CHAPTER 4

No Texting While Driving

This chapter walks you through the development of No Texting While Driving, an app that autoresponds to text messages you receive while you're driving. The app, first created with App Inventor by a beginning computer science student, is similar to a now-mass-produced app developed by State Farm Insurance. It is a prime example of how App Inventor provides access to some of the great features of the Android phone, including SMS text processing, database management, text-to-speech, and the location sensor.

In January 2010, the National Safety Council (NSC) announced the results of a study that found that at least 28 percent of all traffic accidents—close to 1.6 million crashes every year—are caused by drivers using cell phones, and at least 200,000 of those accidents occurred while drivers were texting.[1] As a result, many states have banned drivers from using cell phones altogether.

Daniel Finnegan, a student in the Fall 2010 session of the University of San Francisco App Inventor programming class, came up with a great app idea to help with the driving and texting epidemic. The app he created, which is shown in Figure 4-1, responds automatically (and hands-free) to any text with a message such as "I'm driving right now, I'll contact you shortly."

Figure 4-1. The No Texting While Driving app

1 *http://www.nsc.org/pages/nscestimates16millioncrashescausedbydriversusingcellphonesandtexting.aspx*

Some in-class brainstorming led to a few additional features that were developed for a tutorial posted on the App Inventor site:

The user can change the response for different situations
: For example, if you're going into a meeting or a movie instead of driving, the response can be modified accordingly.

The app speaks the text aloud
: Even if you know the app will autorespond, the jingle of incoming texts can kill you with curiosity.

The response message can contain your current location
: If your partner is at home making dinner, he or she would probably like to know how much longer your commute will last, without endangering you by having you answer the text.

Some weeks after the app was posted on the App Inventor site, State Farm Insurance created an Android app called "On the Move," which has similar functionality to No Texting While Driving.[2] The service is free to anyone, as part of State Farm's updated Pocket Agent→ for Android™ application, which the company announced in a YouTube video that can be found here: *http://www.youtube.com/watch?v=3xtjzO0-Hfw*.

We don't know if Daniel's app or the tutorial on the App Inventor site influenced "On the Move," but it's interesting to consider the possibility that an app created in a beginning programming course (by a creative writing student, no less!) might have inspired this mass-produced piece of software, or at least contributed to the ecosystem that brought it about. It certainly demonstrates how App Inventor has lowered the barrier of entry so that anyone with a good idea can quickly and inexpensively turn his idea into a tangible, interactive app.

What You'll Learn

This is a more complex app than those in the previous chapters, so you'll build it one piece of functionality at a time, starting with the autoresponse message. You'll learn about:

- The `Texting` component for sending texts and processing received texts.
- An input form for submitting the custom response message.
- The `TinyDB` database component for saving the customized message even after the app is closed.

2 *http://www.statefarm.com/aboutus/newsroom/20100819.asp*

- The **Screen.Initialize** event for loading the custom response when the app launches.
- The `Text-to-Speech` component for speaking the texts aloud.
- The `LocationSensor` component for reporting the driver's current location.

Getting Started

For this app to work, you need a text-to-speech module, *Text-To-Speech Extended*, on your phone. This module is included in Android version 2 or higher, but if you are running an Android 1.*x* operating system, you'll need to download it from the Android Market. On your phone:

1. Open the Market app.
2. Search for TTS.
3. Select the app *Text-To-Speech Extended* to install.

Once the Text-To-Speech module is installed, open it to test its features. When it opens, set the default language as desired. Then select "Listen to Preview." If you don't hear anything, make sure the volume on your phone is turned up. You can also change the way the voice sounds by changing the setting for the TTS Default Engine property.

After you've set up the Text-To-Speech module to your liking, connect to the App Inventor website and start a new project. Name it "NoTextingWhileDriving" (project names can't have spaces) and set the screen's title to "No Texting While Driving". Open the Blocks Editor and connect to the phone.

Designing the Components

The user interface for the app is relatively simple: it has a label that displays the automated response, along with a text box and a button for submitting a change. You'll also need to drag in a `Texting` component, a `TinyDB` component, a `TextToSpeech` component, and a `LocationSensor` component, all of which will appear in the "Non-visible components" area. You can see how this should look in the snapshot of the Component Designer shown in Figure 4-2.

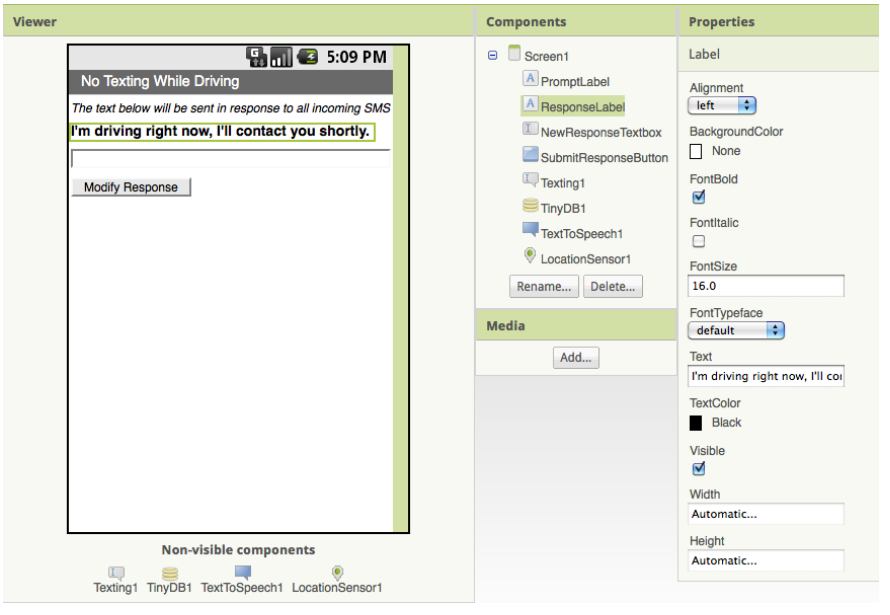

Figure 4-2. The No Texting While Driving app in the Component Designer

You can build the user interface shown in Figure 4-2 by dragging out the components listed in Table 4-1.

Set the properties of the components in the following way:

- Set the `Text` of `PromptLabel` to "The text below will be sent in response to all SMS texts received while this app is running."
- Set the `Text` of `ResponseLabel` to "I'm driving right now, I'll contact you shortly." Check its `boldness` property.
- Set the `Text` of `NewResponseTextbox` to "". (This leaves the text box blank for the user's input.)
- Set the `Hint` of `NewResponseTextbox` to "Enter new response text."
- Set the `Text` of `SubmitResponseButton` to "Modify Response."

Adding Behaviors to the Components

You'll start by programming the basic text autoresponse behavior, and then successively add more functionality.

Table 4-1. All the components for the No Texting While Driving app

Component type	Palette group	What you'll name it	Purpose
`Label`	Basic	`PromptLabel`	Let the user know how the app works.
`Label`	Basic	`ResponseLabel`	The response that will be sent back to the sender of original text.
`TextBox`	Basic	`NewResponseTextbox`	The user will enter the custom response here.
`Button`	Basic	`SubmitResponseButton`	The user clicks this to submit response.
`Texting`	Social	`Texting1`	Process the texts.
`TinyDB`	Basic	`TinyDB1`	Store the response in the database.
`TextToSpeech`	Other stuff	`TextToSpeech1`	Speak the texts aloud.
`LocationSensor`	Sensors	`LocationSensor1`	Sense where the phone is.

Programming an autoresponse

For the autoresponse behavior, you'll use App Inventor's `Texting` component. You can think of this component as a little person inside your phone that knows how to read and write texts. For reading texts, the component provides a **Texting .MessageReceived** event block. You can drag this block out and place blocks inside it to show what should happen when a text is received. In the case of this app, we want to automatically send back a prewritten response text.

To program the response text, you'll place a **Texting1.SendMessage** block within the **Texting1.MessageReceived** block. **Texting1.SendMessage** actually sends the text—so you'll first need to tell the component what message to send, and who to send it to, before calling **Texting1.SendMessage**. Table 4-2 lists all the blocks you'll need for this autoresponse behavior, and Figure 4-3 shows how they should look in the Blocks Editor.

Table 4-2. The blocks for sending an autoresponse

Block type	Drawer	Purpose
Texting1.MessageReceived	Texting	The event handler that is triggered when the phone receives a text.
set Texting1.PhoneNumber to	Texting	Set the `PhoneNumber` property before sending.
value number	My Definitions	The phone number of the person who sent the text.
set Texting1.Message to	Texting	Set the `Message` property before sending.
ResponseLabel.Text	ResponseLabel	The message the user has entered.
Texting1.SendMessage	Texting	Send the message.

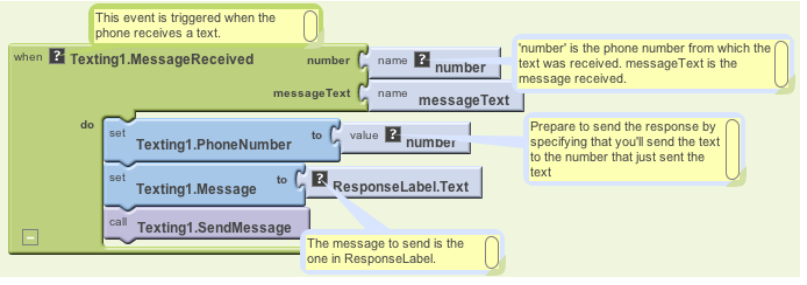

Figure 4-3. Responding to an incoming text

How the blocks work

When the phone receives a text message, the **Texting1.MessageReceived** event is triggered. As shown in Figure 4-3, the phone number of the sender is in the argument `number`, and the message received is in the argument `messageText`. For the autoresponse, the app needs to send a text message to the sender. To send the text, you first need to set the two key properties of the `Texting` component: `PhoneNumber` and `Message`. `Texting.PhoneNumber` is set to the number of the sender, and `Texting.Message` is set to the text you typed into `ResponseLabel`: "I'm driving right now, I'll contact you shortly." Once these are set, the app calls **Texting.SendMessage** to actually send the response.

You may be wondering about the yellow boxes that we have in the Blocks Editor. Those are *comments*, and you can add them by right-clicking a block and selecting Add Comment. Once you add a comment, you can show or hide it by clicking the black question mark that appears. You don't have to add comments in your app—we've simply included them here to help describe each block and what it does.

Most people use comments to document how they are building their app; comments explain how the program works, but they won't make the app behave differently. Comments are important, both for you as you build the app and modify it later, and for other people who might customize it. The one thing everyone agrees on about software is that it changes and transforms often. For this reason, commenting code is very important in software engineering, and especially so with open source software like App Inventor.

Test your app. *You'll need a second phone to test this behavior. If you don't have one, you can register with Google Voice or a similar service and send texts from that service to your phone.*

From the second phone, send a text to the phone running the app. Does the second phone receive the response text?

Entering a Custom Response

Next, let's add blocks so the user can enter her own custom response. In the Component Designer, you added a `TextBox` component named `NewResponseTextbox`; this is where the user will enter the custom response. When the user clicks on the `SubmitResponseButton`, you need to copy her entry (`NewResponseTextbox`) into the `ResponseLabel`, which is used to respond to texts. Table 4-3 lists the blocks you'll need for transferring a newly entered response into the `ResponseLabel`.

Table 4-3. Blocks for displaying the custom response

Block type	Drawer	Purpose
SubmitResponseButton .Click	SubmitResponseButton	The user clicks this button to submit a new response message.
set ResponseLabel.Text to	ResponseLabel	Move (set) the newly input value to this label.
NewResponseTextbox.Text	NewResponseTextbox	The user has entered the new response here.

How the blocks work

Think of how a typical input form works: you first enter something in a text box, and then click a submit button to tell the system to process it. The input form for this app is no different. Figure 4-4 shows how the blocks are programmed so that when the user clicks the `SubmitResponseButton`, the **SubmitResponseButton.Click** event is triggered.

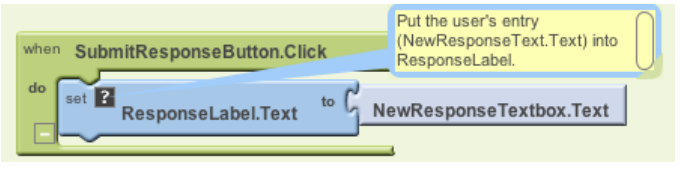

Figure 4-4. Setting the response to the user's entry

The event handler in this case copies (or *sets*, in programming terms) what the user has entered in `NewResponseTextbox` into the `ResponseLabel`. Recall that `ResponseLabel` holds the message that will be sent out in the autoresponse, so you want to be sure to place the newly entered custom message there.

Test your app. *Enter a custom response and submit it, and then use the second phone to send another text to the phone running the app. Was the custom response sent?*

Storing the Custom Response in a Database

You've built a great app already, with one catch: if the user enters a custom response, and then closes the app and relaunches it, the custom response will not appear (instead, the default one will). This behavior is not what your users will expect; they'll want to see the custom response when they restart the app. To make this happen, you need to store that custom response *persistently*.

You might think that placing data in the `ResponseLabel.Text` property is technically "storing" it, but the issue is that data stored in component properties is *transient* data. Transient data is like your short-term memory; the phone "forgets" it as soon as an app closes. If you want your app to remember something *persistently*, you have to transfer it from short-term memory (a component property or variable) to long-term memory (a database).

To store data persistently, you'll use the `TinyDB` component, which stores data in a database that's already on the Android device. `TinyDB` provides two functions: `StoreValue` and `GetValue`. The former allows the app to store information in the device's database, while the latter lets the app retrieve information that has already been stored.

For many apps, you'll use the following scheme:

1. Store data to the database each time the user submits a new value.
2. When the app launches, load the data from the database into a variable or property.

You'll start by modifying the **SubmitResponseButton.Click** event handler so that it stores the data persistently, using the blocks listed in Table 4-4.

Table 4-4. Blocks for storing the custom response with TinyDB

Block type	Drawer	Purpose
TinyDB1.StoreValue	TinyDB1	Store the custom message in the phone's database.
text ("responseMessage")	Text	Use this as the tag for the data.
ResponseLabel.Text	ResponseLabel	The response message is now here.

How the blocks work

This app uses **TinyDB** to take the text it just put in `ResponseLabel` and store it in the database. As shown in Figure 4-5, when you store something in the database, you provide a tag with it; in this case, the tag is "responseMessage." Think of the tag as the name for the data's spot in the database; it uniquely identifies the data you are storing. As you'll see in the next section, you'll use the same tag ("responseMessage") when you load the data back in from the database.

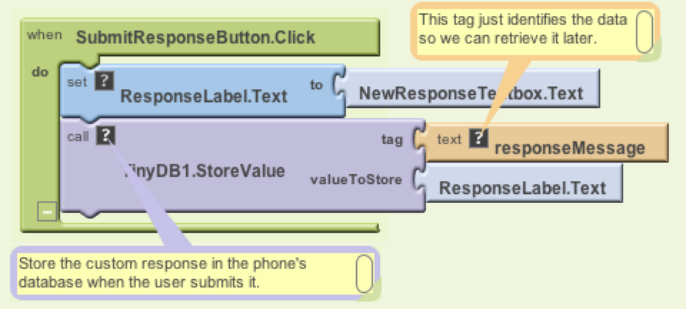

Figure 4-5. Storing the custom response persistently

Retrieving the Custom Response When the App Opens

The reason for storing the custom response in the database is so it can be loaded back into the app the next time the user opens it. App Inventor provides a special event block that is triggered when the app opens: **Screen1.Initialize** (if you completed MoleMash in Chapter 3, you've seen this before). If you drag this event block out and place blocks in it, those blocks will be executed right when the app launches.

For this app, your **Screen1.Initialize** event handler should check to see if a custom response has been put in the database. If so, the custom response should be loaded in using the **TinyDB.GetValue** function. The blocks you'll need for this are shown in Table 4-5.

Table 4-5. Blocks for loading the data back in when the app is opened

Block type	Drawer	Purpose
def variable ("response")	Definition (don't forget: this is different than the My Definitions drawer)	A temporary variable to hold the retrieved data.
text ("")	Text	The initial value for the variable can be anything.
Screen1.Initialize	Screen1	This is triggered when the app begins.
set global response to	My Definitions	Set this variable to the value retrieved from the database.
TinyDB1.GetValue	TinyDB1	Get the stored response text from the database.
text ("responseMessage")	Text	Plug this into the tag slot of **TinyDB .GetValue**, making sure the text is the same as that used in **TinyDB.Store Value** earlier.
if	Control	Ask if the retrieved value has some text.
>	Math	Check if the length of the retrieved value is greater than (>) 0.

Table 4-5. Blocks for loading the data back in when the app is opened (continued)

Block type	Drawer	Purpose
length text	Text	Check if the length of the retrieved value is greater than 0.
global response	My Definitions	This variable holds the value retrieved from **TinyDB1.GetValue**.
number (0)	Math	Compare this with the length of the response.
set ResponseLabel.Text to	ResponseLabel	If we retrieved something, place it in `ResponseLabel`.
global response	My Definitions	This variable holds the value retrieved from **TinyDB1.GetValue**.

How the blocks work

The blocks are shown in Figure 4-6. To understand them, you must envision a user opening the app for the first time, entering a custom response, and opening the app subsequent times. The first time the user opens the app, there won't be any custom response in the database to load, so you want to leave the default response in the `ResponseLabel`. On successive launches, you want to load the previously stored custom response from the database and place it in the `ResponseLabel`.

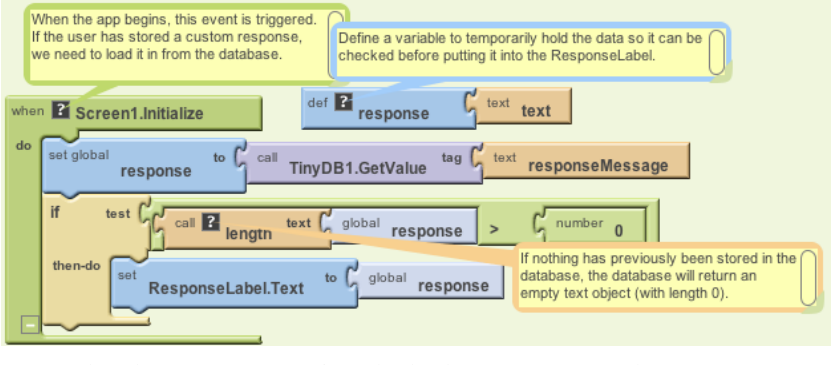

Figure 4-6. Loading the custom response from the database upon app initialization

When the app begins, the **Screen1.Initialize** event is triggered. The app calls the **TinyDB1.GetValue** with a tag of "responseMessage," the same tag you used when you stored the user's custom response entry earlier. The retrieved value is placed in the variable `response` so that it can be checked before we place it as the `ResponseLabel`. Can you think of why you'd want to check what you get back from the database before displaying it as the custom message to the user?

TinyDB returns empty text if there is no data for a particular tag in the database. There won't be data the first time the app is launched; this will be the case until the user enters a custom response. Because the variable response now holds the returned value, we can use the **if** block to check if the length of what was returned by the database is greater than 0. If the length of the value contained in response is greater than 0, the app knows that TinyDB did return something, and the retrieved value can be placed into the ResponseLabel. If the length isn't greater than 0, the app knows there is no previously stored response, so it doesn't modify the ResponseLabel (leaving the default response in it).

Test your app. *You cannot test this behavior through live testing, as the database gets emptied each time you "Connect to Device" to restart the app.*

Instead, select "Package for Phone"→Show Barcode, and then download the app to your phone by scanning the barcode. Once the app is installed, enter a new response message in the NewResponseTextbox *and click the* SubmitResponseButton. *Then close the app and restart it. Does your custom message appear?*

Speaking the Incoming Texts Aloud

In this section, you'll modify the app so that when you receive a text, the sender's phone number, along with the message, is spoken aloud. The idea here is that when you're driving and hear a text come in, you might be tempted to check the text even if you know the app is sending an autoresponse. With text-to-speech, you can hear the incoming texts and keep your hands on the wheel.

Android devices provide text-to-speech capabilities and App Inventor provides a component, TextToSpeech, that will speak any text you give it. (Note that here "text" is meant in the general sense of the word—a sequence of letters, digits, and punctuation—not an SMS text.)

In the "Getting Started" section of this app, we asked you to download a text-to-speech module from the Android Market. If you didn't do so then, you'll need to now. Once that module is installed and configured as desired, you can use the TextToSpeech component within App Inventor.

The TextToSpeech component is very simple to use—you just call its Speak function and plug in the text you want spoken into its *message* slot. For instance, the function shown in Figure 4-7 would say, "Hello World."

Figure 4-7. Blocks for speaking "Hello World" aloud

For the No Texting While Driving app, you'll need to provide a more complicated message to be spoken, one that includes both the text received and the phone number of the person who sent it. Instead of plugging in a static text object like the "Hello World" text block, you'll plug in a **make text** block. An incredibly useful function, **make text** allows you to combine separate pieces of text (or numbers and other characters) into a single text object.

You'll need to make the call to **TextToSpeech.Speak** within the **Texting.Message Received** event handler you programmed earlier. The blocks you programmed previously handle this event by setting the `PhoneNumber` and `Message` properties of the `Texting` component appropriately and then sending the response text. You'll extend that event handler by adding the blocks listed in Table 4-6.

Table 4-6. Blocks for speaking the incoming text aloud

Block type	Drawer	Purpose
TextToSpeech1.Speak	TextToSpeech1	Speak the message received out loud.
make text	Text	Build the words that will be spoken.
text ("SMS text received from")	Text	The first words spoken.
value number	My Definitions	The number from which the original text was received.
text (".The message is")	Text	Put a period in after the phone number and then say, "The message is."
value messageText	My Definitions	The original message received.

How the blocks work

After the response is sent, the **TextToSpeech1.Speak** function is called, as shown at the bottom of Figure 4-8. You can plug any text object into the message slot of the **TextToSpeech1.Speak** function. In this case, **make text** is used to build the words to be spoken—it *concatenates* (or adds) together the text "SMS text received from" and the phone number from which the message was received (**value number**), plus the text ".The message is," and finally the message received (**value messageText**). So, if the text "hello" was sent from the number "111–2222," the phone would say, "SMS text received from 111–2222. The message is hello."

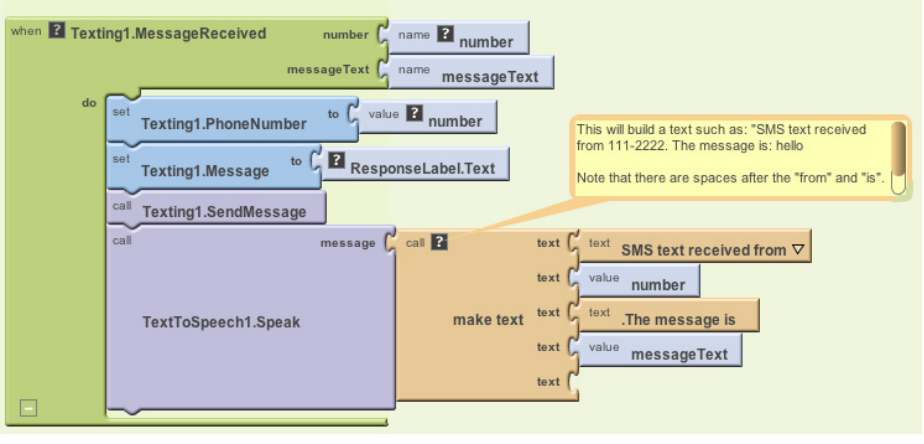

Figure 4-8. Speaking the incoming text aloud

Test your app. *You'll need a second phone to test your app. From the second phone, send a text to the phone running the app. Does the phone running the app speak the text aloud? Does it still send an automated response?*

Adding Location Information to the Response

Apps like Facebook's Place and Google's Latitude use GPS information to help people track one another's location. There are major privacy concerns with such apps, one reason being that location tracking kindles people's fear of a "Big Brother" apparatus that a totalitarian government might set up to track its citizens' whereabouts. But apps that use location information can be quite useful—think of a lost child, or hikers who've gotten off the trail in the woods.

In the No Texting While Driving app, location tracking can be used to convey a bit more information in response to incoming texts. Instead of just "I'm driving," the response message can be something like "I'm driving and I'm at 3413 Cherry Avenue." For someone awaiting the arrival of a friend or family member, this extra information can be helpful.

App Inventor provides the `LocationSensor` component for interfacing with the phone's GPS (or *geographical positioning system*). Besides latitude and longitude information, the `LocationSensor` can also tap into Google Maps to provide the driver's current street address.

It's important to note that `LocationSensor` doesn't always have a reading. For this reason, you need to take care to use the component properly. Specifically, your app should respond to the **LocationSensor.LocationChanged** event handler. A **LocationChanged** event occurs when the phone's location sensor first gets a reading, and when the phone is moved to generate a new reading. Using the blocks listed in Table 4-7, our scheme will respond to the **LocationChanged** event by placing the current address in a variable we'll name **lastKnownLocation**. Later, we'll change the response message to incorporate the address we get from this variable.

Table 4-7. Blocks to set up the location sensor

Block type	Drawer	Purpose
def variable ("lastKnownLocation")	Definitions	Create a variable to hold the last read address.
text ("unknown")	Text	Set the default value in case the phone's sensor is not working.
LocationSensor1.LocationChanged	LocationSensor1	This is triggered on the first location reading and every location change.
set global lastKnownLocation to	My Definitions	Set this variable to be used later.
LocationSensor1.CurrentAddress	LocationSensor1	This is a street address such as "2222 Willard Street, Atlanta, Georgia."

How the blocks work

The **LocationSensor1.LocationChanged** event is triggered the first time the sensor gets a location reading and when the device is moved so that a new reading is generated. Since you eventually want to send a street address as part of the response message, Figure 4-9 shows how the **LocationSensor1.CurrentAddress** function is called to get that information and store it in the `lastKnownLocation` variable. Behind the scenes, this function makes a call to Google Maps (via an *API*, something you'll learn about in Chapter 24) to determine the closest street address for the latitude and longitude that the sensor reads.

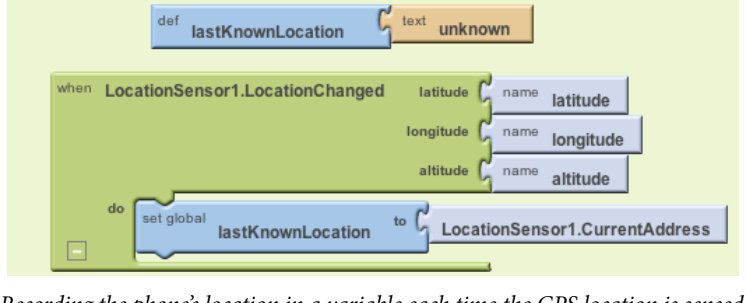

Figure 4-9. Recording the phone's location in a variable each time the GPS location is sensed

Note that with these blocks, you've finished only half of the job. The app still needs to incorporate the location information into the autoresponse text that will be sent back to the sender. Let's do that next.

Sending the Location As Part of the Response

Using the variable `lastKnownLocation`, you can modify the **Texting1.MessageReceived** event handler to add location information to the response. Table 4-8 lists the blocks you'll need for this.

Table 4-8. Blocks to display location information in the autoresponse

Block type	Drawer	Purpose
make text	Text	If there is a location reading, build a compound text object.
ResponseLabel.Text	MessageTextBox	This is the (custom) message in the text box.
text ("My last known location is:")	Text	This will be spoken after the custom message (note the leading space).
global lastKnownLocation	LocationSensor	This is an address such as "2222 Willard Street, Atlanta, Georgia."

How the blocks work

This behavior works in concert with the **LocationSensor1.LocationChanged** event and the variable `lastKnownLocation`. As you can see in Figure 4-10, instead of directly sending a message containing the text in `ResponseLabel.Text`, the app first builds a message using **make text**. It combines the response text in `ResponseLabel.Text` with the text "My last known location is:" followed by the variable `lastKnownLocation`.

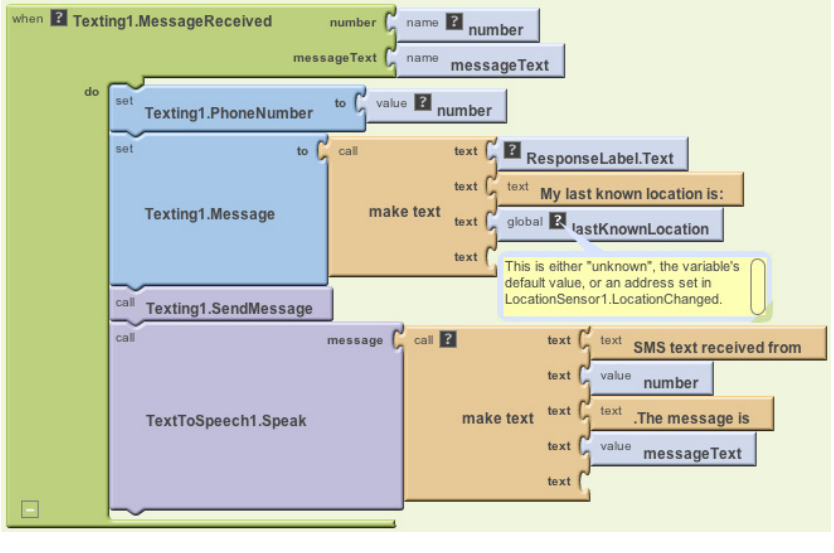

Figure 4-10. Including location information in the response text

The default value of **lastKnownLocation** is "unknown," so if the location sensor hasn't yet generated a reading, the second part of the response message will contain the text "My last known location is: unknown." If there has been a reading, the second part of the response will be something like "My last known location is: 876 Willard Street, San Francisco, CA 95422."

Test your app. *From the second phone, send a text to the phone running the app. Does the second phone receive the response text with the location information? If it doesn't, make sure you've turned GPS on in the first phone's Location settings.*

The Complete App: No Texting While Driving

Figure 4-11 shows the final block configuration for No Texting While Driving.

Variations

Once you get the app working, you might want to explore some variations. For example:

- Write a version that lets the user define custom responses for particular incoming phone numbers. You'll need to add conditional (`if`) blocks that check for those numbers. For more information on conditional blocks, see Chapter 18.
- Write a version that sends custom responses based on whether the user is within certain latitude/longitude boundaries. So, if the app determines that you're in room 222, it will send back "Bob is in room 222 and can't text right now." For more information on the `LocationSensor` and determining boundaries, see Chapter 23.
- Write a version that sounds an alarm when a text is received from a number in a "notify" list. For help working with lists, see Chapter 19.

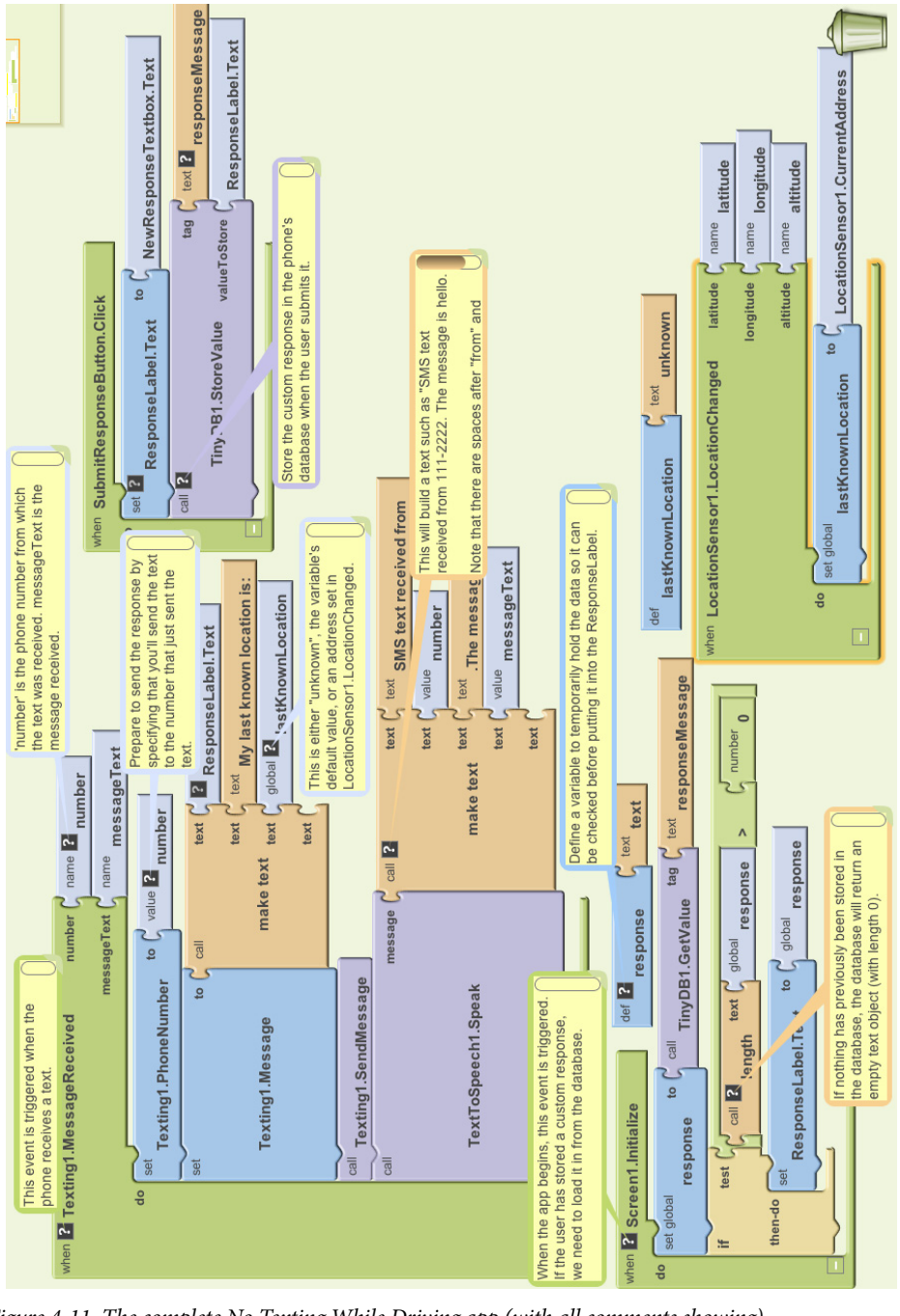

Figure 4-11. The complete No Texting While Driving app (with all comments showing)

Summary

Here are some of the concepts we've covered in this tutorial:

- The `Texting` component can be used to both send text messages and process the ones that are received. Before calling **Texting.SendMessage**, you should set the `PhoneNumber` and `Message` properties of the `Texting` component. To respond to an incoming text, program the **Texting.MessageReceived** handler.
- The `TinyDB` component is used to store information persistently—in the phone's database—so that the data can be reloaded each time the app is opened. For more information on `TinyDB`, see Chapter 22.
- The `TextToSpeech` component takes any text object and speaks it aloud.
- **make text** is used to piece together (or concatenate) separate text items into a single text object.
- The `LocationSensor` component can report the phone's latitude, longitude, and current street address. To ensure that it has a reading, you should access its data within the **LocationSensor.LocationChanged** event handler, which is triggered the first time a reading is made and upon every change thereafter. For more information on the `LocationSensor`, see Chapter 23.

If you're interested in exploring SMS-processing apps further, check out the Broadcast Hub app in Chapter 11.