Creating mBlock Extensions

by Wang Yu, Product Manager at Makeblock

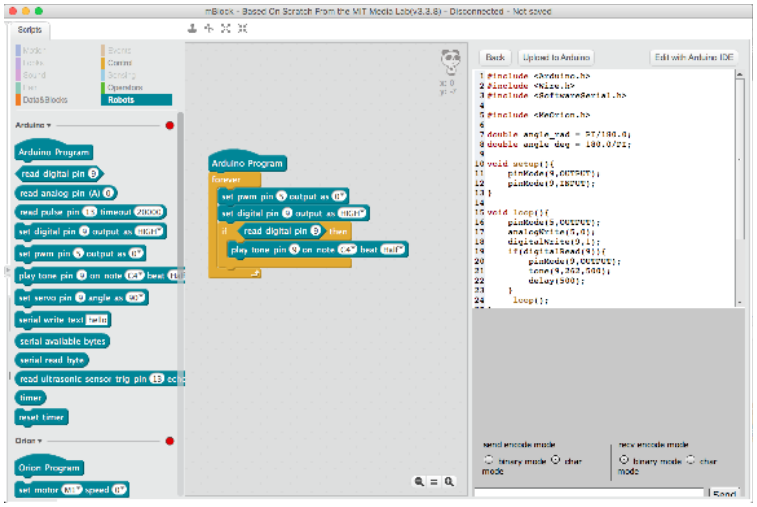
How Extensions Work

Extensions allow custom blocks for mBlock. You can use extensions to support third-party Arduino sensors or other robotic products such as Lego or LittleBits.

Anybody can write extensions for mBlock. This makes mBlock an awesome platform for every types of hardware-related programming.

Scratch Mode and Arduino Mode

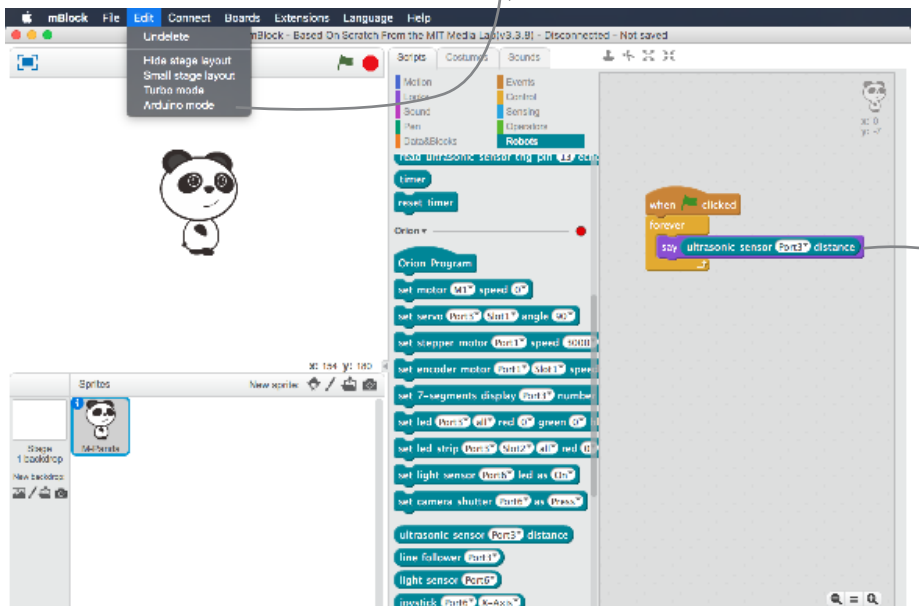
Every block in mBlock has two modes: **Scratch Mode** and **Arduino Mode**. It's important to know their difference before writing extensions.



▲ **Arduino Mode**

In Arduino Mode, the program is **uploaded into the robot** and the robot is **run on its own**. However, you **cannot** use graphics from Scratch since the computer is no longer there.

use Edit/Arduino Mode menu item to toggle between Scratch and Arduino modes



▸ **Scratch Mode**

In Scratch Mode, the robot or Arduino board **must be connect to the computer** in order to run the program. You **can** use Scratch blocks to create graphics or make games.

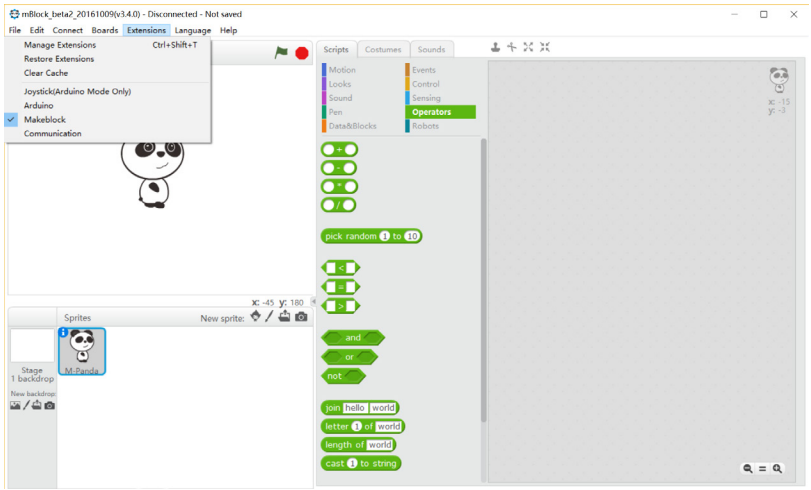
this **"Say"** block can only be used in the Scratch mode; while the "**(Repeat) Forever**" block can be accessed in both modes.

Using an Extension

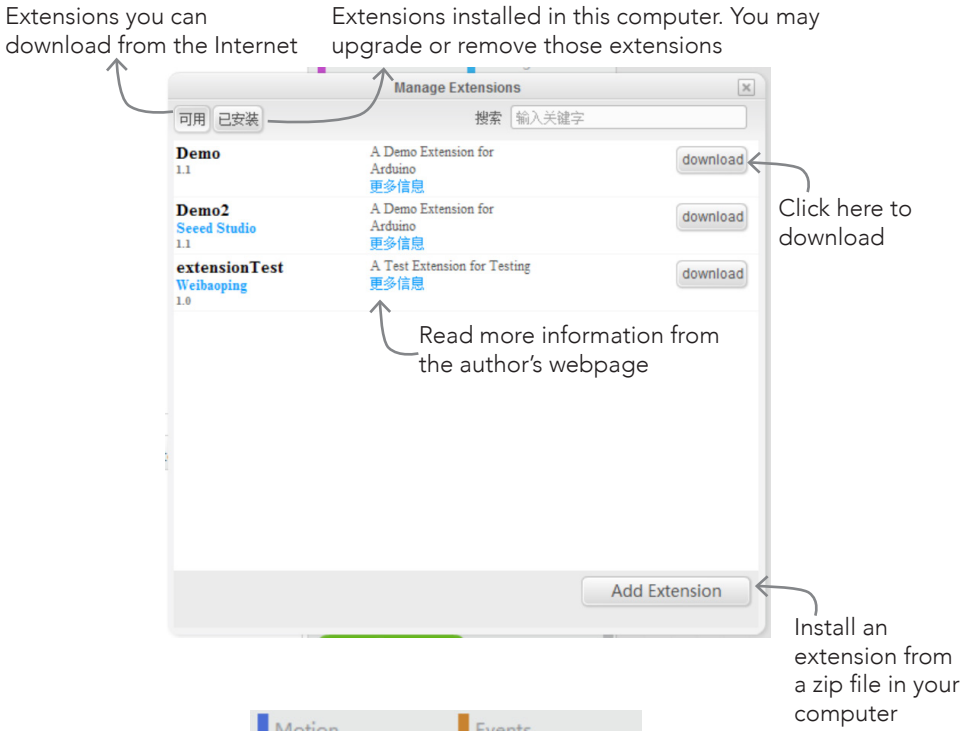
Using the Extension Manager, you can browse extensions, find one you want, and get the blocks included in the extension with a single click.

The following instructions shows how to add an extension to mBlock.

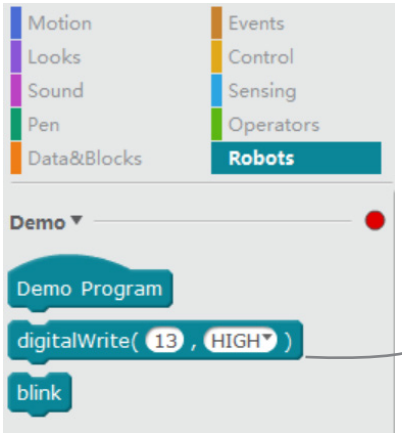
1 Use the menu item “Extensions”, “Manage Extensions” to open up the Extension Manager.



2 You may search for extensions with the search box. Click “Download” at the right of the table item to download the extension. **Internet connection is required.**



3 The extension you downloaded will show up in the “Robots” group of mBlock.



Writing an Extension

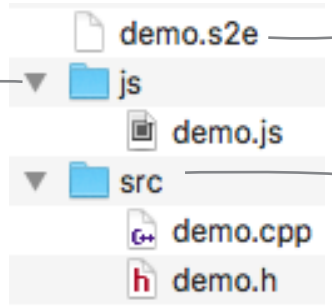
Writing an extension is not as hard as it seems. Mostly it is done by editing some text files. **Arduino** knowledge is required in implementing the Arduino Mode, while **Javascript** is used in the Scratch Mode. You may skip one of the modes, and the block will simply not work in that mode.

Here's a list of things extension writers need to do:

- Write basic information, such as the name and author
- Define how the blocks look like
- Tell mBlock how to generate the Arduino code
- Write the functions run in the Scratch mode
- Include additional Arduino header / C files.

The File Structure of an Extension

Every extension is a .zip file compressed from a folder. Here is the folder of the "Demo" extension when unzipped:



.s2e file: basic description, blocks definition of the extension, and the Arduino code generated by the extension in the Arduino Mode. "s2e" stands for "Scratch 2 extension" files

Javascript code that run on the Scratch Mode stays in the "js" folder

"src" folder is for additional files included in the Arduino mode. You may skip this if you don't use additional libraries

Tips

Source code of existing extensions are always the best resource in writing extensions. In mBlock, you can always view the source code of existing extensions through the Extension Manager.

Click here to view the source code of any existing extension.

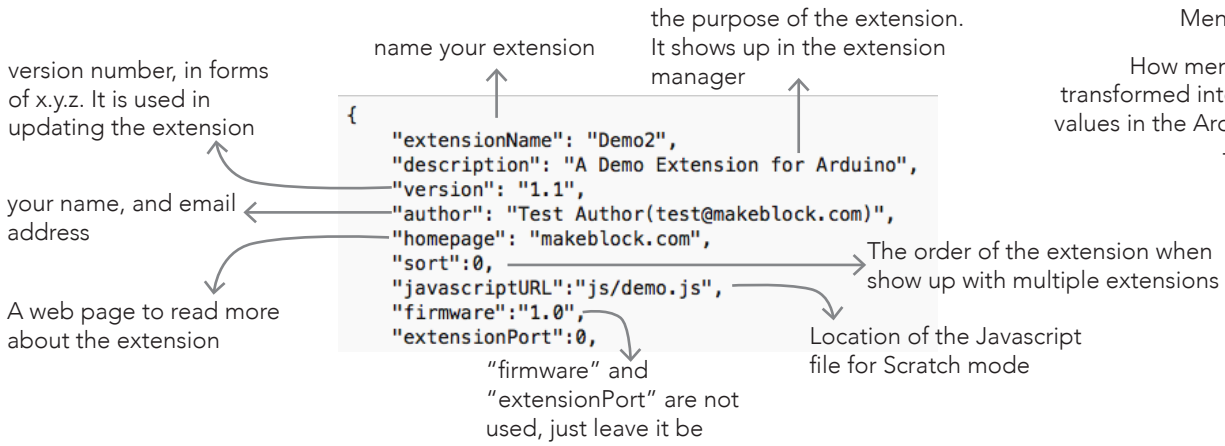
"View Source" will bring up a window of that extension's folder. You may try to change the extension code and it will take effect in the next time mBlock is launched (this is one way of debugging). However, **any changes made here will be discarded when upgrading mBlock or running the "clear cache" command**.

1 Filling out Basic Information

Download the "Demo" extension as a starting point at:
http://www.mblock.cc/site-images/Demo.zip

The next step is editing the s2e file. You need a text editor. the Notepad program is fine, but I recommend editors dedicated for code editing, such as Github Atom, Visual Studio Code, Brackets, or Sublime text.

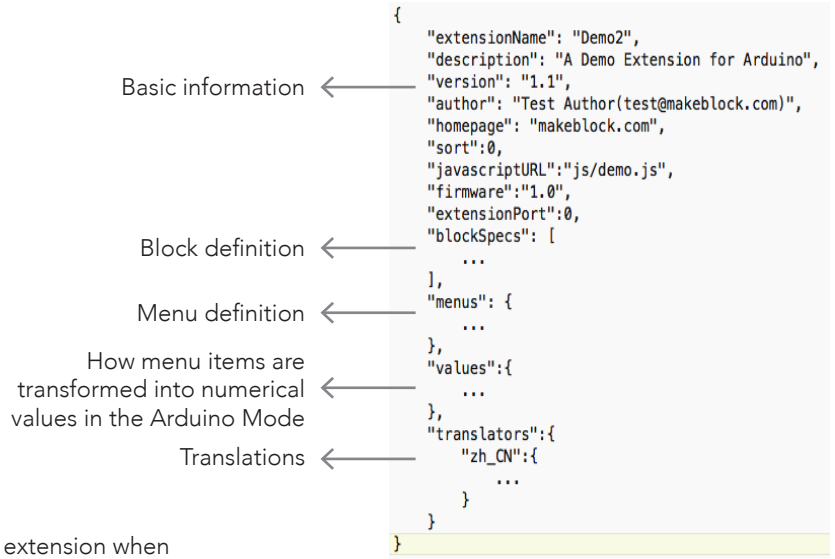
At the beginning of the .s2e file lies some basic information. You need to tell the user what does the extension do and who wrote it. Here is the starting lines of demo.s2e:



Tips

the .s2e file is the main file of the extension. Aside from the basic information, it defines blocks, tells mBlock how the dropdown menus look like, and how to translate the extension to another language (if you want to).

If you are familiar with Javascript, you may notice it is written in a plain JSON (JavaScript Object Notation) object.



2 Defining Blocks

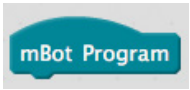
the "blockSpecs" section tells mBlock how blocks look like, and gives clues on how they behave in Scratch and Arduino mode.

```
"blockSpecs": [
    ["h","Demo Program","runArduino"],
    [
        "w",
        "digitalWrite( %n , %d.digital )",
        "digitalWrite",
        "13",
        "HIGH",
        {
            "setup":"pinMode({0},OUTPUT); \n",
            "inc":"",
            "def":"",
            "work":"digitalWrite({0},{1});\n",
            "loop":""
        }
    ],
    [
        "w",
        "blink",
        "blink",
        {
            "setup":"",
            "inc":"#include \"demo.h\"",
            "def":"DemoClass demo; \n",
            "work":"demo.blink(); \n",
            "loop":""
        }
    ]
],
```

Every block is described in a Javascript Array form

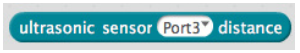
▾ **Block Types are indicated by a single letter:**

"h" stands for **"header blocks"**; they are rarely used in custom extensions.



"w" stands for **"write blocks"**; they send commands to the hardware and do not expect a response.

"r" is for **"reading blocks"**; in Scratch Mode, it waits the function to return a value; **"R" is for asynchronous reading** - the value is not returned by the function but told later ("callback") through a function in Javascript. "r" and "R" looks the same and have no difference in the Arduino mode.



"b" is for **"binary blocks"**; they return a binary yes-or-no value. Similarly, **"B" is for asynchronous reading of binary values**

▾ **The text that appears on the blocks**

words begin with "%" are parameters - "slots" that users can fill up by typing or with other blocks. This block ("digitalWrite(%n , %d.digital)") looks like:

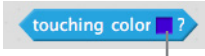
%n gives a round slot for numbers, and will give a number in the Arduino Mode.

%d.name gives a round dropdown box, and will give a number in the Arduino Mode. The dropdown's content is defined in the "menus" section and identified by **name**.

Other parameter types include:

%s gives a rectangular hole for strings

%m.name gives a rectangular dropdown box for strings



%c gives a color picker

3 Making Menus

the "digitalWrite" block has a convenient menu for selecting HIGH or LOW output. It is defined in the following way:


```
"blockSpecs": [
    ["h","Demo Program","runArduino"],
    [
        "w",
        "digitalWrite( %n , %d.digital )",
        "digitalWrite",
        "13",
        "HIGH",
        {
            "setup":"pinMode({0},OUTPUT); \n",
            "inc":"",
            "def":"",
            "work":"digitalWrite({0},{1});\n",
            "loop":""
        }
    ],
    [
        "w",
        "blink",
        "blink",
        {
            "setup":"",
            "inc":"#include \"demo.h\"",
            "def":"DemoClass demo; \n",
            "work":"demo.blink(); \n",
            "loop":""
        }
    ]
],
"menus": {
    "digital":["HIGH","LOW"]
},
"values":{
    "HIGH":1,
    "LOW":0
},
```

"%d.digital" means menu information is stored in the "digital" list of the menus section

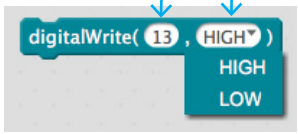
this means the menu has two options: **"HIGH" and "LOW"**

in **Arduino mode**, **"HIGH"** generates a value of 1 and **"LOW"** generates 0. This does not affect the Scratch Mode.

4 Finishing Up the Block Definition

give a **Javascript function name** to use in the **Scratch Mode**

```
"blockSpecs": [
    ["h","Demo Program","runArduino"],
    [
        "w",
        "digitalWrite( %n , %d.digital )",
        "digitalWrite",
        "13",
        "HIGH",
        {
            "setup":"pinMode({0},OUTPUT); \r
            "inc":"",
            "def":"",
            "work":"digitalWrite({0},{1});\r
            "loop":""
        }
    ],
```



give **default values** to the block you just defined

5 Arduino Code Generation

When defining each block, .s2e file also decides how Arduino code is generated, following rules below:


```
[
    "w",
    "set servo pin %n angle as %d.servovalue",
    "runServoArduino",
    "9",
    90,
    {
        "inc":"#include <Servo.h>\n",
        "def":"Servo servo_{0};\n",
        "setup":"servo_{0}.attach({0});\n",
        "work":"servo_{0}.write({1});\n",
        "loop":""
    }
],
```

insert **include statements**

define **variables**

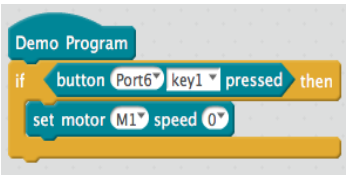
insert **once** in the setup() function

can appear **many times** when the block is used

{0} means the first parameter, "9" in this case

Back | Upload to Arduino | Edit with Arduino IDE

```
#include <Arduino.h>
#include <Wire.h>
#include <SoftwareSerial.h>

#include <Servo.h>

double angle_rad = PI/180.0;
double angle_deg = 180.0/PI;
Servo servo_9;

void setup(){
    servo_9.attach(9);
    for(int __i__=0;__i__<10;++__i__)
    {
        servo_9.write(90);
    }
}

void loop(){
    _loop();
}

void _delay(float seconds){
    long endTime = millis() + seconds * 1000;
    while(millis() < endTime)_loop();
}

void _loop(){
}
```

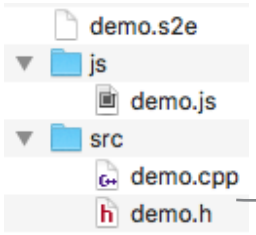



```
[
    "B",
    "button %d.blackPorts %m.button_keypressed",
    "getButton",
    "Port6",
    "key1",
    {
        "setup":"",
        "inc":"",
        "def":"Me4Button buttonSensor_{0}({0});\n",
        "work":"(buttonSensor_{0}.pressed()=={1})",
        "loop":"buttonSensor_{0}.pressed();"
    }
]
```

insert **once** in the loop() function

Back | Upload to Arduino | Edit with Arduino IDE

```
#include <Arduino.h>
#include <Wire.h>
#include <SoftwareSerial.h>

double angle_rad = PI/180.0;
double angle_deg = 180.0/PI;
Me4Button buttonSensor_6(6);
MeDCMotor motor_9(9);

void setup(){
    if((buttonSensor_6.pressed()==1)){
        motor_9.run(0);
    }
}

void loop(){
    _loop();
}

void _delay(float seconds){
    long endTime = millis() + seconds * 1000;
    while(millis() < endTime)_loop();
}

void _loop(){
    buttonSensor_6.pressed();
}
```



if you included other source files, be sure to copy them to the **"src"** folder

6 Coding Scratch Mode

The Javascript file is pretty long. I recommend take the demo.js and only make necessary changes. If your extension does not support Scratch Mode, skip this part.

demo.s2e

```
{
    "extensionName": "Demo2",
    "description": "A Demo Extension for Arduino",
    "version": "1.1",
    "author": "Test Author(test@makeblock.com)",
    "homepage": "makeblock.com",
    "sort":0,
    "javascriptURL":"js/demo.js",
```

file name must match

demo.js

```
// demo.js

(function(ext) {
    var device = null;

    var levels = {
        HIGH:1,
        LOW:0
    };

    ext.resetAll = function(){};

    ext.runArduino = function(){};

    ext.digitalWrite = function(pin,level) {
        device.send([pin, levels[level]])
    };

    ext.blink = function(){
        device.send([0x22, 0x23])
    }

    function processData(bytes) {
        trace(bytes);
    }

    // Extension API interactions
    var potentialDevices = [];
```

don't change

define your own variables here

setup code for the extension

write code for your blocks here

send bytes via the serial ports using an array of bytes

this is called every time the computer receives bytes from serial ports

trace(string) function is really useful for debugging because it produces logs in the Arduino Mode panel.

demo.s2e

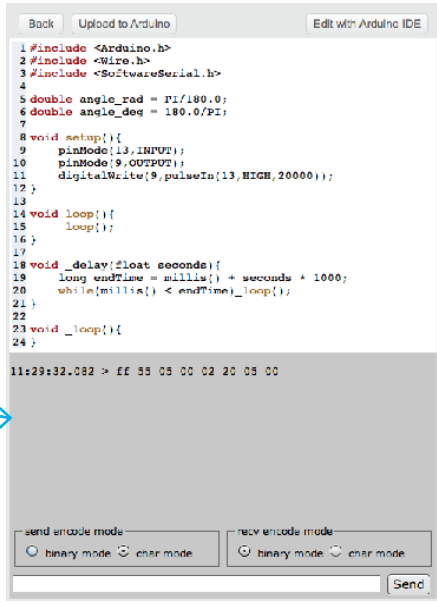
```
"blockSpecs": [
    ["h","Demo Program","runArduino"],
    [
        "w",
        "digitalWrite( %n , %d.digital )",
        "digitalWrite",
        "13",
        "HIGH",
        {
            "setup":"pinMode({0},OUTPUT); \n",
            "inc":"",
            "def":"",
            "work":"digitalWrite({0},{1});\n",
            "loop":""
        }
    ],
```

function name defined here

(at the end of) demo.js

```
    ext._getStatus = function() {
        if(!device) return {status: 1, msg: 'demo disconnected'};
        return {status: 2, msg: 'demo connected'};
    }

    var descriptor = {};
    ScratchExtensions.register('demo', descriptor, ext, {type: 'serial'});
})({});
```

change this to the name of your extension.



Publishing Your Extension

After you are satisfied with the extension, zip it into a .zip file. In MacOS, right click the folder and choose "**Compress xxx...**"; in Windows, right click the folder and choose "**Send To**", "**Compressed (zipped) folder**".

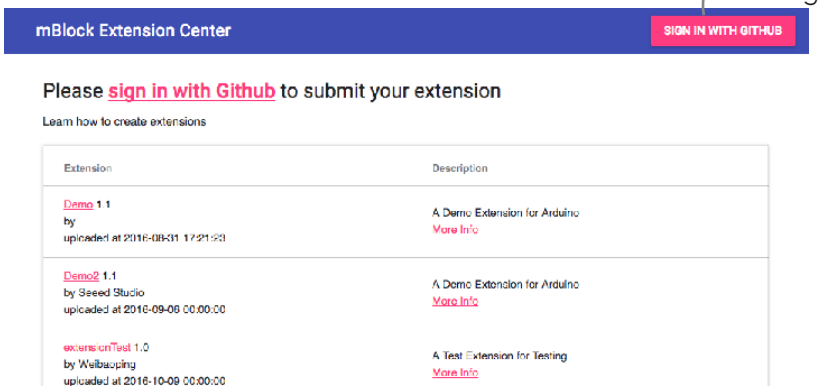
Then you may import the extension through the "Add Extension" button in the Extension Manager. But It'll be easier for the users if the extension is uploaded to the Online Extension Center.

1 Sign in with your Github account

Go to the extension center website:
http://www.mblock.cc/extensions/

And click "Sign-in with Github". If you don't have Github account, you need to register one.

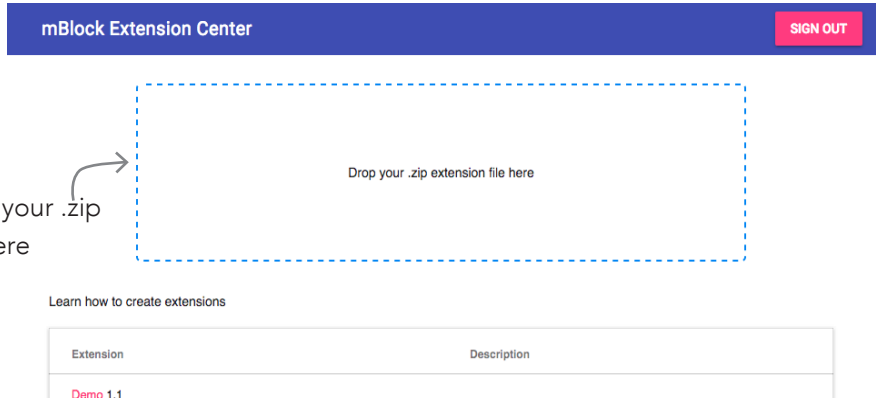
Click here to sign-in



2 Upload your extension

After Signing in, drag your .zip file to the large box (or click the large box to choose with a file browser)

Drag your .zip file here



Congratulations! You've uploaded an extension and contributed to the world mBlock community!

Tips

If you want to update your extension, simply upload a .zip file with a higher version number. And you're all set.

Tips

Makeblock, the company maintaining mBlock, reserves every right to remove an extension, with or without explanation. But we do welcome extensions of every kind from every product and hope we only moderate spam (like one with the name "test123") and those with improper content. Simply use your common sense.