## D.5 Eine Klasse für serielle Geräte schreiben

Man kann viel lernen, wenn man in einer interaktiven Umgebung mit einer neuen Bibliothek herumspielt. Allerdings müssen Sie irgendwann auch richtigen JavaScript-Code schreiben, den Sie in einem Projekt verwenden können.

Da JavaScript objektorientierte Programmierung unterstützt, ist es sinnvoll, den gesamten Code rund um den Zugriff auf den seriellen Port in eine eigene Klasse zu packen. Dadurch können Sie ihn in anderen Projekten wiederverwenden und müssen Fehlerkorrekturen und Verbesserungen nur an einer einzigen Stelle vornehmen. Außerdem bestehen gute Aussichten, dass es in Kürze eine browserübergreifende Lösung für den Zugriff auf den seriellen Port geben wird.[8] Wenn das geschieht, können Sie einfach das Innenleben Ihrer Klasse durch die neue Standard-API ersetzen, woraufhin Ihre Anwendungen automatisch in allen Browsern funktionieren werden. Nun, zumindest gilt das für die Teile, die auf den seriellen Port zugreifen. Bei allen sonstigen Teilen können natürlich andere Browserinkompatibilitäten eine Rolle spielen.

In diesem Abschnitt erstellen wir die Klasse *SerialDevice*, die wir in vielen unserer Projekte nutzen können. Sie basiert auf den Standardbeispielen von Google Chrome Apps.[9] Schauen Sie sich auch die anderen Beispiele an. Das ist eine großartige Möglichkeit, etwas zu lernen.

Es ist nicht schwierig, in JavaScript Klassen zu erstellen, allerdings läuft dieser Vorgang ganz anderes ab als in den meisten anderen objektorientierten Sprachen. Das Objektsystem von Java basiert nicht auf Klassen, sondern auf Prototypen. Anstatt eine Vorlage (Klasse) zum Erstellen neuer Objekte bereitzustellen, legen Sie unmittelbar neue Objekte an. Anschließend verfeinern Sie diese Objekte und können sie dann als Vorlage (oder Elternobjekt) für andere Objekte verwenden.

Unabhängig von allen Unterschieden zwischen klassen- und prototypbasierten Sprachen brauchen Sie irgendeine Möglichkeit, um Objekte zu erstellen. In JavaScript verwenden Sie dazu eine Konstruktorfunktion:

ChromeApps/SerialDevice/js/serial_device.js

```
Zeile 1  var SerialDevice = function(path, baudRate) {
     -     this.path = path;
     -     this.baudRate = baudRate || 38400;
     -     this.connectionId = -1;
     5     this.readBuffer = "";
     -     this.boundOnReceive = this.onReceive.bind(this);
     -     this.boundOnReceiveError = this.onReceiveError.bind(this);
     -     this.onConnect = new chrome.Event();
     -     this.onReadLine = new chrome.Event();
    10     this.onError = new chrome.Event();
     -   };
```

Mithilfe dieser Funktion können Sie neue *SerialDevice*-Objekte wie folgt erstellen:

8 *http://whatwg.github.io/serial*
9 *https://github.com/GoogleChrome/chrome-app-samples/tree/master/samples/serial/ledtoggle*

```
var arduino = new SerialDevice("/dev/tty.usbmodem24321");
```

Beachten Sie die häufige Verwendung des Schlüsselworts *this*. In JavaScript verweist *this* auf den Ausführungskontext der aktuellen Funktion. Sie können dieses Schlüsselwort für verschiedene Zwecke nutzen. Beim Anlegen von Objekten dient es hauptsächlich dazu, Attribute und Methoden zu erstellen, die an das Objekt gebunden sind. In den Zeilen 2 bis 5 definieren wir damit Instanzvariablen wie z. B. *path*.

Auch in den darauffolgenden zwei Zeilen definieren wir Instanzvariablen. Hier verwenden wir *this* auch auf der rechten Seite der Zuweisung und übergeben es der Methode *bind*. *bind* erstellt eine neue Funktion und setzt deren *this*-Schlüsselwort auf den Wert, den Sie *bind* ursprünglich übergeben haben. Mit *bind* können Sie eine Funktion schon zu diesem Zeitpunkt definieren, allerdings müssen Sie darauf achten, dass sie einen bestimmten Kontext hat, wenn Sie sie schließlich aufrufen.

Diese Vorgehensweise ist bei der Arbeit mit Ereignishandlern oft notwendig. Die Benutzer der Klasse *SerialDevice* müssen in der Lage sein, ihre eigenen Callback-Funktionen zu übergeben, die aber im Kontext der Klasse ausgeführt werden sollen. Wie das funktioniert, werden Sie in Kürze erfahren.

Am Ende des Konstruktors definieren wir drei Instanzvariablen, bei denen es sich um Instanzen der Klasse *chrome.Event*[10] handelt. Diese Klasse bietet einige praktische Merkmale, um Ereignisse in Chrome-Apps zu definieren und zu senden. Wir verwenden sie hier, um die drei Ereignisse zu definieren, auf die die Benutzer der Klasse *SerialDevice* lauschen können. Beispielsweise können sich die Benutzer mithilfe der Eigenschaft *onReadLine* für *readLine*-Ereignisse registrieren.

Die folgenden drei Methoden der Klasse *SerialDevice* implementieren alles, was erforderlich ist, um Verbindungen mit seriellen Geräten herzustellen und zu trennen:

ChromeApps/SerialDevice/js/serial_device.js

```
SerialDevice.prototype.connect = function() {
  chrome.serial.connect(
    this.path,
    { bitrate: this.baudRate },
    this.onConnectComplete.bind(this))
};

SerialDevice.prototype.onConnectComplete = function(connectionInfo) {
  if (!connectionInfo) {
    console.log("Could not connect to serial device.");
    return;
  }
  this.connectionId = connectionInfo.connectionId;
  chrome.serial.onReceive.addListener(this.boundOnReceive);
  chrome.serial.onReceiveError.addListener(this.boundOnReceiveError);
  this.onConnect.dispatch();
};
```

10 *https://developer.chrome.com/extensions/events*

```
SerialDevice.prototype.disconnect = function() {
  if (this.connectionId < 0) {
    throw "No serial device connected.";
  }
  chrome.serial.disconnect(this.connectionId, function() {});
};
```

Als Erstes fällt auf, dass wir alle Methoden in der Eigenschaft *prototype* des *SerialDevice*-Objekts erstellen. Ich möchte hier nicht in die Details gehen. Merken Sie sich nur, dass dies eine Möglichkeit ist, um Objekten in JavaScript Methoden hinzuzufügen.

Die Methode *connect* delegiert ihre Aufgaben an die Funktion *chrome.serial.connect*, die Sie bereits im vorhergehenden Abschnitt kennengelernt haben. Das einzige Beachtenswerte ist hier die Callback-Funktion, die wir im Funktionsaufruf übergeben. Abermals verwenden wir *bind*, um den Kontext der Ausführungsfunktion ausdrücklich festzulegen. Dadurch stellen wir sicher, dass *onConnectComplete* Zugriff auf die Eigenschaften des *SerialDevice*-Objekts hat.

Das nutzen wir in der Methode *onConnectComplete* aus. Dort können wir die Eigenschaft *connectionId* unseres *SerialDevice*-Objekts festlegen, sobald wir erfolgreich Verbindung zu einem seriellen Gerät aufgenommen haben. Hätten wir *onConnectComplete* nicht zuvor gebunden, hätte *this* in dieser Funktion eine ganz andere Bedeutung. Dann könnten wir nicht auf die Eigenschaften des *SerialDevice*-Objekts zugreifen.

In Zeile 14 und 15 fügen wir dem *chrome.serial*-Objekt mit der gleichen Technik Empfangs- und Fehlerlistener hinzu. Hier verwenden wir die Listener, die wir in der Konstruktorfunktion vorbereitet haben. Nachdem die Verbindung hergestellt ist, rufen wir die Methode *dispatch* des *onConnect*-Objekts auf, um allen Listenern die Erfolgsmeldung mitzuteilen.

Schließlich müssen wir noch die eigentlichen Listenerfunktionen für ein- und ausgehende Daten und für Fehler implementieren:

ChromeApps/SerialDevice/js/serial_device.js

```
SerialDevice.prototype.onReceive = function(receiveInfo) {
  if (receiveInfo.connectionId !== this.connectionId) {
    return;
  }

  this.readBuffer += this.arrayBufferToString(receiveInfo.data);

  var n;
  while ((n = this.readBuffer.indexOf('\n')) >= 0) {
    var line = this.readBuffer.substr(0, n + 1);
    this.onReadLine.dispatch(line);
    this.readBuffer = this.readBuffer.substr(n + 1);
  }
};
```

```
SerialDevice.prototype.onReceiveError = function(errorInfo) {
  if (errorInfo.connectionId === this.connectionId) {
    this.onError.dispatch(errorInfo.error);
  }
};

SerialDevice.prototype.send = function(data) {
  if (this.connectionId < 0) {
    throw "No serial device connected.";
  }
  chrome.serial.send(
    this.connectionId,
    this.stringToArrayBuffer(data),
    function() {});
};
```

*onReceive* funktioniert im Grunde genommen genauso wie der Beispiellistener, den wir in Abschnitt D.4, *Die Chrome-API für serielle Verbindungen*, geschrieben haben. Der einzige Unterschied besteht darin, dass die neue Version auf Zeilenumbruchzeichen achtet. Wenn sie eines findet, übergibt sie den aktuellen Lesepuffer an die Funktion, die auf *onReadLine*-Ereignisse lauscht. Beachten Sie, dass in einer einzigen Portion Daten mehrere Zeilen übertragen werden können. Außerdem prüft *onReceive*, ob die Daten vom richtigen seriellen Port eingegangen sind.

Auch die Methode *onReceiveError* vergewissert sich, dass sie die Fehlerinformationen über die richtige Verbindung erhält. Wenn ja, sendet sie das Ereignis an die Funktion, die auf *onError*-Ereignisse lauscht.

Für unsere Zwecke brauchen wir keine *send*-Methode, aber es kann nicht schaden, sie der Vollständigkeit halber hinzuzufügen. Dadurch erhalten Sie eine *SerialDevice*-Klasse, die Sie auch in anderen Projekten verwenden können.

Als Letztes brauchen wir noch unsere beiden Hilfsmethoden, um *ArrayBuffer*-Objekte in Strings umzuwandeln und umgekehrt:

**ChromeApps/SerialDevice/js/serial_device.js**

```
SerialDevice.prototype.arrayBufferToString = function(buf) {
  var bufView = new Uint8Array(buf);
  var encodedString = String.fromCharCode.apply(null, bufView);
  return decodeURIComponent(escape(encodedString));
};

SerialDevice.prototype.stringToArrayBuffer = function(str) {
  var encodedString = unescape(encodeURIComponent(str));
  var bytes = new Uint8Array(encodedString.length);
  for (var i = 0; i < encodedString.length; ++i) {
    bytes[i] = encodedString.charCodeAt(i);
  }
  return bytes.buffer;
};
```

Das war es auch schon! Wir haben jetzt eine allgemeine Klasse, um von einer Chrome-App aus mit seriellen Geräten zu kommunizieren. Wir wollen sie auch gleich in einer kleinen Beispielanwendung nutzen, nämlich in der einfachsten möglichen Form eines seriellen Monitors. Er liest ständig Daten vom seriellen Port und zeigt sie auf einer HTML-Seite an. Diese Seite sieht wie folgt aus:

ChromeApps/SerialDevice/main.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8"/>
    <title>Serial Device Demo</title>
  </head>
  <body>
    <div id="main">
      <p>The Arduino sends:</p>
      <p id="output"></p>
    </div>
    <script src="js/serial_device.js"></script>
    <script src="js/arduino.js"></script>
  </body>
</html>
```

Die HTML-Seite enthält ein Absatzelement mit dem *id*-Wert *output*. Dieses Element füllen wir mit den vom seriellen Port gelesenen Daten. Am Ende des Dokuments binden wir unsere neue JavaScript-Bibliothek für den Zugriff auf serielle Geräte und die Datei *arduino.js* ein:

ChromeApps/SerialDevice/js/arduino.js

```
var arduino = new SerialDevice('/dev/tty.usbmodem24311');

arduino.onConnect.addListener(function() {
  console.log('Connected to: ' + arduino.path);
});

arduino.onReadLine.addListener(function(line) {
  console.log('Read line: ' + line);
  document.getElementById('output').innerText = line;
});

arduino.connect();
```

Hier erstellen wir ein neues *SerialDevice*-Objekt namens *arduino* und fügen dann Listenerfunktionen für die Ereignisse *onConnect* und *onReadLine* hinzu. Beide schreiben eine Meldung in die Konsole. Der Listener *onReadLine* stellt die gelesene Zeile in das DOM (Document Object Model) des Browsers.

Achten Sie darauf, in der ersten Zeile von *arduino.js* den richtigen seriellen Port anzugeben. Schließen Sie dann Ihren Arduino an den Computer an und laden Sie einen Sketch hoch, der ständig Textzeilen am seriellen Port ausgibt, beispielsweise den aus Abschnitt 6.5, *Ihren eigenen Game-Controller bauen*. Wenn Sie dann die Chrome-App starten, erhalten Sie eine Ausgabe wie die folgende:

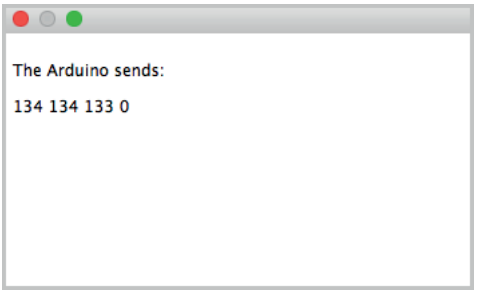


***Abb. D–4*** *Die Chrome-App kommuniziert mit dem Arduino.*

Sobald die Chrome-App eine neue Datenzeile empfängt, aktualisiert sie die HTML-Seite.

Ist es nicht faszinierend, wie leicht wir moderne Webtechnologien mit Microcontrollern kombinieren können?

Es gibt jedoch ein kleines Manko. Die Unterstützung der API für serielle Verbindungen in Google Chrome ist noch ziemlich neu. Es kann passieren, dass Ihr Browser hin und wieder abstürzt, vor allem dann, wenn Sie Chrome-Apps, die auf den seriellen Port zugreifen, starten oder anhalten oder wenn Sie Geräte trennen, während der Browser läuft. Abgesehen davon funktioniert jedoch alles reibungslos und stabil. Die Situation wird sich wahrscheinlich mit jedem neuen Browserrelease bessern.