Creating Scratch 2.0 Extensions

John Maloney

June 18, 2013

NOTE: This specification is preliminary and may change.

Introduction

Scratch 2.0 can be extended to control external devices (e.g. robotic kits, musical instruments) and to access data from external sensor hardware (e.g. sensor boards). A Scratch 2.0 extension extends Scratch with a collection of special command and reporter blocks that can be used to interact with a particular device. When the extension for that device is enabled, these new blocks appear in the "More Blocks" palette.

Due to browser security restrictions, Scratch 2.0 cannot interact with most hardware devices directly. Instead, most extensions come with a helper app, a separate application that the user must install and run on their computer. Scratch communicates with the helper app via TCP/IP socket, and the helper app talks to the hardware.

The Scratch 2.0 extension mechanism is still under development. Eventually, the user will be able to browse and import extensions from a library of published extensions, just as they currently import costumes, backdrops, sprites and sounds from the Scratch media library, but that mechanism is not yet implemented. Meanwhile, to allow extension development and testing, a semi-hidden menu command can be used to import an extension from a local file.

This document is aimed at Scratch 2.0 extension developers. It describes the extension description file format and the protocol used to communicate between Scratch extension helper apps. It also outlines the extension development process.

Extension Description File

An extension description file is a text file in JSON format (www.json.org) that describes the extension. By convention, a Scratch 2.0 extension file ends in .s2e.

The JSON object in the extension description file includes the extension's name, the TCP/IP port number used to connect to the extensions helper app, and a list of Scratch block descriptions.

Here is an example:

```
{  "extensionName": "Extension Example",
   "extensionPort": 12345,
   "blockSpecs": [
                [" ", "beep", "playBeep"],
                [" ", "set beep volume to %n", "setVolume", 5],
                ["r", "beep volume", "volume"],
   ]
}
```

The extension's name is "Extension Example" and it connects to its helper app on port 12345. The "blockSpecs" field describes the extension blocks that will appear in the "More Blocks" palette. In this case, there are three blocks: (1) a command block that plays a beep; (2) a command block that sets the beep volume; and (3) a reporter (value-returning block) that returns the beep volume setting.

Block Descriptions

Each block is described by an array with the following fields:

* block type
* block format
* operation or remote variable name
* (optional) zero or more default parameter values

The block type is one of these three strings:

" " - command block
"r" - number reporter block (round ends)
"b" - boolean reporter block (pointy ends)

The block format is a string that describes the labels and parameter slots that appear on the block. Parameter slots are indicated by a word starting with "%" and can be one of:

%n - number parameter (round ends)
%s - string parameter (square ends)
%b - boolean parameter (pointy ends)

In the example extension, the "set beep volume to %n" takes one numeric parameter. The block description for that command also includes a default value (5). This default value will be displayed in the parameter slot when that block appears in the palette. Default parameters allow users to easily test blocks and suggest the range of a given parameter. The default value of "5" in this example suggests that the range of the beep volume is 0 to 10.

The operation field is used in two ways. For command blocks, it is sent to the helper application along with any argument values to invoke the desired operation. For reporter blocks, it is the name of an extension state variable (e.g. the name of a sensor). Executing the reporter block returns the most recently reported value for that extension variable.

Helper Apps

Helper apps run in the background, ready for use by any Scratch project that uses that extension. Each extension has a unique port number. The helper app behaves like a server, waiting for Scratch to connect to its port. When an extension is imported or when a Scratch project using that extension is opened, Scratch attempts to connect to the helper app port on the local computer. If the connection attempt fails, Scratch will retry every five seconds. (Thus, the helper app can be started at any time and Scratch will find it.) Once the connection to the helper app is established, it persists until the Scratch project is closed. When its connection to Scratch is closed, the helper app should clean up and wait for the next connection.

What should a helper app should do if several instances of Scratch (e.g. running in separate browser windows) try to connect to it at the same time? That's up to the extension author to decide. In some cases (e.g. controlling a robot), it might make sense to allow only a single connection at a time, rejecting additional connections. In other cases, (e.g. a music synthesizer) it might be able to support multiple simultaneous Scratch clients.

Helper App Protocol

Once a connection is established, Scratch sends commands to the helper app and the helper app sends variable updates (e.g. sensor values) back to Scratch. Messages in both directions are sent as JSON objects separated by newline characters ('\n').

A message object has the following format:

```
 {"method": <operation>, "params": […]}
```

The method field holds the operation string from the block description and the params field is an array containing zero or more parameters for that command. Here is an example:

```
 {"method": "setVolume", "params": [7]}
```

A variable (sensor) update message looks like an "update" command sent by the helper app to Scratch, where the params field holds an array of name-value pairs. Here is an example:

```
 {"method": "update", "params": [["brightness", 75], "slider", 17]}
```

A helper app can send variable updates at any time. However, to make it easier to build helper apps, Scratch sends a "poll" command roughly 30 times a second:

```
 {"method": "poll", "params": []}
```

The polling mechanism makes it possible to build helper apps to with a simple, single-threaded structure. There's no need to use a separate thread or timer-based callback to send sensor updates; the helper app can simply wait until a command arrives (i.e. it can block on the call to read from the socket), then respond as appropriate for the given command. Even if no other commands are sent, the helper app can be sure of seeing frequent "poll" commands. Of course, a helper app can choose to simply ignore the poll messages.

Building and Testing Extensions

Here are the steps for creating and testing a Scratch extension:

1. Create an extension description file
2. Create your helper app and start it
3. Open the Scratch 2.0 editor
4. Import the extension description (shift-click on "File" and select "Import Experimental Extension" from the menu)
5. The new extension blocks will appear in the More Blocks palette
6. Test and iterate!

Eventually, the Scratch Team will assign socket numbers for extensions in the library. However, for testing an extension, any unused socket number over 1024 can be used.

Helper apps can be written in any language that supports server sockets, such as Python, Node.js, Java, C/C++, etc. Many of these languages include JSON parsers in their libraries. You can also check www.json.org for a list of third-party JSON libraries.

Distributing Extensions

As of this writing, the extension distribution strategy is still being worked out. One possible path is for the Scratch website to host a library of "published" extensions. Users would be able to browse and select extensions from this library and would be able to download the necessary helper app from the Scratch website. Published extensions would be checked for quality and safety by the Scratch team. There might be other criteria for inclusion in the the Scratch library, such as the associated hardware being readily available and the policy is likely to evolve over time.

Meanwhile, you can share experimental extensions with users by sharing both the extension description file and a helper app with them and telling them how to use the testing interface to import the extension.

One request: For now, until we have a better understanding of how to present and manage non-standard extensions to other users on the website, please do not upload projects that use extension blocks to website; save them locally instead. Thanks!