Creating Scratch 2.0 Extensions

John Maloney
MIT Media Laboratory
September, 2013

NOTE 1: This specification supersedes all earlier drafts.
NOTE 2: This specification is still preliminary and may change.

Introduction

Scratch 2.0 can be extended to control external devices (e.g. robotics 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 command and reporter blocks that can be used to interact with a particular device. When an extension is enabled, its blocks appear in the "More Blocks" palette.

Due to browser security restrictions, Scratch 2.0 cannot interact with hardware devices directly. Instead, hardware 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 HTTP requests, and the helper app talks to the hardware. In the future, some extensions may package their helper apps as browser plugins, but that mechanism is not described here.

This document is aimed at Scratch 2.0 extension developers. It describes the extension description file format, the protocol used to communicate between Scratch extension helper apps, and 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 communicate with the extensions helper app, and a list of Scratch block specifications.

Here is an example:

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

This 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 reports the current beep volume setting.

Block description

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 strings:

" " - command block
"w" - command block that waits
"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, 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 operation field in a block specification is used in two ways. For command blocks, it is sent to the helper application, along with any parameter values, to invoke an operation. For reporter blocks, it is the name of a sensor variable. Sensor variable values are kept in a dictionary. Executing a reporter block simply returns the most recently reported value for that sensor variable.

Menu parameters

Both command and reporter blocks can include menu parameters:

%m.menuName - menu parameter (not editable)
%d.menuName - editable number parameter with menu

The first of these provides a simple drop-down menu parameter slot, similar to the parameter of the "broadcast" block. The second provides a numeric parameter slot with an auxiliary menu, similar to the "point in direction" block. In both of these cases, the string after the period character is the name of the menu. The contents of the menu are supplied by the optional "menu" field in the extension descriptor. For example, a Microsoft Kinect extension might look like this:

```
{  "extensionName": "Kinect",
  "extensionPort": 12345,
  "blockSpecs": [
        ["r", "get %m.coordinate position of %m.bodyPart", "position"],
  ],
  "menus": {
        "coordinate": ["x", "y", "z"],
        "bodyPart": ["head", "shoulder", "elbow", "hand"],
   },
}
```

In this example, the "position" reporter block has two menus. One menu specifies the coordinate

(x, y, or z) and the other specifies the body part (head, shoulder, elbow, or hand).

The sensor name reported by the helper app for a reporter block with menu parameters is simply the concatenation of the reporter's base name with all its parameters separated by forward slash characters. In the Kinect example, the y position of the user's hand might be reported as:

    position/y/hand 247

This example has been greatly simplified. The Kinect actually tracks many more body parts, including both the left and right sides of the body, and up to four human figures. Menu parameters allow the number of blocks needed for a Kinect extension to be collapsed from over a hundred separate reporter blocks to, potentially, just a single block with four menu parameters.

Communicating with the helper app

A helper app runs in the background, ready for use by Scratch projects that uses that extension. Each extension has a unique port number. Scratch looks for the helper app at the given port number on the local computer.

Scratch communicates with the helper app using the HTTP protocol. Scratch sends commands to the helper app and the helper app sends sensor values and status information back to Scratch via HTTP GET requests. Since the protocol is standard HTTP, any browser can be used to test and debug helper apps.

Polling

Scratch to retrieves sensor values and status information from the helper app by sending a poll command:

    /poll

In response to a poll command, the helper app sends back a list of (sensor name, value) pairs, one pair per line. Each line in the poll response should end with a newline character (0xA) and the sensor name and value should be separated by a space character. String values should be URL-encoded. Scratch sends poll commands roughly 30 times per second.

Here's an example poll response:

    brightness 75
    slider 17

Reporting problems

A common problem with hardware extensions is that the required hardware is not attached to the computer. The response to a poll request can include a problem report to help users troubleshoot. A problem report consists of the string "_problem" followed by a space followed by a short description of the problem. For example:

    _problem The Scratch Sensor board is not connected.

Reserved sensor names

Scratch uses special sensor names to report extension problems, track busy commands, etc. These special names always start with an underscore character ("_"). To avoid possible name conflicts with potential future feature, extensions writers should avoid starting either command or sensor names with an underscore.

Commands

A command and its parameters are formatted as a URL request, using the forward slash ("/") as a separator. Some examples:

    /beep   (command with no parameters)
    /setVolume/5   (command with a numeric parameter)

String parameters must be URL-encoded so that the request is a well-formed URL.

Commands that wait

Some commands wait until some action completes. For example, the block:

    turn motor on for 3 seconds

turns on the motor, waits three seconds, then turns it off again. When this block is used in a script, execution does not continue to the next block until the command completes. A command that waits is indicated by the "w" block type in the command descriptor.

When a "w" command is invoked, Scratch adds a unique *command_id* parameter to the request (before any other parameters). For example, for the motor command above Scratch would send:

    /motorOn/2437/3

The first parameter, 2437, is a unique identifier for this invocation of the command. For the three seconds that this command takes to complete, the helper app adds a *busy line* to the poll request:

_busy 2437 …

A busy line consists of the string "_busy" followed by a list of unique identifiers separated by spaces. When Scratch gets a poll result that doesn't include 2437 in the busy line (or doesn't even have a busy line), it knows that the command is complete and allows the script that invoked that command to proceed.

Reporters that wait (future feature)

Note: This feature is not yet implemented because the Scratch interpreter does not yet support reporter blocks that can suspend their thread until the result is ready.

In the future, the extension mechanism may be extended to support extension reporter blocks that request data from the helper app and and wait until the result is returned before proceeding. To handle such requests (sometime called a "blocking" request), Scratch will add a unique *request_id* parameter to the request (before any other parameters). For example, a weather extension might include a block to get the current temperature in a given city:

temperature in *city_name*

To get the temperature in Boston, this command would send the helper app a request like this:

/getTemperature/7639/Boston

The first parameter, 7639, is a unique request ID string. In response, the helper app would initiate a request to get the weather information for Boston. Some time later, when the result of that request is returned from the server, the helper app would include the following line in its response to the next poll request:

_result 7639 82

This line consists of the special string "_result", the request ID, and the result value separated by space characters. As with other sensor values, result value should be URL-encoded.

Reset command

Scratch extensions can control motors or music synthesizers. Users expect to be able to stop everything -- turn off motors, silence music synthesizers, and reset hardware back to it's original state -- by clicking the stop button in the Scratch editor. Thus, when the stop button is clicked, Scratch sends a reset command to all active extensions:

/reset_all

In response to this command, the helper app should turn off motors, lights, sounds, etc. and reset the hardware to its power-up state. For example, if the Scratch program had set the volume of music synthesizer to zero, it should be reset to it's default level. The helper app should also cancel any ongoing processes and clear its list of busy commands.

Cross-Domain policy requests

There's one other requirement for the helper app: it must respond to Flash's request for a cross-domain policy file. This gives Flash permission to send HTTP requests to the helper app.

Flash sends the following request to the app:

```
/crossdomain.xml
```

The helper app must respond by sending a null-terminated policy file like this:

```
<cross-domain-policy>
  <allow-access-from domain="*" to-ports="<yourPort>"/>
</cross-domain-policy>
```

where <yourPort> is the port number of the helper app. Be sure to include a zero byte after the policy file or Flash won't recognize it. Unfortunately, Flash doesn't allow a wildcard (asterisk) in the "to-ports" field, so you have to fill that in with the correct port number. The example helper app code shows how to handle a policy request.

Once Flash has received and checked the policy request, it can communicate with the helper app.

Building and Testing Extensions

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.

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 Offline 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 your extension and iterate!

Helper apps can be written in any language that supports server sockets, such as Python, Node.js, Java, C, etc.

Eventually, the Scratch Team may assign socket numbers for popular extensions to avoid possible conflicts. For now, any unused socket number over 1024 can be used.

Distributing Extensions

As of this writing, an extension distribution strategy is still being worked out. The Scratch team will probably host a small library of "supported" extensions. Users will be able to browse and select extensions from this library from within the Scratch editor, and will be able to download the necessary helper app from the Scratch website. Supported extensions would be checked for quality and safety by the Scratch team. There are likely be strict criteria for including an extension in the Scratch-team supported extensions library, such as command set clarity and ease of use, size of the potential audience, wide-spread availability of any associated hardware, and a long-term commitment to support the extension.

To avoid potential confusion on the Scratch website, projects with unsupported extensions should only be used with the Scratch Offline Editor and should be saved only locally, not uploaded to the website. Extension developers can share their extensions by distributing copies of the extension description file and helper app. Users would then use the "Import Experimental Extension" command in the Offline editor to import the extension.

Protocol Summary

Scratch sends the following commands to a helper app:

```
/poll
/reset_all
```

The response to a /poll request is one or more lines separated by newline characters (0xA). The response may include one or more of the following:

- a sensor name and value separated by whitespace
- "_busy" followed by a list of unique ID's for commands in progress
- "_problem" followed by a problem description
- "_result" followed by a unique ID and value (not yet implemented)