ECE155: Engineering Design with Embedded Systems Winter 2016

Lecture 6 — Events, Polling vs. Interrupts

Jeff Zarnett & Patrick Lam

Event-Driven Programming

We're going to talk about events in today's lecture, because we use the *event-driven paradigm* to receive sensor data in the labs. Other programming paradigms for embedded systems exist, e.g. the superloop structure.

Events are applicable to many application areas beyond embedded systems, notably graphical user interfaces (e.g. mouse clicks, keystrokes are typical events).

An *event* is a notification of a change to the state of your system, such as a button click.

Typically, you don't create events yourself; the system creates them (perhaps in response to notifications from hardware devices—sensors, timers, or the processor itself) and calls your *event listener*, telling you that an event occurred. You just wait for events to occur and react to them. Event-driven software is reactive, not proactive: traditionally, events come to you. This represents an inversion of control. (You may, however, schedule an event to be delivered at some point in the future, which includes "right now".)

Why would you use an event-driven programming model? It can help you better structure your code, and it's more efficient when you have diverse event handlers rather than looping through each possible event and checking to see if it happened.

Event driven programming analogy: you pay your bills only once the mail carrier brings them to your door.

Here's how to do event-driven programming for the click event, then.

- To receive click events:
  - the application registers an event listener with the object representing the button.
    `go.setOnClickListener(...);`
- When the user clicks the button:
  - the system executes the click event listener.

Implementing Event Listeners. We've seen that the application has to call `setOnClickListener()` on the button. What does it pass to that method? A `View.OnClickListener` object.

You could declare one:

```
class MyClickListener
      extends View.OnClickListener {
  public void onClick(View v) {
    Log.d("A2", "clicked!");
  }
}
```

```
// (somewhere else:)
go.setOnClickListener(new MyClickListener());
```

But, there's a better way:

```
go.setOnClickListener(new View.OnClickListener() {
  public void onClick(View v) {
    Log.d("A2", "clicked!");
  }
  });
```

This is called an *inner class*.

Advantages of Inner Classes.

- They don't litter your code with one-time-use classes.
- They can access fields and (final) local variables.

```
class MainActivity {
  int i;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    Button go = (Button) findViewById(R.id.go);
    final int j = 2;
    go.setOnClickListener(new View.OnClickListener() {
      public void onClick(View v) {
        Log.d("A2", "i is "+i+" and j is "+j);
      }
      });
  }
}
```

An Alternative to Inner Classes. You have another option. From the Android documentation [Gui13], you could use the following in your Activity XML:

```
  <Button
     android:layout_height="wrap_content"
     android:layout_width="wrap_content"
     android:text="@string/self_destruct"
     android:onClick="selfDestruct" />
```

Then, in your activity, you must include the method:

```
  public void selfDestruct(View view) {
     // Kaboom
  }
```

Callback methods. Applications sometimes need a particular method to be called in the future. *Callback methods* are a way of implementing this design. For instance, event handlers should be called when the appropriate event occurs. One use of callback methods is to structure applications with time-consuming tasks:

- The application registers a callback to run upon completion (or near-completion) of the time-consuming task.
- The application spawns the time-consuming task (perhaps in a different thread), and doesn't wait for it to finish.
- The application continues normally, forgetting about the time-consuming task for now.
- Once the time-consuming task finishes (or is about to finish), the callback executes, notifying the main application about the completion.

Callback methods therefore permit asynchronous (non-blocking) execution.

Can you name a real-life analogy to the workflow described above?

Synchronous vs. Asynchronous Execution. The programs we've seen so far are *synchronous* (or sequential): each instruction executes in sequence, and an instruction can only execute after its predecessor completes. In particular, when a program calls a function, it waits for the function to return before continuing with its own execution, no matter how long the function takes to complete.

Because the application is only doing one thing at a time, it is much easier to understand than asynchronous programs.

It is also possible to write *asynchronous* (or concurrent) programs. In these programs, a main program may spawn a function, or task, and continue executing before the function returns.

Writing asynchronous programs permits higher performance on modern multi-core architectures; for some application domains, concurrency is also a better way to structure the code.

Miscellaneous Comments about Events

Here are a couple notes about events that you might want to know.

Priorities. Some problems are more important to deal with right away than others, e.g.:

- your mom calls;
- supper is burning; and
- the laundry is done.

Which would you do first?

What about events with similar priorities? How would you choose? Do you always need to choose?

Events and finite-state machines. One way of implementing functionality in event-driven software is via finite-state machines (FSMs), containing an infinite loop between states. You'll see FSMs in ECE124 this term. FSMs consume events and update system state. You may choose to implement your signal processing code using an FSM.

Polling versus Interrupts

For most of this course, you're just going to assume that events come at you from nowhere. We'll briefly look at sources of events to get a better understanding of event-driven programming.

One source of events is through polling the sensors every so often; sensor readings then generate events. Or, important events may interrupt the processor to let you know about their existence. It's also possible to use both polling and interrupts at the same time.

Polling

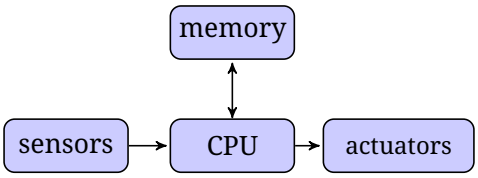
Polling means that the processor requests readings from the device at its convenience. (e.g. what is the current light level?) A polled device does not impose any schedule on the processor, so we can call this *passive synchronization*.

When should a processor check device readings?

- whenever convenient (occasional polling);
- at fixed time intervals (periodic polling);
- constantly (tight polling).

Any of these strategies may be appropriate (with different constants), depending on the importance of the event and the frequencies at which events occur.

Digression: Input/Output. Recall this picture from lecture 1:



How does the CPU actually talk to the sensors and actuators? Two methods: 1) memory-mapped I/O and 2) port-mapped I/O, or special instructions. In memory-mapped I/O, the CPU executes what it thinks are reads and writes to memory. In particular, it sends out the appropriate requests on the system bus. Devices listen on the bus and manufacture the appropriate responses. For special instructions, as seen on Intel ia32 processors, the CPU instead executes special `in` and `out` instructions, which may transmit data on a special bus, or set a specific signal on the bus.

Pseudocode for Tight Polling Loop. Here is pseudocode for a memory-mapped I/O system.

```
while( statusRegister == 0x0000 ) {
    // Do nothing until statusRegister changes value
```

```
}
//  Read data that has changed from a dataRegister and store in memory
incomingData = dataRegister;
```

We'd expect this loop to terminate based on some hardware specification promising `statusRegister` eventually becoming non-zero due to an external event. Data exchange occurs once the device indicates that it is ready to emit data (by setting `statusRegister`); we can call this *polling synchronization*.

Interrupts

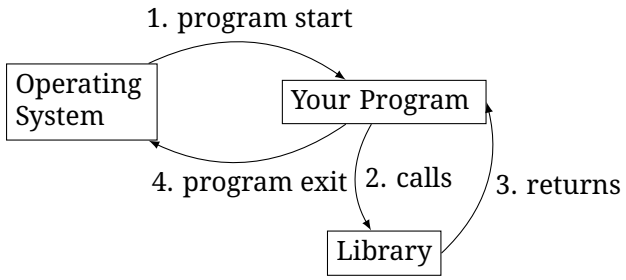
"In Soviet Russia, event polls you."

Another way that a processor can find out about an event is via an *interrupt*. Interrupts *actively synchronize* a device and a processor. An interrupt tells the processor one bit of information: that something worth knowing about (high-priority) is occurring. When the processor gets an interrupt, it stops what it's currently doing, saves its state, and starts executing a pre-defined *interrupt handler*. The interrupt handler will typically read the event information (how?) and store it somewhere accessible. After the handler returns, the processor restores its state and resumes what it was doing before.

Inversion of Control

Android programming is event-driven programming, which changes the basic structure of a program from what you saw in ECE150.

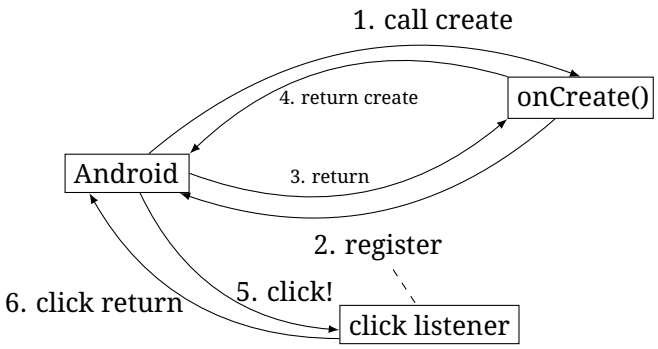
Here's a representation of how a program works, as you saw it in ECE150:



The main idea is that when the user chooses to launch your program, you have all the control until you exit. If you want some input, you'll ask the library/operating system for the input (using a system call), and it'll return it to you when it returns.

Note that calling the library does not lead, by itself, to an inversion of control.

New Event-Driven Paradigm. Here's an example of the new paradigm.



Now, Android takes all of the responsibility for calling your code when appropriate. It'll start your Activity by calling the `onCreate()` method. In that method, you may choose to register event listeners, or programmatically add widgets. But you only do setup work there. Don't do any real computation.

Next, when something happens, Android is going to call you and let you know about it. It's doing something that looks like this:

```
while (!done) {
 r <- fetch Runnable from Queue
 dispatch r
}
```

This is like a tight polling loop, but it goes to sleep between events (in the fetch).

References

[Gui13] Android Developer Guide. Android reference: Widgets: Button. `http://developer.android.com/reference/android/widget/Button.html`, 2013.