**Eventually**

**An Arduino Event-based Programming Library**

Arduino programming is a bit of a bait-and-switch. At first, people are like, "look, it's super-easy - see my blink program!" Then, when it comes to writing anything with even the most minute complexity - with states and input and output - one has to switch to a horrid event-programming mindset with thousands of state variables everywhere.

Then, when you have to debounce the buttons, it gets even worse!

The goal of Eventually is to make a more event-oriented environment for Arduino programming. To make it *actually* easy to use to build worthwhile projects.

**Writing a Simple Eventually Program**

To show the program in action, we will write a program that blinks a light,when a button is pushed.

To use Eventually, you need to download the library and then include it in your code (Sketch -> Include Library). That should put the following line at the top of your program:

#include <Eventually.h>

Next, you need to include a global variable for the event manager. I usually call this "mgr":

EvtManager mgr;

Eventually programs do not use the loop() function like other Arduino programs. Instead, remove the auto-generated loop() function and in its place put in this line of code (it SHOULD NOT be within a function) to use our event manager instead of the normal Arduino loop (notice that there is NO SEMICOLON at the end):

USE\_EVENTUALLY\_LOOP(mgr)

Now, during our setup() function, instead of just doing pinModes, you also need to setup one or more listeners. The two primary types of listeners for Eventually are EvtPinListener and EvtTimeListener. We are going to use both. We will use the pin listener to tell us when the button is pushed and the time listener to blink the light. Therefore, our setup function will look like this:

#define LIGHT\_PIN 5

#define BUTTON\_PIN 3

void setup() {

/\* Set the pin modes \*/

pinMode(LIGHT\_PIN, OUTPUT);

pinMode(BUTTON\_PIN, INPUT);

/\* Add a button listener \*/

mgr.addListener(new EvtPinListener(BUTTON\_PIN, (EvtAction)start\_blinking));

}

Alternatively, if our BUTTON\_PIN uses INPUT\_PULLUP:

void setup() {

// ...

/\* Button is LOW when pushed, so use INPUT\_PULLUP \*/

pinMode(BUTTON\_PIN, INPUT\_PULLUP);

/\* Add a button listener, listening for LOW \*/

/\* (To use this constructor, we also have to specify the debounce delay) \*/

mgr.addListener(new EvtPinListener(BUTTON\_PIN, 40, LOW, (EvtAction)start\_blinking));

}

This creates a listener for the given button pin, and will call start\_blinking when the button is pushed. Note that by default, EvtPinListeners will automatically debounce the pin, and also make sure that it starts in the opposite state before beginning listening (though both of these are configurable).

Also note that we put (EvtAction) in front of our function name. This makes sure the compiler knows that this will be used as an event action. It may not compile without this marking.

Now, when the button is pushed, we need start\_blinking to install a new time-based listener to blink the light, and another button listener to stop the blinking. So the code will look like this:

bool start\_blinking() {

mgr.resetContext();

mgr.addListener(new EvtTimeListener(500, true, (EvtAction)blink\_pin));

mgr.addListener(new EvtPinListener(BUTTON\_PIN, (EvtAction)stop\_blinking));

return true;

}

This starts by calling resetContext(), which will uninstall all current listeners to prepare for a new set. Then, it adds a time listener. The first parameter to the EvtTimeListener constructor is the number of milliseconds to wait until firing. In this case, 500 (half a second). The second parameter is whether or not to continually fire. If it is set to false, it will fire once and then be done. The last parameter is the function to call.

Then it adds a pin listener to call the stop\_blinking function when pushed. All triggered actions need to return true or false. "true" stops the current action chain and is usually what you want.

Now, let's look at the blink\_pin function:

bool blink\_state = LOW;

bool blink\_pin() {

blink\_state = !blink\_state;

digitalWrite(LIGHT\_PIN, blink\_state);

return false;

}

This creates a global variable that tells us the blink state, and whenever this is called, it alternates the blink state and writes that to the pin. This one returns false because, since the blinking doesn't affect the action of other pins, the event chain should continue.

Finally, we need a stop\_blinking() function. All this will do is create a new listener to wait for it to start again:

bool stop\_blinking() {

mgr.resetContext();

mgr.addListener(new EvtPinListener(BUTTON\_PIN, (EvtAction)start\_blinking));

return true;

}

Now you are ready to go!

**C++ Issues**

One of the goals of this library was to make it use the flexibility of C++ without the user having to know a lot of C++. If you don't know C++, just use the listeners in the basic way mentioned. However, if you want to do more advanced things, this section covers the C++ issues.

One main goal is to take advantage of the flexibility available with C++ pointers but without exposing the user to them. Therefore, listeners are always created with the "new" keyword, but they are immediately added to the manager (technically, the context) which then manages it. Once the listener is added to the manager/context, then the responsibility for its memory management goes to the manager/context. Note that at this time, there is no ability for user code to detect when a listener gets destroyed, so if you add data to a listener (see section below), then it should not rely on the listener to destroy the data at the right time.

**Writing Your Own Listeners**

If you need a more advanced listener, you can write your own fairly easily. You need to create a subclass of EvtListener and add (a) a constructor, (b) a setupListener() method, and (c) an isEventTriggered() method.

The constructor simply takes whatever arguments control the listener's behavior. The setupListener() method gets called when the listener is added to the chain. This should handle doing things such as sampling the current time or pin states that you will need to determine if the event is triggered.

The isEventTriggered() method returns true if the event is triggered.

Below is a simple class that simply always fires:

class EvtAlwaysFiresListener : public EvtListener {

public:

EvtAlwaysFiresListener();

void setupListener();

bool isEventTriggered();

}

EvtAlwaysFiresListener::EvtAlwaysFiresListener() {

// Nothing to construct

}

void EvtAlwaysFiresListener::setupListener() {

// Nothing to setup

}

bool EvtAlwaysFiresListener::isEventTriggered() {

return true;

}

**Adding Data to Listeners**

This library also has the ability to add small bits of data to listeners. Each listener had a "data" member, which allows arbitrary data stored into it (type is a pointer to void, and the value defaults to 0). The data variable is completely unmanaged, so use it for whatever you want. It doesn't have to store a pointer, in fact, it is probably easier to manage if it doesn't.

So, for instance, we can rewrite the blink\_pin function to not rely on a global variable, like this:

bool blink\_pin(EvtListener \*lstn) {

lstn->data = !lstn->data;

digitalWrite(LIGHT\_PIN, lstn->data);

return false;

}

This uses the data variable to store the pin state. Note that our action function changed signature as well. The actual signature for the action function is (EvtListener \*, EvtContext \*). However, the C++ stack will allow you to have the function with fewer arguments, as long as they are in the right order. That is why we have to put (EvtAction) in front of our action functions - to get it to the right type. Note that if we want access to more parts of our listener, we can also use the subclass type instead of just EvtListener if we know which one it will be.

**Future Expansion**

The two biggest features I want to add are (a) support for multiple contexts, and (b) declarative creation of a state machine. There is some code in there for (a), but it is not ready to use.