the servo after the program has been started. Through the code below, the `servo1_Attach` event handler will just rotate the 1st servo motor (counting from 0) to a random position between 0 and 180 degrees by setting that motor's `Position` property.

```
private void servo1_Attach(object sender, System.EventArgs e)
{
   Random random = new Random ();
   servo1.Motors[0].Position = random.Next (180);
}
```

Execute the program. The servo motor should rotate to a random position when the program starts (assuming the servo is attached to the computer by the USB cable). With the program running, try unplugging the USB cable and then plugging it back in again; you should see the motor rotate again to a new random position.

The complete program. Even though you have only written two lines, Visual Studio .NET filled in – as it usually does – additional code. The program is provided below for completeness, but there should be no reason for you to type this in (unless you are not using Visual Studio). Most of the code creates basic classes and constructs the window interface necessary to make this program executable; only a small amount of this code creates and configures the servo component and its event handler.

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace servoTester
{
   public class Form1 : System.Windows.Forms.Form
   {
      private GroupLab.Phidgets.Components.Servo servo1;
      private System.ComponentModel.Container components = null;

      public Form1()
      {
         InitializeComponent();
      }

      protected override void Dispose( bool disposing )
      {
         if( disposing )
            if (components != null) components.Dispose();
         base.Dispose( disposing );
      }

      private void InitializeComponent()
      {
         this.servo1 = new GroupLab.Phidgets.Components.Servo();
         this.servo1.AutoAttach = true;
         this.servo1.PhidgetDevice = null;
         this.servo1.Attach +=new System.EventHandler(this.servo1_Attach);
         this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
         this.ClientSize = new System.Drawing.Size(292, 266);
         this.Name = "Form1";
         this.Text = "Form1";
      }

      [STAThread]
```

```
        static void Main()
        {
           Application.Run(new Form1());
        }

        // Whenever the servo device is attached (plugged in), rotate
        // the motor to a random position between 0 and 180 degrees
        private void servo1_Attach(object sender, System.EventArgs e)
        {
           Random random = new Random ();
           servo1.Motors[0].Position = random.Next (180);
        }
   }
}
```

For the adventurous. If you wish, try the following on your own. What the steps do is let you operate a servo motor through a trackbar (a slider), where feedback is given on the attach state of the Phidget Servo as well as the motor position. Don't worry if you have problems or don't fully understand everything; a later chapter will detail a robust version of this exercise.

- Add a trackbar to the form. Through the properties window, set its `Enabled` property to false and its `Maximum` property to 180.
- In code, enable the trackbar in the `servo1_Attach` event handler. Assign the motor's `Position` property to the trackbar's `value` property.
- Add a label to the form, and set its text to the trackbar's position. This will let you see the current angle of the motor.
- To make it more robust, create an event handler for the servo's `Detach` event and disable the trackbar within it. As a consequence, a person can change a motor's position only when the Phidget Servo is plugged in.
- Using the `Title` property of the form, provide textual feedback (added in the Attach and Detach events) that describes if the Phidget Servo is attached or detached.

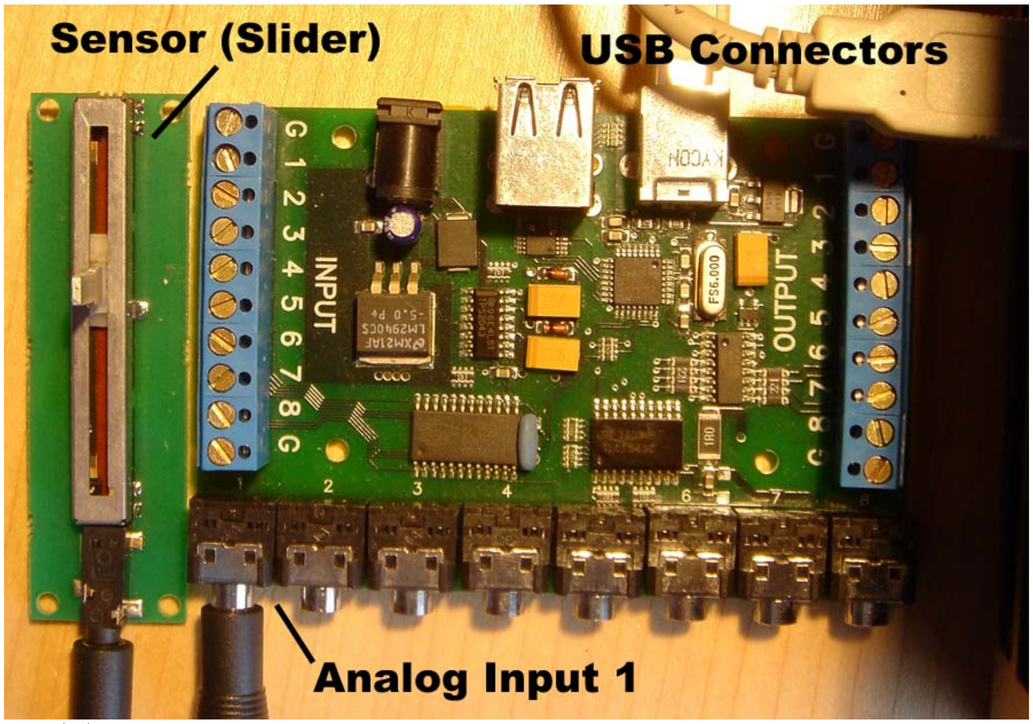
Getting Analog Input from an InterfaceKit

You will create a program that displays input gathered from an analog input (a sensor) plugged into an InterfaceKit.

What you need

You should have a version of an InterfaceKit that includes the ability to connect sensors to it. The most common version is an 8/8/8 InterfaceKit, which means that you can connect up to 8 Analog Inputs (i.e., sensors) to it, up to 8 Digital Inputs (i.e., switches), and up to 8 Digital Outputs (i.e., low powered devices such as LEDs). In the following discussion, 'sensors' and 'analog inputs' are the same thing.

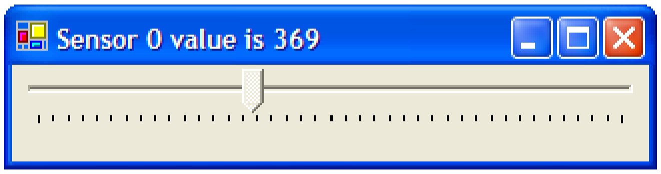
As illustrated in the figure below, plug a sensor into Analog Input port #1. You can use any of the many different sensors provided with the InterfaceKit – force sensors, light sensor, rotation sensor, slider, joy stick, motion sensor – but probably the slider is the easiest to try if you have one. Using the USB cable, plug the Interface Kit hardware board into your computer. As with the servo, you may get a message on your computer saying "New Hardware found", but no further action needs to be taken by you (if it says to restart the computer, just ignore that message).



An 8/8/8 Phidget InterfaceKit, with a slider sensor plugged into Analog Input #1.

What you have to do

You will create a program that resembles the one below. A trackbar will graphically display the current value of the first analog input, while the title of the window will show the sensor and its value in textual form.



To do this, you will follow a sequence of steps that is very similar to those you already did in the earlier Servo example.

Start Visual Studio .NET and create a new C# Windows Application project called 'interfacekitTester'.

Drag and drop a TrackBar component from the Toolbox onto the Form1 window and select it. After you do so, you will see the **trackBar1** properties appear in the properties window. Set the following properties of the trackBar1 component:

Minimum	0
Maximum	1000
Tick Frequency	25
Enabled	False
Dock	Fill

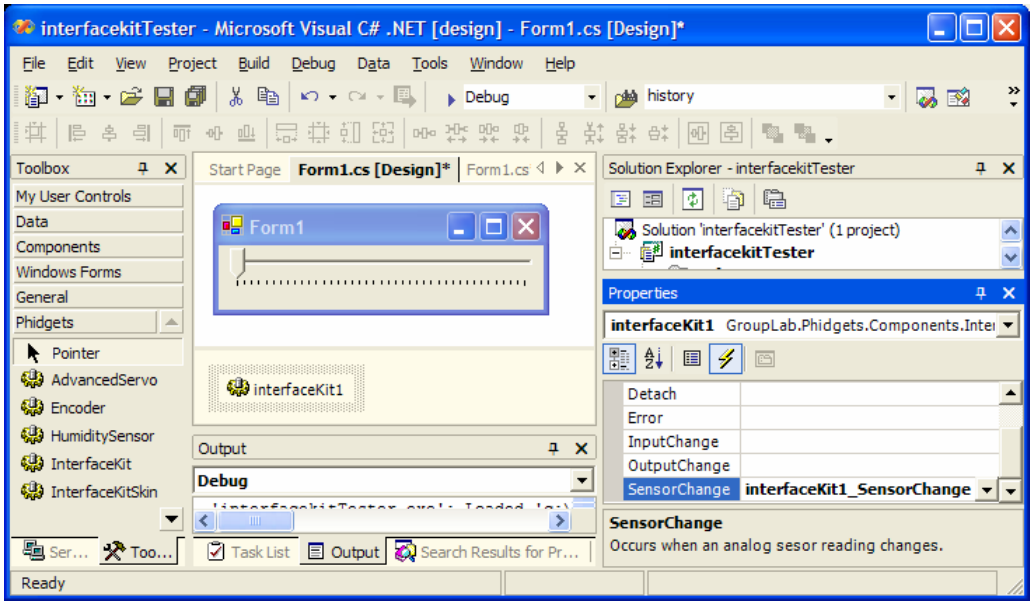
The trackbar will be used to display the sensor values; Enabled is set to False because there is no need for the trackbar to be interactive. The sensor values returned are always between 0 and 1000, which is why the Minimum and Maximum properties of the trackbar are set to those values. Finally Dock is set to Fill so the trackbar always resizes itself to fill the window.

Open the Phidget Tab in the Toolbox and find the component labeled 'InterfaceKit in the list.

Drag and drop the InterfaceKit component from the Toolbox onto the Form1 window. After you do so, you will see an **interfacekit1** icon appear below the form. If you click on **interfacekit1** icon, it will raise the properties of the interfacekit1 object in the properties window.

Create an event handler for interfaceKit1's SensorChange event. This is done in the usual .NET way. That is, click on the **interfacekit1** icon to see its properties in the properties window, then click on the event button (the lightning bolt) to see the list of events, then click in the empty field next to the `SensorChange` event and hit return. A method called `interfaceKit1_SensorChange` will be automatically created.

At this point, your project should resemble the figure below.



Detach	
Error	
InputChange	
OutputChange	
SensorChange	interfaceKit1_SensorChange

SensorChange
Occurs when an analog sesor reading changes.

Insert the following code in the interfaceKit1's SensorChange event handler. The `SensorChange` event is raised whenever any Analog Input generates a new value, which is returned in the events argument `e`. This events argument contains two new properties above and beyond what is normally found : `e.Index` is the number of the Analog Input that generated the event, while `e.Value` is the current value of that Analog Input. The value is always within 0 to 1000.

```
private void interfaceKit1_SensorChange(object sender,
GroupLab.Phidgets.Components.EventArguments.IndexSensorChangeEventArgs e)
{
   if (e.Index != 0) return;
   this.Text = "Sensor " + e.Index + " value is " + e.Value;
   trackBar1.Value = e.Value;
}
```

In this case, we are only interested in the first Analog Input (numbering starts at 0), so we check if that is the one that generated the event. If it isn't, we exit the event handler. Otherwise, we change the title bar to provide feedback as to which sensor generated the event and its current value, and we set the trackbar's value to be the same as the sensor's value.

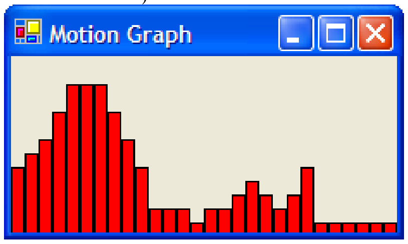
Execute the program. When you manipulate the sensor, the trackbar and window should reflect the current value of the sensor. Try unplugging that sensor and plugging another one in; you will see that the program continues to display the values of that new sensor.

For the adventurous. If you wish, try these 3 different exercises on your own. Don't worry if you have problems or don't fully understand everything; a later chapter will fully detail the Phidget InterfaceKit.

1. Add other trackbars to the form, each corresponding to a different Analog Input. By using a switch statement on the e.Index value, you should be able to track the values of many sensors at the same time.

2. Create a graphical object (e.g., a rectangle). Scale the object so that its size is adjusted by the value of the sensor.

3. If you have a motion sensor, create a bar graph that displays the motion detected over time, such as the one illustrated here.

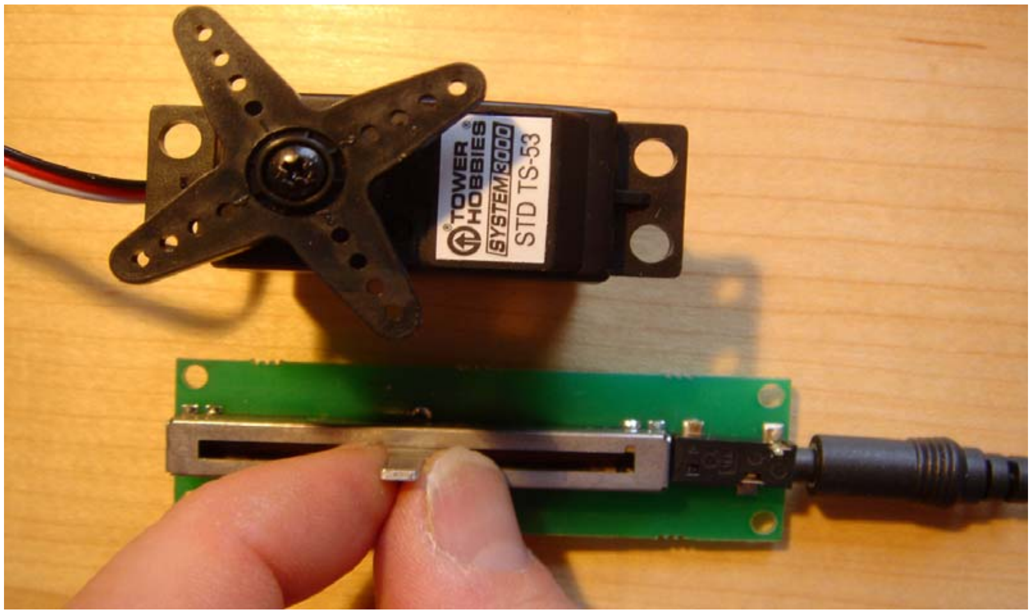


Using an Analog Input to Operate a Servo

You will create a program that uses the input gathered from an analog input sensor plugged into an InterfaceKit to rotate a servo.

What you need

This final example will use both a servo and an InterfaceKit. You will use a sensor attached to the InterfaceKit (e.g., a slider) to control the servo motor, as illustrated in the figure below. To build this example, you will need everything you used in the two earlier examples.



As before, plug a sensor such as a slider into Analog Input #1 of the Phidget InterfaceKit. Using the USB cable, plug the Phidget InterfaceKit into a USB port on your computer. Using another USB cable, plug the Phidget Servo into another USB port on your computer.

Note: if you do not have two USB ports available on your computer, you will have to purchase a USB hub. Alternatively, you can use the USB hub that is built into certain versions of the 8/8/8 InterfaceKit (note the power supply is required for the USB hub to work).

What you have to do

Either add the following steps to the program you created previously for the InterfaceKit example, or repeat all the previous steps to create a new program. Test it to make sure it works, i.e., that the slider repositions that trackbar as before.

Now, **drag and drop a servo component onto the form** (as in the Servo example). However, don't bother setting up the `Attach` event.

Modify the interfaceKit1_SensorChange method as follows.

```
private void interfaceKit1_SensorChange(object sender,
GroupLab.Phidgets.Components.EventArguments.IndexSensorChangeEventArgs e)
{
   interfaceKit1.Sensors[0].Sensitivity = 1;
   if (e.Index != 0) return;
   this.Text = "Sensor " + e.Index + " value is " + e.Value;
   trackBar1.Value = e.Value;
   // If there is a servo attached, make its position match the value
   // returned by the sensor, translated to a number between 0 and 180
   if (servo1.Attached)
   {
      float position = (e.Value / 1000f) * 180;
      servo1.Motors[0].Position = position;
      this.Text += ". Servo at: " + position;
   }
}
```

The first few lines are identical to what was seen in the previous example, although we added a line that makes the sensor highly sensitive by changing that sensor's Sensitivity value. By default, sensors will only raise an event if their value has changed by at least 10 units from the previously generated value. By setting the sensitivity to 1, the event will be raised any time there is a difference in the current value from the previously generated one.

We then check to see if the `servo1` component is in an `Attached` state. If it is, we calculate the servo motor's position so that the sensor value (which is between 0 to 1000) is mapped onto the allowable motor positions (i.e., between 0 to 180 degrees). We then set the servo motor `Position` property to that position, and append a bit of text to the Window's title so that it also shows what position the motor is in.

Execute the program. The servo motor position should now follow any changes to the sensor. It is also robust; if you unplug the servo, the program continues to run and the sensor value is still reported.

Summary

The purpose of this chapter was to give you a taste of programming phidgets. As you should now know, it is a fairly simple matter to control phidgets (such as with the Phidget Servo) or to collect input from them (such as with the Phidget InterfaceKit). It is also fairly simple to have one type of phidget hardware control another type of phidgets hardware, all mediated by software. Or you can have software controlling the hardware , or hardware controlling the software!

The following chapters add detail to what you have learnt here. They describe features common to all phidgets, and then walk through the full capabilities of each phidgets. They also describe a special component called the Phidget Manager.

While many details are provided in the following chapters, what is important is that – as with these examples – most phidgets can be operated at their basic level in just a few lines of uncomplicated code.

Chapter

3

Concepts Common to All Phidgets

All Phidgets have a variety of concepts in common. They all behave the same way when the hardware is attached (plugged into the computer) or detached. They all share several programmable properties that describe the particular phidget. They (almost) all come with graphical 'skins' that lets you control the phidgets and/or see its state.

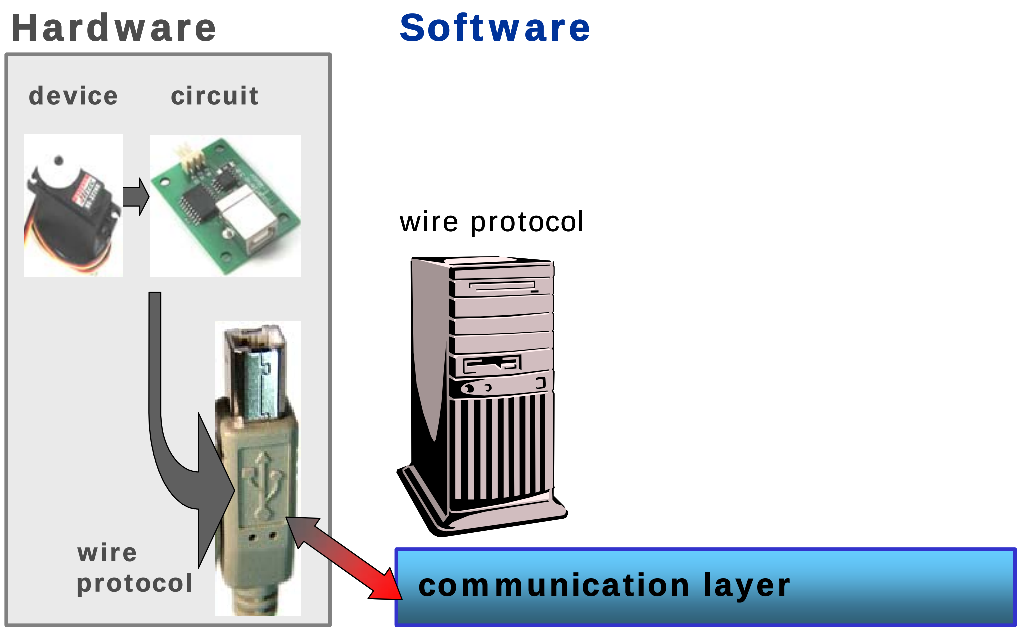
All phidgets share several things. Once you learn what these are, you can apply them to your treatment of any phidget. This chapter describes these common concepts. We will use the Phidget Servo as our running example, but all phidgets are treated the same way.

The Phidget Philosophy: Hardware as Software

In C#, phidgets present themselves as software components (i.e., objects). All phidget programming is done through these components using standard notions such as properties, methods, events and event handlers. No knowledge of hardware, firmware, or wire protocols is needed.

Hardware complexity

Until phidgets were available, most programmers had to build hardware from scratch. This included finding appropriate devices to control, assembling a circuit board to control that device (based around a micro-controller chip), writing firmware for that micro-controller, figuring out how to connect that circuit board to a computer via some wire (e.g., serial port, USB, etc.), defining and writing the wire protocol between the controlling computer and the software, wrapping this up in some kind of programmable interface, debugging and testing the hardware and software, and so on. This is hard work, even for something as simple as a servo motor (see figure below). To make matters worse, the average programmer does not even have the necessary electrical engineering skills to do some of these tasks. No wonder so few programmers dealt with hardware!



A programmer's nightmare: low level construction of hardware and wire protocol.

Hardware as a software component

The Phidget philosophy is to make hardware accesible to the average programmer, even those with zero hardware knowledge. It does this by presenting the Phidget hardware as a 'black box'. All the low level implementation details of the hardware, firmware, and network protocol are hidden from the end programmer. Instead, the programmer only sees a software component – an object in C# – with a well-defined application programmer's interface (API). Technically speaking, phidgets inherit from `System.ComponentModel.Component`. Thus, excepting a few notable exceptions discussed later in this chapter, the programmer uses this component in a way that is almost identical to other C# .NET components (e.g., timers).

Properties, methods, events and event handlers

Like other components, Phidget components have properties, methods and events. Event handlers (or callbacks) can be attached to an event in the standard C# way, so that the event handler is invoked automatically whenever the appropriate event is raised by the component. As well, the programmer can use the Visual Studio .NET interface builder to create instances of the component simply by dragging and dropping components from the Toolbox onto the form. Some (but not all) of a component's properties and events of the component are displayed in Visual Studio .Net's Properties window. Using the form-filling properties window at design time, the programmer can then redefine property values or specify the component's event handlers.

Graphical Skins

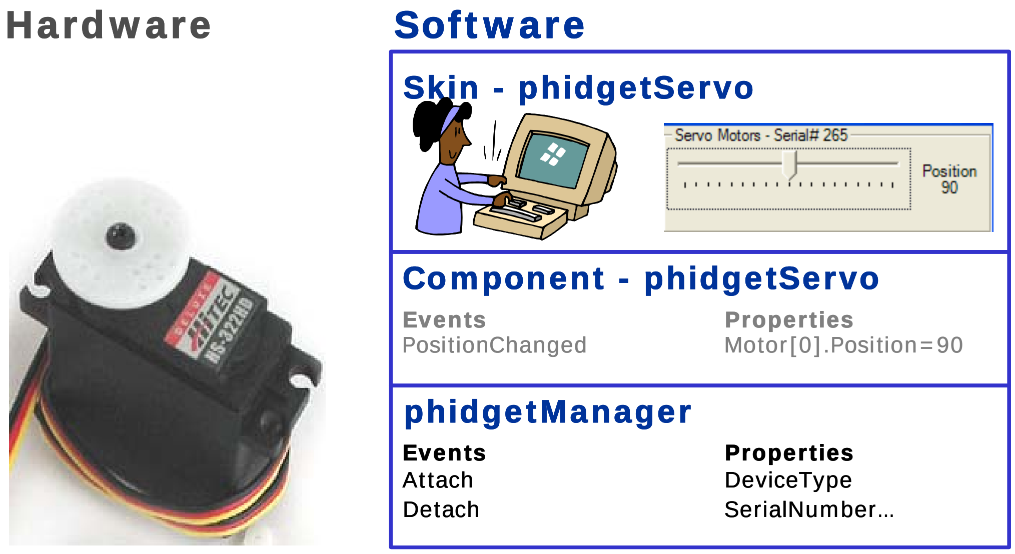
In practice, programmers often build a graphical interface that visually shows the end user the current state of a particular Phidget, and/or that lets the end user control the hardware via graphical controls. While this is straight-forward to build from scratch, it is still a burden. Thus each Phidget component is accompanied by a 'skin': a graphical component that visually represents a Phidget component. As with user controls, skins can be dragged from the Toolbox and dropped into a form. After the skin is connected to the Phidget component, it displays the Phidget hardware's state and shows various controls. Chapter 4 shows how to use existing skins, while details of each Phidget's skin will be discussed in chapters describing that Phidget. A later chapter illustrates how custom skins are constructed.

Attaching and detaching hardware

There is one very important difference between Phidgets and normal components. Unlike software which is always present, the actual Phidget hardware may or may not be plugged into the computer. When the hardware is not plugged in, the component is in a 'detached' state: while the software component exists, it has no hardware to control. When the hardware is plugged in, the component is in an 'attached' state: the software component is actually connected to the hardware and (under the covers) it is communicating with it.

To simplify the job of detecting when hardware is attached or detached, Phidgets include a special kind of software called the Phidget Manager. As we will see shortly, the Phidget Manager can be exploited so that it automatically connects your component to phidget devices, or it allows you to do it by hand.

The Phidget Manager, the specific Phidget components, and the Phidget skins all work together to provide the end programmer with a powerful means to control hardware devices without worrying about the details of hardware programming. As the figure below illustrates, all the hardware and networking complexity is hidden; the only thing the programmer has to do is use familiar programming concepts to control the device.



A programmer's delight: programming hardware through familiar software concepts.

Language independence

While this book refers only to C#, Phidgets are language independent. Under the covers, they are implemented as COM objects. Thus they are accessible to all the old Visual Studio 6 languages (C++, J++, Visual Basic, etc). While COM objects are also accessible in .NET, the Phidgets.NET version wraps the COM object so that all

Phidgets appear as true .NET objects. This makes them easier to program and accessible from any .NET language e.g., C# and VB7.

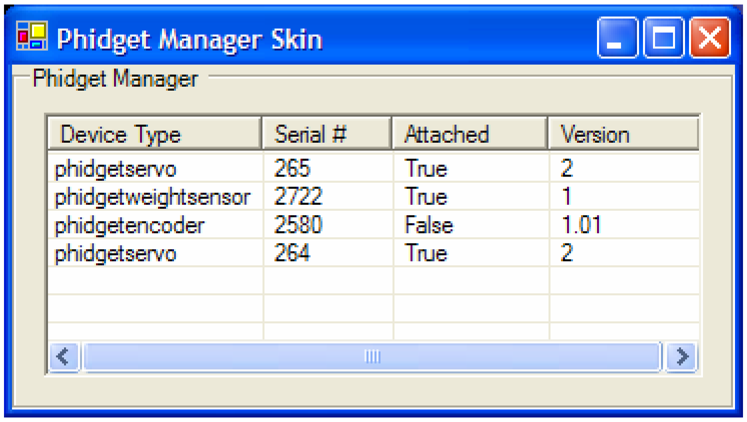
The Phidget Manager Concept

The Phidget Manager automatically monitors phidgets as they are attached or detached from the computer.

The Phidget Manager usually works behind the scenes. It monitor all phidgets as they are plugged or unplugged (attached or detached) from the computer. Perhaps the best way to illustrate what the Phidget Manager does is by example.

Example

In this example, you will create the program illustrated below. You will use the Phidget Manager Skin to show a list of all phidgets that have been plugged into the computer since the program was started. What you will see on your list will likely differ from this one, as you may plug in different devices. However, in this case we see 4 phidgets listed: two Phidget Servos, a Phidget Weight Sensor, and a Phidget Encoder. All are identified by their 'Device type' name, a unique serial number and the version number of the hardware. We also see that only three of the devices are currently attached to the computer; the encoder is detached.



Device Type	Serial #	Attached	Version
phidgetservo	265	True	2
phidgetweightsensor	2722	True	1
phidgetencoder	2580	False	1.01
phidgetservo	264	True	2

What you have to do

This program is trivial to build: really! All you have to do is drag and drop a Phidget Manager Skin onto the form.

Drag and drop a Phidget Manager Skin onto the form. You do this following the steps quite similar to those in the previous chapter. Find the PhidgetManagerSkin component on the Toolbox under the Phidgets tab. Drag and position this component onto the window. Resize it as needed: make sure all four columns are visible. You do not have to change any properties or write any code.

Test it. Run the program. Plug in several phidgets. You will see new ones appear on the list, with their Attached state shown as true. Try unplugging phidgets. When a particular phidget is unplugged, its attached state is shown as false. Plug it back in again and you will see the attached state shown as true.

That's it! You can now use this program as a diagnostic tool, i.e., to see what phidgets are recognized and even to look up their basic properties.

It should now be clear that the Phidgets platform allows you to dynamically plug and unplug devices at run time, and that the Phidget Manager detects this. While you can work directly with the Phidget Manager to detect these devices (as discussed in a future chapter), most of the time you will let the Phidget Manager work automatically behind the scenes. Instead, you will use each phidgets component's Attach and Detach events, discussed next.

Attach and Detach Events

The Attach and Detach events are raised whenever a phidget is plugged in (attached) or unplugged (detached) from the computer.

All Phidget components have two special events, called `Attach` and `Detach`. These events are triggered behind the scenes by the Phidget Manager. All also have a special property called `Attach` which reflects whether a particular component is attached to a Phidget hardware device.

event **Attach** is invoked whenever a Phidget hardware device attaches to its matching component, but only if the component is not already connected to another Phidget hardware device.

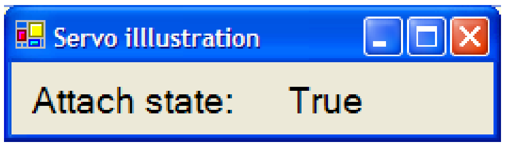
event **Detach** is invoked whenever a Phidget hardware device is disconnected from its matching component. This usually happens when you unplug the Phidget hardware from the USB port.

bool **Attached** [get] returns true if the Phidget hardware is plugged in and is attached to its matching component, or false otherwise.

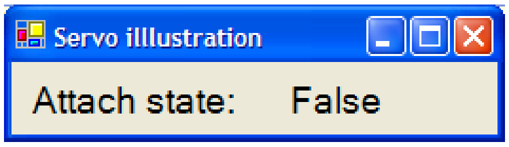
As we will see in a later section, you can restrict attachment to devices that have particular serial numbers.

Example

To illustrate how attach and detach works, we will build an example that shows how a servo component recognizes these events. When you run this program and repeatedly plug and unplug a servo device into the computer, the program will alternate between these two states:



Attach state: True



Attach state: False

If you don't have a Phidget Servo, just use whatever Phidget device you have; just select the corresponding component for that device. For example, if you have a Phidget RFID device, use the RFID component.

What you have to do

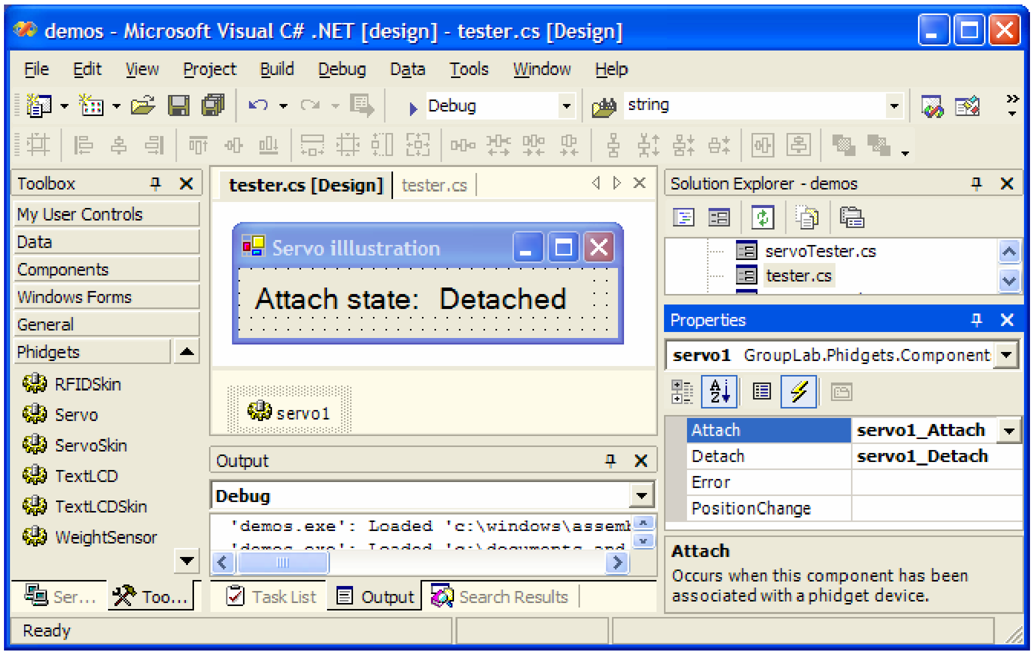
Start a new program.

Add two labels to the form to the left and then to the right of each other. They will automatically be called `label1` and `label2`. If you want, increase their font size to (say) 12 point. Rename `label2` (via its `Name` property) to `labelAttachStatus`. Set their properties via the Properties window or via code as:

```
label1.Text                = "Attach state:" ;
labelAttachStatus.Text     = "False";
```

Drag and drop a servo component onto the form.

Create event handlers for servo1's Attach and Detach events. These are done in the normal Visual Studio .NET way, i.e., click on the servo1 component to show it in the Properties window (don't forget to click on the lightning bolt icon to show the events). Then click and hit return in the `Attach` field to automatically create the `servo1_Attach` event handler. Do the same to the `Detach` field to create the `servo1_Detach` event handler. At this point, your project should look something like this. We are now ready to start coding.



```
'demos.exe': Loaded 'c:\windows\asseml
'demos.exe': Loaded 'c:\documents and
```

servo1 GroupLab.Phidgets.Component

Attach	servo1_Attach
Detach	servo1_Detach
Error	
PositionChange	

Attach
Occurs when this component has been associated with a phidget device.

Program the Attach event handler via the `servo1_Attach` event handler. This is executed when the servo device is seen by the system (i.e., when you start the program if the servo is plugged in, or if you plug in the servo after the program has been started.

```
private void servo1_Attach(object sender, System.EventArgs e)
{
   //A servo device just attached. Provide feedback.
   labelAttachStatus.text = servo1.Attached.ToString ();
}
```

Program the Detach event handler via the servo1_Detach event handler.

```
private void servo1_Detach(object sender, System.EventArgs e)
{
   //A servo device just deatached. Provide feedback.
   labelAttachStatus.text = servo1.Attached.ToString ();
}
```

Test it. Start the program. Plug in a servo and unplug it. You will see the display provide the appropriate feedback.

Don't throw away this program! You will modify it for our next example.

Identifying the Phidget Hardware

Through software, you can discover various properties about the Phidget hardware such as its name, its unique serial number, and the hardware's version number.

Whenever a Phidget hardware device is attached, you can discover through software various details about it. This is held in a phidget component's `PhidgetDevice` property.

PhidgetDevice is a handle that lets you access properties that describe particular device details. While some of these properties are specific to the particular type of hardware, all PhidgetDevice will have at least the following:

string **Name** [get] returns the name of the device as a string.

For example, a Phidget Servo device will always return the name "phidgetservo" if you call (say) the `servo1.PhidgetDevice.Name`.

int **SerialNumber** [get] returns a unique serial number for this device as an integer. This serial number is actually burnt into the hardware. This is an important field, for it means you can differentiate between particular Phidget Servos if you have several plugged in.

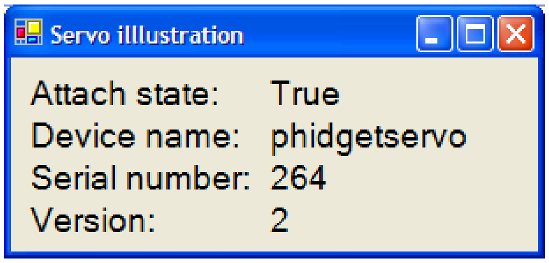
For example, if you plugged in a Phidget Servo whose hardware had the serial number 325, a call to `servo1.PhidgetDevice.SerialNumber` would return 325.

float **Version** [get] returns the version number of the hardware. This too is burnt into the hardware. While this can be useful if you have to check the version due to a change in capabilities, it is rarely used.

For example, if you plugged in version 2.1 of a Phidget Servo, a call to `servo1.PhidgetDevice.Version` would return 2.1.

Example

To illustrate how these properties work, we will extend our previous example to show how a servo component can list their values. When you run this program and plug in a servo device into the computer, the program will display something like:



```
Attach state:   True
Device name:    phidgetservo
Serial number:  264
Version:        2
```

Again, if you don't have a Phidget Servo, just use whatever Phidget device you have. This will work as long as you select its corresponding component.

What you have to do

Open up the previous example as we will just add to it.

Add 3 rows of two labels each to the form below the original ones, so that it resembles the figure above. Change the names of the labels on the right to labelName, labelSerialNumber, and labelVersion, and clear their text properties. As before, increase all font sizes to (say) 12 point. Set the text properties of the left labels to reflect the example above, either through the properties window or through code. For example, (note that the label names may differ from what is below):

```
label2.Text = “Device name:” ;
label3.Text = “Serial number:”
label4.Text = “Version:”
```

Modify the Attach and Detach event handler so that it includes the code below.

```
private void servo1_Attach(object sender, System.EventArgs e)
{
   labelAttachStatus.Text = servo1.Attached.ToString ();
   labelName.Text = servo1.PhidgetDevice.Name;
   labelSerialNumber.Text = servo1.PhidgetDevice.SerialNumber.ToString ();
   labelVersion.Text = servo1.PhidgetDevice.Version.ToString ();
}

private void servo1_Detach(object sender, System.EventArgs e)
{
   labelAttachStatus.Text = servo1.Attached.ToString ();
   labelName.Text = "";
   labelSerialNumber.Text = "";
   labelVersion.Text = "";
}
```

It should be clear that all we are doing is accessing and printing the properties of the phidgets device. When the deivice is detached, we are just clearing the text fields.

Test it. Start the program. Plug in a servo. You will see the display provide the appropriate feedback. If you unplug it and plug in a different servo, the serial number will change. If you have different versions of a servo device, the version numbers will change as well.

Again, don't throw this program away. There are a few more things you will do with it before we are done.

AutoAttach and Serial Number Filtering

Using the AutoAttach event, a Phidget component will automatically attach itself to an appropriate hardware device. By filling in the FilterSerialNumber collection with serial numbers, auto-attachment will ignore devices whose serial numbers are not on the list.

Each phidget component has two other special properties: AutoAttach and FilterSerialNumbers. These are defined below.

bool **AutoAttach** [get, set] If the phidgets component is not already attached to a device, it will automatically try to attach to any matching phidget device that becomes available. To further restrict what the component should attach to, it looks at the **`FilterSerialNumber`** collection. If it contains any serial numbers, the component will only attach to appropriate devices that contain a serial number included in that collection. Default is true.

IntegerCollection **FilterSerialNumbers** [get, set] is a collection of serial numbers. It is used by `AutoAttach` to filter the devices it is allowed to attach to.

Whenever you create a new phidget component (e.g., by dragging and dropping a component onto a form), the `AutoAttach` property is (by default) set to True and the `FilterSerialNumber` collection is empty. Because `AutoAttach` is true, the component will work behind the scenes with the Phidget Manager to look for a matching device. If `AutoAttach` were set to false, you would have to use the Phidget Manager directly to look for devices (as explained in a future chapter).

Because the `FilterSerialNumber` collection is empty, any matching device will be connected to the component (no matter what its serial number) and an `Attach` event will be raised.

The programmer can use the properties window at design time to add serial numbers to the `FilterSerialNumber` collection. Alternately, these numbers can be added at run time programmatically using the standard .NET `Collection` methods. Because `AutoAttach` is True and this collection is not empty, the component will check each matching device to see if its number is in the collection. If and only if it is, that device will be connected to the component and an `Attach` event will be raised.

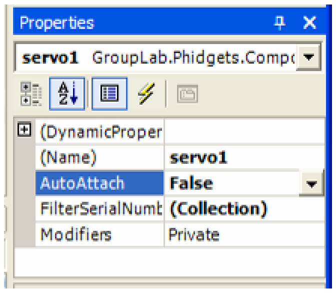
Example

To illustrate how these two properties work together, we will again extend our previous example. In this case, no changes to the code are required.

What you have to do

Run the previous example and write down the serial number of the attached servo shown on the display. In the version above, this serial number was 264.

Change the `AutoAttach` property to False. Go to the servo's property window, and choose False from the pull down menu, as illustrated on the right.

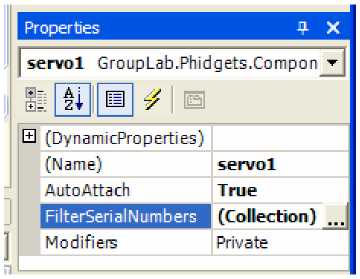

Properties

servo1 GroupLab.Phidgets.Compo

(DynamicProper	
(Name)	servo1
AutoAttach	False
FilterSerialNumt	(Collection)
Modifiers	Private

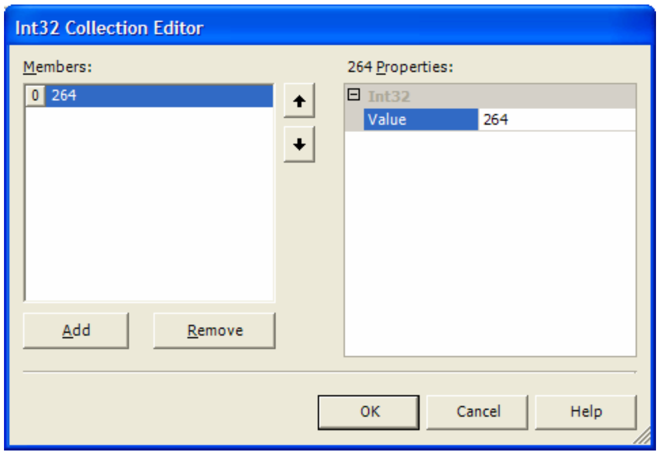
Run the example. The component does not attach to the Phidget Servo even though it is plugged in. The Attach event is not raised. Then set the `AutoAttach` property back to True

Add 264 to the `FilterSerialNumber` collection. You can do this by clicking the '…' ellipses in the `FilterSerialNumber` Property, and adding the value 264 to the list via the dialog box. This is illustrated below. Don't forget to set the `AutoAttach` property back to True!


Properties

servo1 GroupLab.Phidgets.Compon

(DynamicProperties)	
(Name)	servo1
AutoAttach	True
FilterSerialNumbers	(Collection)
Modifiers	Private


Int32 Collection Editor

Members:

0 264

264 Properties:

Int32

Value	264

Run the example. The component attaches to the Phidget Servo, as you would expect. If you have a second Phidget Servo, try unplugging the first one and plugging in the second one. You will see that the other servo does not attach, as its serial number is not on the list. You can also plug both servos in, and restart the program. Again, you will see that only the phidget servo with the right serial number is included.

Why the FilterSerialNumber property is important

Filtering by serial numbers is important whenever you need to use multiple instances of a device, e.g., several Phidget Servos. This occurs when the product you build contains several of the same kind of device and you have several phidgets components to represent them. By tying each component to a particular instance of the Phidget hardware (by its serial number), each component will control the right device.

Filtering by serial number is also important when you plan to plug in several products, as each product may use the same kind of device.

Chapter

4

Phidget Skins

Almost all Phidgets come with graphical 'skins' that lets you try it out, where you can interactively control the Phidget and/or see its state.

Almost all Phidget components are accompanied by a graphical skin. While Phidget components are objects with a programming API accessed through methods, properties and events, skins are graphical interfaces that visually show the end user the current state of a particular Phidget, and/or that lets the end user interactively control the hardware via graphical controls. Skins are very easy to use, where they require no programming to access their basic operation. Instead, a programmer just drags and drops both a component and its accompanying skin from the Toolbox onto a form, and sets a single property in the skin to associate it to the component. The program is then run, and the skin appears.

This chapter shows how an existing skin is used. A few examples will be shown, but full details on particular Phidget skins will be discussed in each chapter describing that Phidget.. A later chapter illustrates how custom skins are constructed.

Programming the Servo Skin

You will create a program that displays a servo skin, which in turn will let you adjust the servo motor positions by a slider.

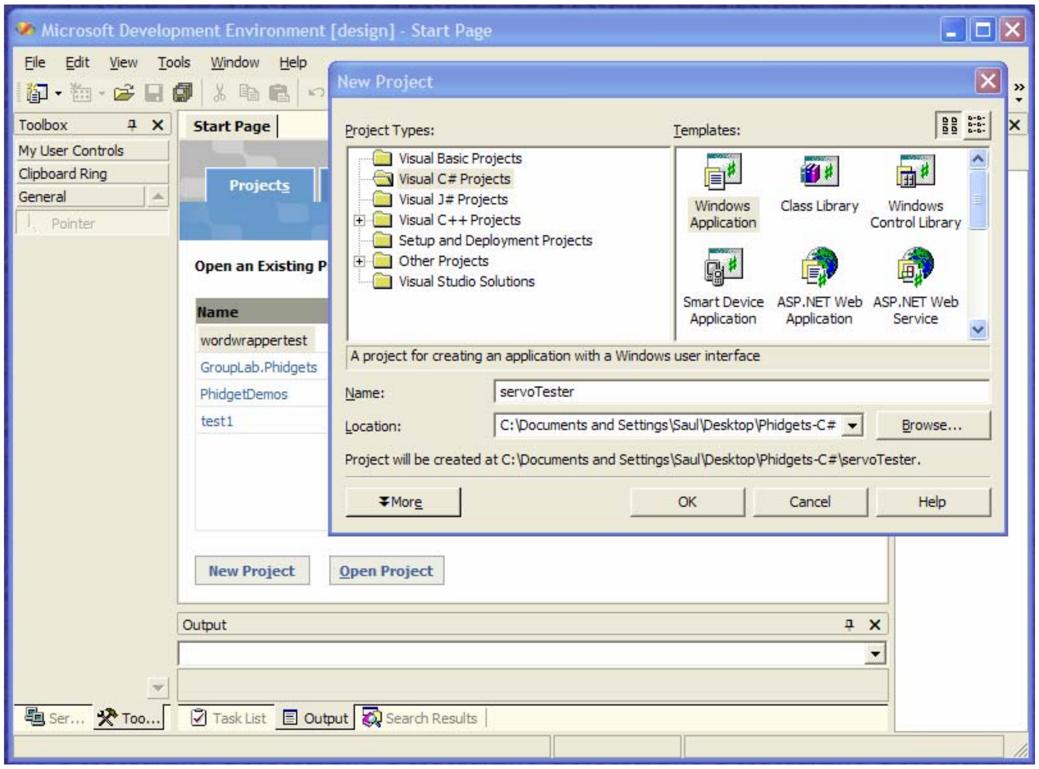
What you need

We assume that you have assembled a 1 or 4 motor servo as described in Chapter 2, that you have Visual Studio .NET and are reasonably familiar with using it, that all the phidget software has also been installed on your machine, and that you have already tried to run the servo example from that chapter.

If you don't have a Phidget Servo, you can use any other phidget. However, you will have to choose the component and the skin representing the Phidget you choose (e.g., an RFID and RFIDSkin for a Phidget RFID, an InterfaceKit and InterfaceKitSkin for a Phidget InterfaceKit, etc). If you follow the basic idea of this example, your Phidget should work

What you have to do

Start Visual Studio .NET and create a new C# Windows Application project. Give it a descriptive name, e.g., servoTester, servoSkinExample, etc.



A project for creating an application with a Windows user interface

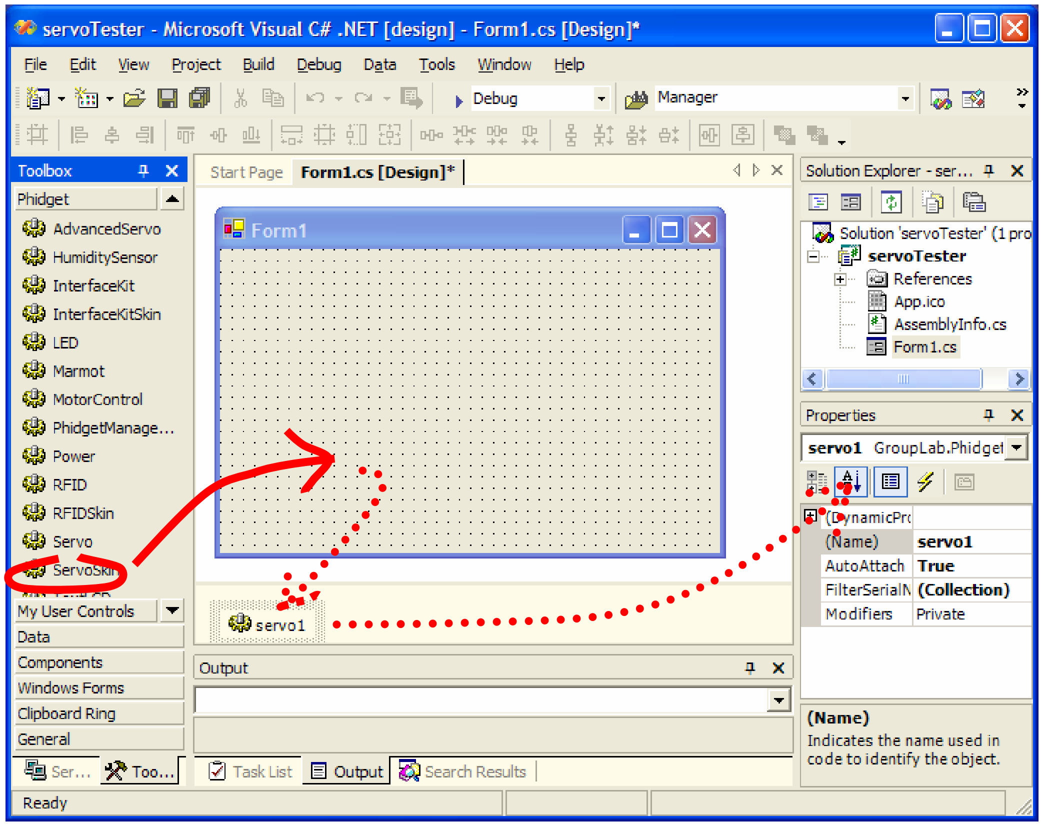
Name: servoTester

Location: C:\Documents and Settings\Saul\Desktop\Phidgets-C#

Project will be created at C:\Documents and Settings\Saul\Desktop\Phidgets-C#\servoTester.

Open the Phidget Tab in the Toolbox**.** You will see quite a few Phidgets and Phidget Skins listed (see figure below). Through this Phidget tab, you will be able to use the interface builder supplied by .NET to rapidly construct the basics of your Phidget programs.

Drag and drop the Servo component from the Toolbox onto the Form1 window. After you do so, you will see a **servo1** icon appear below the form. If you click on this, it will raise the properties of the servo1 object in the properties window. The servo is an object with properties and events; it does not have a graphical representation. This sequence is annotated in red in the figure below.

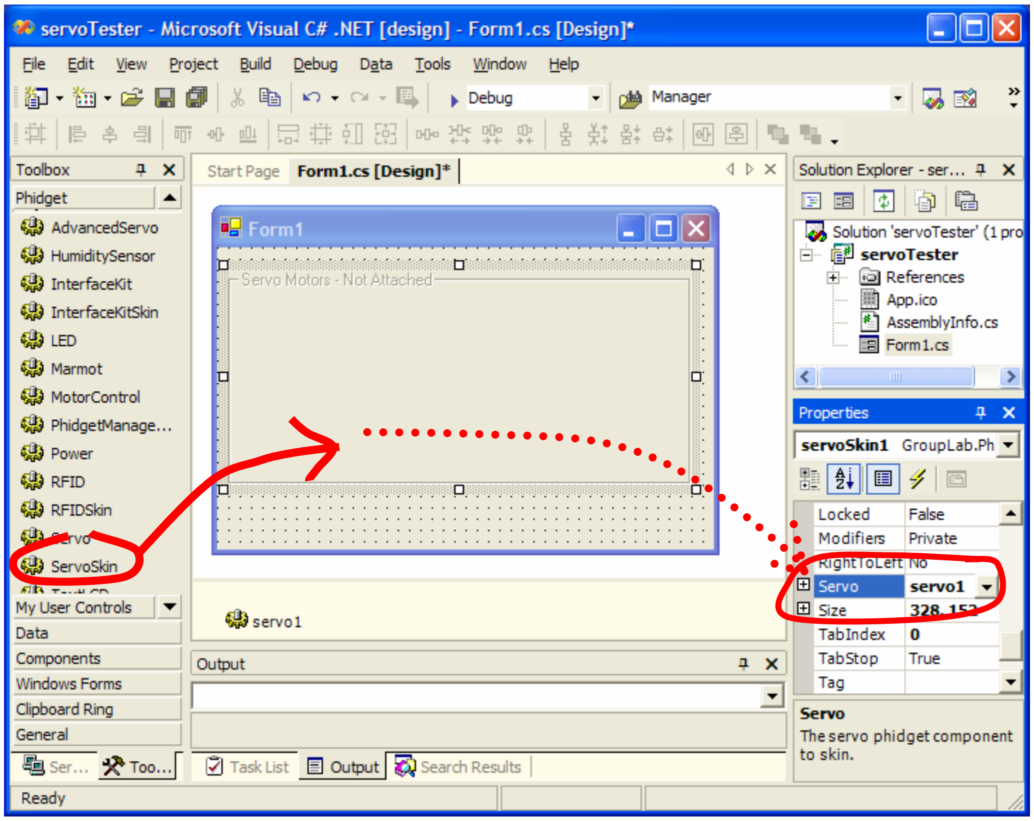


(DynamicPro	
(Name)	servo1
AutoAttach	True
FilterSerialN	(Collection)
Modifiers	Private

(Name)
Indicates the name used in code to identify the object.

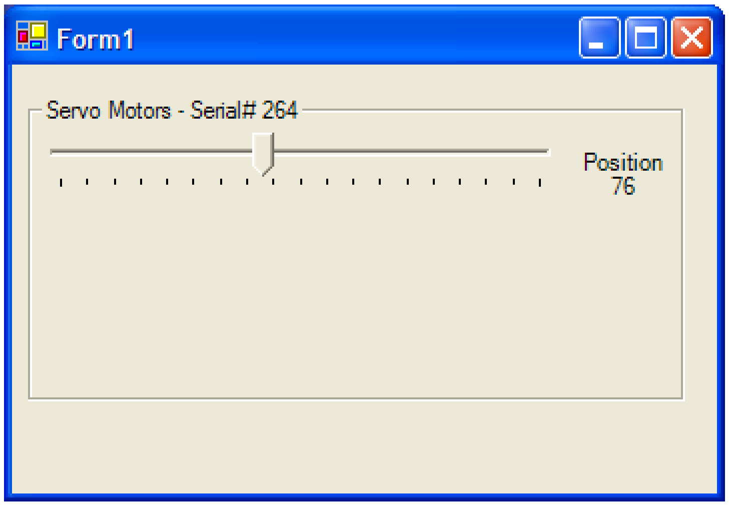
Drag and position a ServoSkin from the Toolbox onto the Form1 window, as illustrated in the next figure. This creates the graphical skin – a widget – that will let you see and control the servo motor. It will likely be called servoSkin1. It too is an object, but with graphical properties. Click on it to see its properties in the Properties window.

Set the Servo property of the servoSkin1 to 'servo1'. This connects the graphical skin to the actual servo component. This step is annotated in red in the figure below.



Run the program

Build and execute the program in the usual way. The window below shows you what should appear if you have properly connected a 1-motor servo. A 4-motor servo would look almost identical, except you would see four trackbars, one for each motor.



Move the slider on the trackBar. As you move it, the servo motor plugged into the Phidget Servo should rotate to the given angle. This angle can be between 0 to 180 degrees. The actual angle will appear on the right of the trackbar.

For 4 motor servos, try the other trackbars. Each trackbar, from top to bottom, corresponds to the motors numbered 0 to 3.

Unplug the USB cable. You should see the window go gray (inactive), and the label will say that the motor is not attached.

Plug the USB cable back in. The window will reactivate, and you can continue to rotate the motor.

Try the skins for your other phidgets

If you have other phidgets, try them out. Almost all phidgets have accompanying skins, and they are inserted in a program in almost the same way as shown in the servo example.

Skins are also a great way to explore the capabilities of your Phidget: most have controls and/or displays that reflect all the things you can do with your particular Phidget.

Appendices

Appendix

1

Installing Phidgets

The phidget software is fully integrated into .NET via a two-stage process. The first stage installs the software on you r computer, while the second stage installs the various phidgets components into Visual Studio .Net's Toolbox

Phidgets comprise both software and hardware. If you want to use Phidgets, you must make sure the proper version of the .NET-enabled Phidget software is installed on your machine.

Preconditions

Before you install the Phidget software, you need the following.

- A computer with at least one available USB port
- The Windows 2000 or XP operating system (or later)
- Microsoft Visual Studio .NET

You should also be familiar with C# and the Microsoft Visual Studio .NET programming environment. While this book provides basic details to guide the Visual Studio and C# neophyte, it is not a tutorial on how to use Visual Studio nor how to program in C#.

If you have installed previous versions of Phidgets

If you have previous versions of Phidget software on your machine, you should uninstall them. You can do this by raising the Add or Remove Programs dialog box (accessed through the Control Panel on the Start Menu), and removing any software that has "Phidgets" as part of its title.

If you had previously installed any Phidget components in the Microsoft Visual Studio .NET Toolbox, you should also remove those. For example, if they are all collected in a tab, you can right-clicking on the tab to raise the context menu, and then select the menu option that deletes the tab. If you don't see any phidgets listed in your Microsoft Visual Studio Toolbox, then you probably have no need to worry about this.

Software Installation

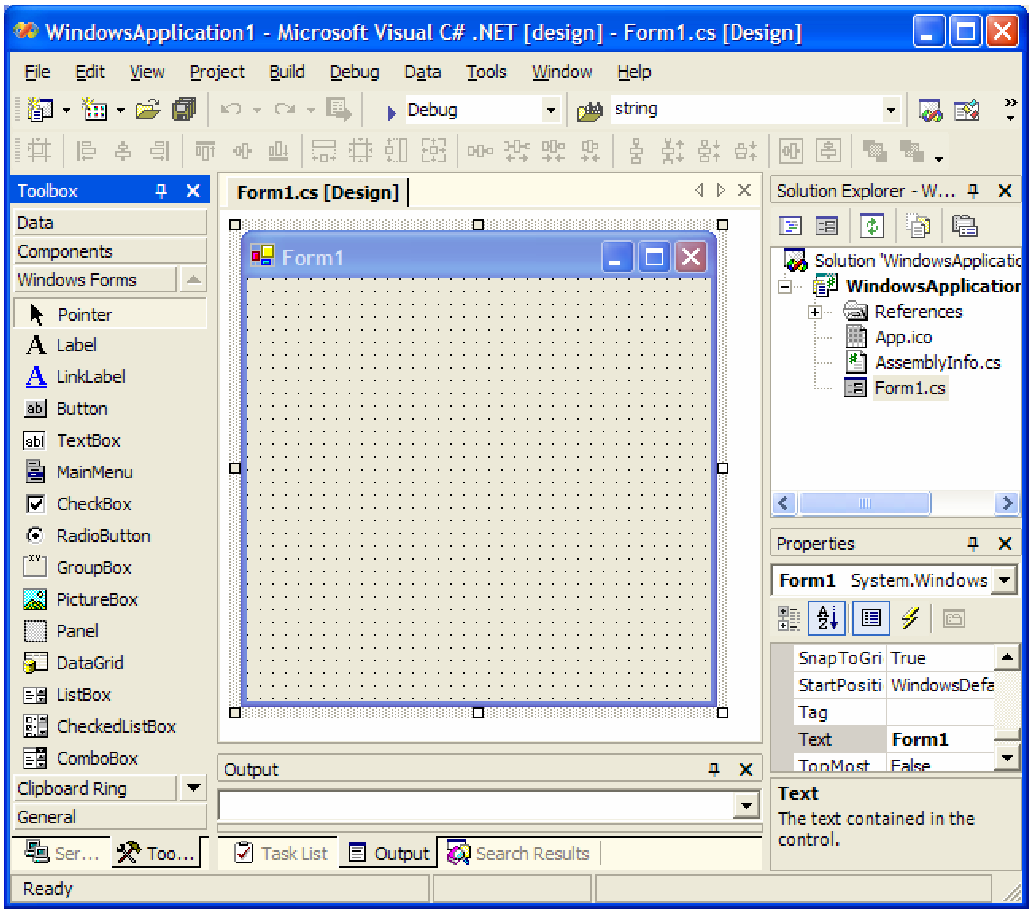
Installing phidgets software is similar to how most software is installed on a Windows machine. Go to http://grouplab.cpsc.ucalgary.ca/phidgets. Follow the directions to download the Phidgets.Net version. Click the link, which will execute the

Grouplab.Phidgets.msi (the installer package) and accept all the default recommendations of the dialog boxes.

Adding Phidgets to the Toolbox

While your phidget software is now installed, there is one further step required if you want it to appear in a convenient form within your Visual Studio .NET environment. As you will shortly see, phidgets are designed as software components that can be dragged and dropped from the Windows Toolbox onto the Form. For this to work, you have to manually add the phidgets components into the Toolbox through the following sequence of steps.

Open Visual Studio .Net and create a new project. You will probably see something similar to this window.



SnapToGri True
StartPositi WindowsDefa
Tag
Text Form1
TopMost False

Text
The text contained in the control.

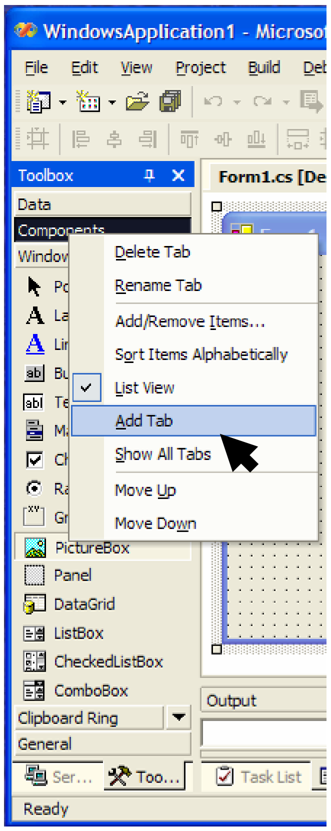
An empty C# project. The Visual Studio .NET Toolbox in on the left.

Find the Toolbox. .NET's Toolbox is a panel usually positioned on the left side of the above screen. You are likely familiar with it already, as it is the normal place you go to when you wish to drag and drop user controls (such as a Button, found in the Windows Forms tab) and components (such as a Timer, found in the Components tab) onto the form. If it is not visible, select the View→Toolbox menu item.

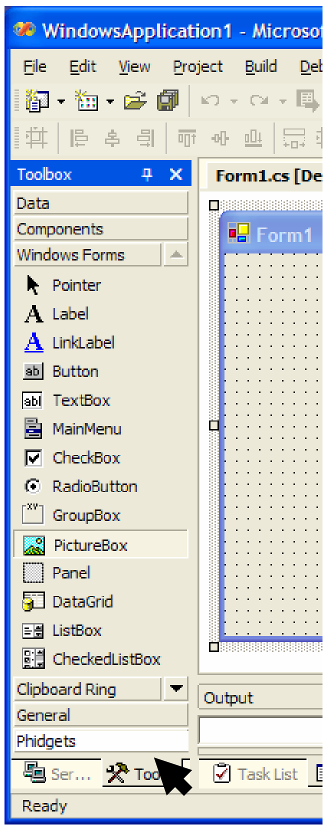
Add a new tab in the Toolbox. Right click on the title of an existing tab (e.g., the Components tab) to raise a context menu. Select the 'Add Tab' menu item (left side of figure, below), which will create a new empty tab at the bottom of the Toolbox (center of figure, below). The text cursor will be placed in there: type in the label "Phidgets" to name this tab.

Open the Phidgets Tab by clicking on it (right side of figure, below). It should be empty except for a cursor.

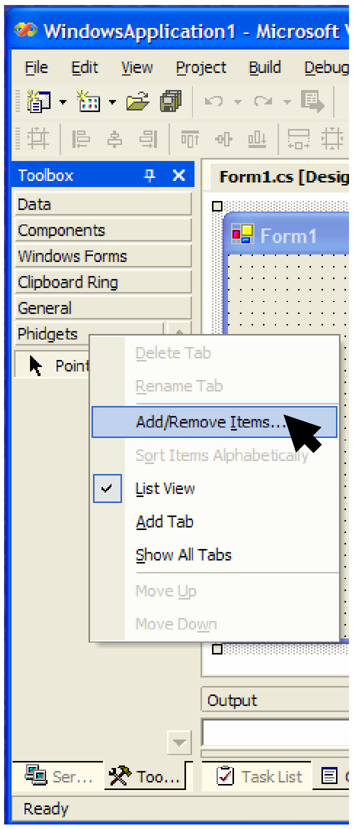
Select the Add/Remove Items menu item by right-clicking within the tab and raising the context menu (right side of figure, below). The Customize Toolbox dialog box will appear, and will resemble the window below.



First Add the Tab

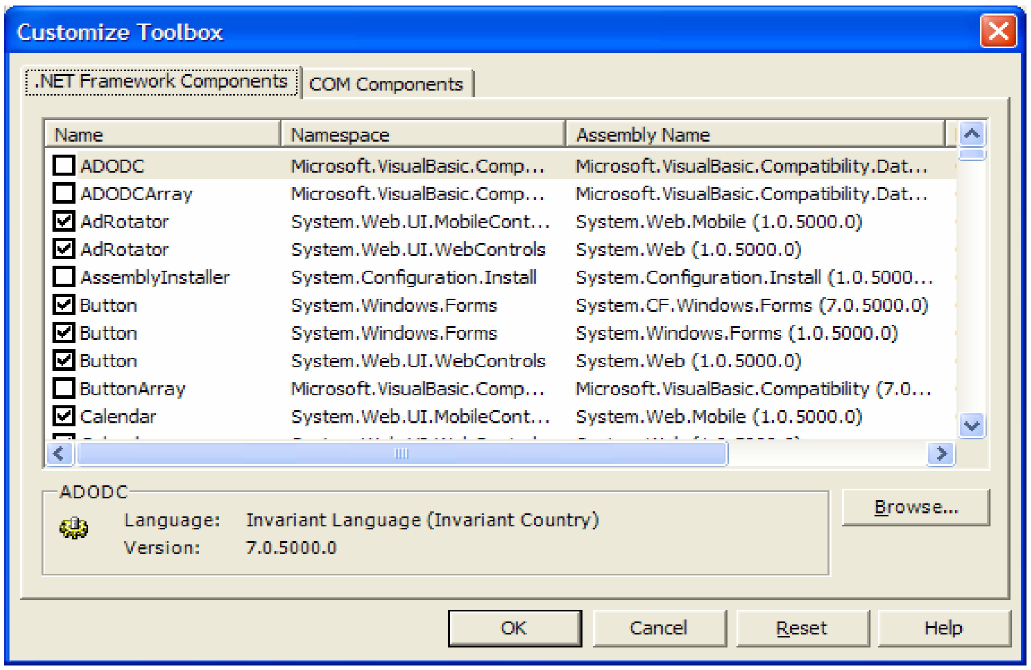


Then name it Phidgets



Open the tab and select Add Items

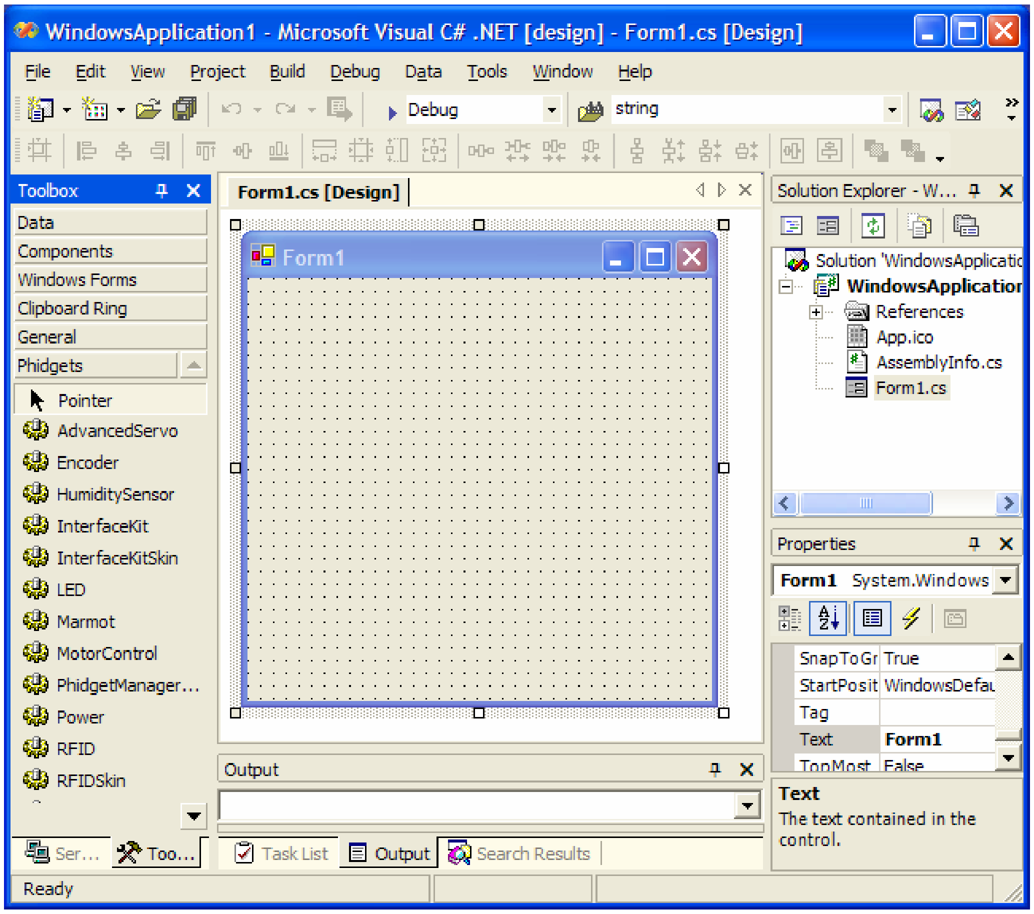
Using the Customize Toolbox, add the Grouplab.Phidget.dll. Click the Browse button (bottom right of the window on the following figure), and navigate to C:\Program Files\UofC\PhidgetSetup\. Select Grouplab.Phidgets.dll. All the phidgets components will now appear within your Customize Toolbox window, as shown below.



Name	Namespace	Assembly Name
☐ ADODC	Microsoft.VisualBasic.Comp...	Microsoft.VisualBasic.Compatibility.Dat...
☐ ADODCArray	Microsoft.VisualBasic.Comp...	Microsoft.VisualBasic.Compatibility.Dat...
☑ AdRotator	System.Web.UI.MobileCont...	System.Web.Mobile (1.0.5000.0)
☑ AdRotator	System.Web.UI.WebControls	System.Web (1.0.5000.0)
☐ AssemblyInstaller	System.Configuration.Install	System.Configuration.Install (1.0.5000...
☑ Button	System.Windows.Forms	System.CF.Windows.Forms (7.0.5000.0)
☑ Button	System.Windows.Forms	System.Windows.Forms (1.0.5000.0)
☑ Button	System.Web.UI.WebControls	System.Web (1.0.5000.0)
☐ ButtonArray	Microsoft.VisualBasic.Comp...	Microsoft.VisualBasic.Compatibility (7.0...
☑ Calendar	System.Web.UI.MobileCont...	System.Web.Mobile (1.0.5000.0)

ADODC
Language: Invariant Language (Invariant Country)
Version: 7.0.5000.0

Click the Browse button, and browse to the Grouplab.Phidget.dll and add it



The Phidgets Tab now lists a variety of Phidget components.

Appendix

2

A Catalog of Phidgets and Peripherals