LINUX VOICE TUTORIAL

CODE NINJA: CALLBACKS AND NON-BLOCKING IO

BEN EVERARD

Nobody likes to sit around doing nothing – here's how your machine carries on processing while it's waiting for something to do.

WHY DO THIS?

- Speed up your code by avoiding unnecessary waits
- Learn to use one of the trendiest programming platforms (Node.js)
- Build your own website testing tool to find out which sites load fastest

When people learn to program, they're often taught that it involves listing a series of steps that the computer takes one after another until it's solved the problem.

Take, for example, the following pseudo code:

```
print "what is your name"
name = input("?")
print "hello", name
```

First it prints out a question, then it calls the function **input()**, which waits for the user to enter their name, and then returns this as a string. Finally, the computer says hello. This approach works fine, and we could adjust the program to ask lots of questions about the user to get any information we wanted.

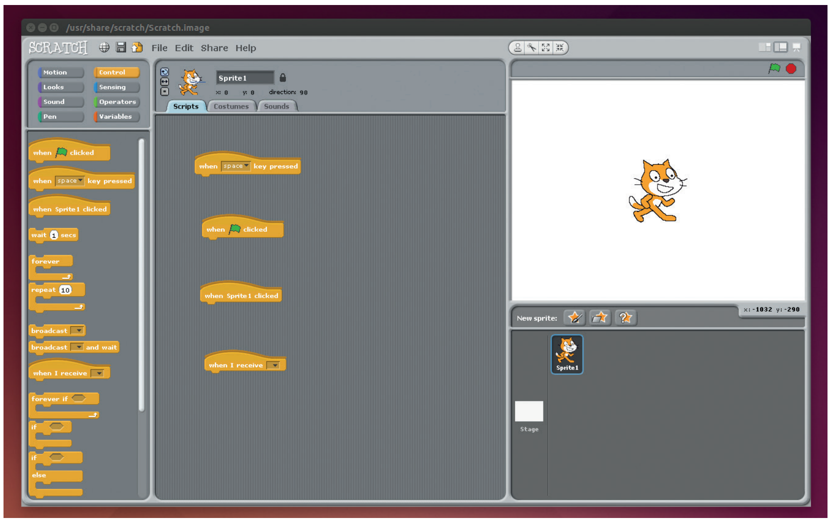
The input is what we call blocking, which means it stops the program running until it gets a response from the user. Now, let's look at another program:

```
print "what is your name"
name = input(?)
picture = get_pic_from_server(name)
biography = get_biography_from_database(name)
work = get_work_from_file(name)
```

Here, it asks for your name, then it gets three pieces of information about you: your picture, your biography and your work. We haven't defined the functions that get this data, but we'll say that they're also blocking.

The process then will be:

1. Get your name
2. Wait for the server to return your picture
3. Wait for the database to return your biography
4. Wait for the filesystem to return your work
5. Continue processing



Scratch's "When ..." code blocks are assigning callbacks to particular events.

That means the computer waits three times. Depending on what system you're running on, it could mean quite a long delay between asking for a name and continuing processing. For most of the time that this program is running the processor will be sitting idle, so it's a waste of resources as well.

A better way to run this would be to send all three requests for data at the same time, and then process the data as it comes back so that whichever data comes back first could be processed first. Doing it this way means the computer only has to wait once. This gives us a problem though, because it means that our program won't simply run one line after the next. We need some new way to define the order for the code to run in, because you can't just run all the lines of your program at the same time and hope that it all works.

Callbacks

The solution to this problems is to create functions that should run once a specific event occurs – such as when a particular file is returned from a server. These functions are known as callbacks, and they allow you to run non-blocking code, yet still define the order in which you want code to be processed. For example, take a look at the following pseudo code:

```
get_picture_from_server(function_to_run_when_picture_
returned)
```

When the computer got to this line, it would request the picture from the server. However, if the function **get_picture_from_server** is non-blocking, the computer won't wait for the picture to be returned; it'll just keep processing the main code. Once the server has processed the request and returned the picture, the computer will pause processing the main code and run **function_to_run_when_picture_returned** (the callback function). When this function is completed, it resumes processing the main code. This way the computer isn't wasting time while it waits for the server to respond, because it carries on processing the main program, yet it can still process the file when it is returned.

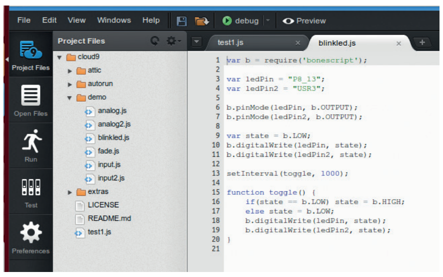
Let's switch this from pseudo code to real code and have a look at how it works in practice. Many languages have the ability to use callbacks, but they are used extensively in JavaScript. Almost all JavaScript has callbacks in one form or another, whether it's to run code when the user performs a particular action, or to handle loading data over unpredictable internet connections. These callbacks

Node.js

For people who think of JavaScript as a way of introducing some simple processing power to web pages, it may seem a little weird to use it in server-side programs. Despite that little bit of cognitive dissonance, Node.js provides a really powerful way of building back-end services using a non-blocking IO, and has dramatically risen in popularity over the last few years. It works particularly well when acting as the glue that holds a load of data sources together, and is used to power a lot of data-intensive web apps.

This isn't its only useful application though. It's also used as the programming language of choice for the BeagleBone single-board computers. This may seem a little odd at first, but it enables you to connect your BeagleBone to your computer via a USB cable. The BeagleBone then runs a web server, which has a HTML-based development environment that enables you to create Node.js applications on a desktop computer, then run them on the BeagleBoard.

Thanks to BoneScript (a Node.js library), they can interact with the GPIOs on the BeagleBoard. You can then assign callbacks to things like button presses or getting data from an I2C connection.



```
var b = require('bonescript');

var ledPin = "P8_13";
var ledPin2 = "USR3";

b.pinMode(ledPin, b.OUTPUT);
b.pinMode(ledPin2, b.OUTPUT);

var state = b.LOW;
b.digitalWrite(ledPin, state);
b.digitalWrite(ledPin2, state);

setInterval(toggle, 1000);

function toggle() {
    if(state == b.LOW) state = b.HIGH;
    else state = b.LOW;
    b.digitalWrite(ledPin, state);
    b.digitalWrite(ledPin2, state);
}
```

The BeagleBone's HTML-based IDE makes it easy to get started with GPIO programming.

allow the JavaScript program to remain responsive while waiting for events to happen. We're going to use Node.js, which is a server-side JavaScript engine. This means it's used to run code on your server rather than in a web browser.

It's not usually installed by default, so you'll need to grab it from your package manager. On Debian-based systems, this is done with **sudo apt-get install nodejs**.

You can then run Node files with:

```
nodejs <filename>
```

We're not going to dive too deep into how to use Node.js because that's far beyond the scope of this article. We'll have a look at a couple of examples that both use the **http.get** method from the **http** module. This takes two parameters. The first is a series of options enclosed between **{** and **}**. The second is a callback function that is run once the request has been processed. The first example will just dump the HTML content to the terminal:

```
var http = require('http');
var print_html = function(res) {
 res.setEncoding('utf8');
 res.on('data', function (chunk) {
 console.log(chunk);
});
}
http.get({host:'www.bbc.com', path:'/'}, print_html);
```

Notice that **print_html** isn't a function; it's a variable that holds a function. The fact that it's a variable allows it to be passed to **http.get**, and called from there.

The function in **print_html** is called from **http.get** once data starts coming from the server. The single parameter is a HTTP response object, and this has various events on which you can set callbacks. In this case, we set a callback on the data event that is called each time a chunk of data is available. This time, we haven't stored the function in a variable, but declared it in the parameters to the method **res.on**.

This example shows how callbacks work, but it hasn't really shown the advantages of them. After all, you could quite easily program the above as a series of steps that happen one after the other. In fact, the code would probably be a little clearer if written in the order it executes.

Our second example will be a really simple website speed test. We'll send requests to four popular websites, and see which ones respond the fastest.

```
var http = require('http');
function wait_end(res, name) {
 console.log('starting ' + name);
 res.setEncoding('utf8');
 res.on('data', function (chunk) {});
 res.on('end', function() {
  console.log(name + ' finished');
 });
}
http.get({host:'www.bbc.com', path:'/'},
 function(res) {wait_end(res, 'bbc')});
http.get({host:'stackoverflow.com', path:'/'},
 function(res) {wait_end(res, 'stackoverflow')});
http.get({host:'www.ubuntu.com', path:'/'},
 function(res) {wait_end(res, 'ubuntu')});
http.get({host:'en.wikipedia.org', path:'/wiki/Main_Page'},
 function(res) {wait_end(res, 'wikipedia')});
```

Since **http.get** is non-blocking, the computer will execute the four **http.get** calls one after the other without waiting for the servers to respond. When the severs do respond, Node.js will run the callback, which is a function that runs **wait_end**. This time **wait_end** is the name of the function, not a variable, and it's called by an anonymous function (the callback). This is because of the extra parameter.

When the server first responds, it prints out a message saying that it's starting. Once all the data is received (and the end event on the response object happens), it prints out it's finished. Running this, you can see how the callbacks allow the computer to process multiple requests at the same time, because the start and end messages are interlinked.

This isn't a perfect speed test, as the initial HTTP requests won't be sent at the same time, but it should give you an idea of how the different servers perform (especially when there are only a few of them).

There is a real danger that using callbacks can leave your code hard to read and difficult to maintain, so they should be used with caution. However, when used well, they are a powerful tool to make your programs faster and more flexible.