Utilización de células P20 con Raspberry PI: Creación de una línea de braille

Resumen. Este artículo se centra en mostrar el resultado de una prueba de concepto donde creamos una tira de 4 células de bralle P20, utilizando una RaspberryPI. Con la ayuda de software diseñado por nosotros podemos controlar esta tira ayudando a mostrar el contenido de una página web en un formato legible para una persona con visión limitada o nula. Con este trabajo se demuestra la posibilidad de desarrollar alternativas de solución de bajo costo y código abierto, para la percepción de lectura de los contenidos web mediante salida Braille.

Palabras claves: accesibilidad web, Braille, placa electrónica, HTML, CSS, Javascript

1 Introducción

Actualmente tenemos muchas herramientas que permiten a las personas no videntes poder navegar por internet. Lamentablemente a pesar de contar con estas herramientas de adaptación o tecnologías de apoyo para poder utilizar esta plataforma, necesitan que los programadores realicen un trabajo extra para poder funcionar. Los sitios web de hoy en día poseen demasiado código fuente de programación que puede entorpecer el funcionamiento de las herramientas y a veces confundir a los usuarios, no hay que olvidarnos que las páginas web poseen demasiado código para los *bots* de búsqueda, interacción con el usuario, código CSS para el estilo de las mismas, codigo Java Script y demás [1]. Muchas veces estos fuentes dan un estilo único y particular a nuestros sitios web, pero cuando hablamos en términos de accesibilidad estos elementos pueden empantanar la usabilidad de una persona cuya visión es reducida o nula y que utilizan tecnologías de apoyo para su acceso [2].

Para solucionar este problema, los programadores suelen realizar prácticas incorrectas como hacer accesible solo la portada del sitio o generar 2 sitios web distintos, uno *full* funcionalidad, el cual está destinado a la mayoría de los usuarios del sitio, y otro con funcionalidad reducida, el cual apunta a un particular grupo de personas. Este *approach* está orientado a muchos desarrollos, pero la realidad es que por temas de tiempos o costos los desarrollos inclusivos son descartados y nunca llegan a ser implementados. Esto excluye a las personas con capacidades reducidas de muchos sitios web, y mucha funcionalidad en la web, el enfoque dado en este artículo es poder incluir a esas personas dentro de sitios que no fueron pensados ni diseñados para ellos.

La idea de este desarrollo nace como trabajo final de la materia Diseño de experiencia de usuario de la carrera Licenciatura en Informática de la Universidad Nacional de La Plata y tiene como objetivo poder mostrar una alternativa a herramientas existentes, adaptando la tecnología en post de la accesibilidad a personas no videntes.

1.1 Objetivo

El objetivo principal de este trabajo será el poder transformar código HTML o JS mostrado en una página web en información impresa por células de Braille físicas, esto podría incluir aún más a las personas con visibilidad reducida al uso de herramientas informáticas y como se planteó anteriormente, podríamos darles acceso a sitios web que no fueron pensados originalmente para ellos.
Con el propósito de lograr este objetivo, lo propuesto es realizar 2 (dos) desarrollos, para posteriormente unirlos. El primero de los desarrollos contempla un plugin Java Script en Firefox, el cual será el encargado de poder leer el sitio web y comunicarse con el segundo desarrollo [3]. La segunda parte de nuestro trabajo apunta a un código Java *deployado* sobre una plataforma Raspberry cuya función es manejar las células de Braille físicas [4]. Finalizado el desarrollo debemos ser capaces de leer un sitio web y traducido a Braille sin demasiado esfuerzo.

1.2 Sistemas similares

Actualmente existen muchas plataformas que permiten leer una página web y transformar la salida en audio como lo son los lectores de pantalla. El problema con estas plataformas es que si el sitio no está diseñado de manera particular, no es posible crear un mensaje de salida claro para el usuario, esto muchas veces termina complicado la experiencia del usuario y hasta muchas veces termina siendo imposible. No dejemos de pensar que los sitios web de hoy poseen mucha información en pantalla en un determinado momento: imágenes, banners publicitarios, links a otros sitios, código cuyo objetivo es minar datos o generar estadísticas para el sitio, entre otras cuestiones. Con estas limitaciones, si el sitio no es pensado de forma inclusiva, es muy probable que su acceso y utilización por una persona con capacidades reducidas sea complicado o hasta infructuoso [4].

Algunos ejemplos de estos lectores de pantalla, son:

JAWS[1]: Job Access With Speech es uno de los más populares lectores de pantalla que podemos encontrar para windows. Viene en muchos distintos productos, uno de los cuales es un lector para personas con visión disminuida, donde agranda la letra de la pantalla a un porción importante, el otro producto es un lecto propiamente dicho.

[1] https://www.freedomscientific.com/products/software/jaws/

NVDA[2]: Otro de los lectores más populares para Windows. Este es otro lector gratis de pantalla para personas con visión disminuida o nula.

ORCA[3]: Proyecto open source, esta vez aparece en la plataforma de Linux, ORCA es un lector de pantalla similar a los anteriormente mencionados.

VisionOver[4]: Es un lector de pantalla desarrollado por Apple, está disponible en casi todos los productos como ser Ipad, Iphone, Apple Watch y Ipod. Soporta distintos lenguajes de salida como se Ingles, Español, Chino, Japonés, etc.

ATK[5]: Herramienta open source desarrollado por Gnome para GTK+, al igual que VisionOver esta herramienta nos permite transformar en audio de salida lo que se está visualizando en pantalla.

Uno de los productos que más atrapó mi interés fue los propuestos por la misma compañía donde se consiguen los elementos de Braille. Esta empresa es llamada Metec, está localizada en Alemania y su sitio web es: https://www.metec-ag.de
Entre sus productos podemos encontrar líneas de Braille que constituyó las bases para encarar el trabajo expuesto en este artículo, con los mismos objetivos funcionales y sobre una mayor cantidad de células [5]. Pero el que más llamó mi atención es una pantalla de Braille que puede traducir cualquier gráfico 2D en Braille [6], tipo de componentes abre una puerta enorme para estudios a futuro donde poder realizar no sólo la conversión de texto, sino también de imágen.

2 Propuesta

Como mencionamos anteriormente el desarrollo de una solución va a contar con dos partes, donde la primera será un plugin Firefox que leerá el contenido de un sitio y lo enviará a través de HTTP a un servidor instalado en una placa Raspberry, en lugar de sockets preferimos instalar un servidor HTTP dado que esto facilita el desacoplamiento de la plataforma y además nos suma la posibilidad de integrar nuestro servidor Raspberry con distintas otras plataformas que puedan enviar HTTP request [7]. Éste además contará con un proceso que interactúa con los pin GPIO del Raspberry con el propósito de controlar las células de Braille conectadas a ellas, este sería el último eslabón de nuestra cadena.

En la figura 1 podemos ver cómo sería el *workflow* de convertir el contenido de un sitio web en Braille. En primer medida el plugin de Firefox lee el contenido del sitio

[2] https://nvda.es/
[3] https://help.gnome.org/users/orca/stable/introduction.html.es
[4] https://www.apple.com/accessibility/iphone/vision/
[5] https://developer.gnome.org/platform-overview/stable/tech-atk.html

web en HTML [7] y lo convierte en texto plano, este texto es partido en porciones de 4 caracteres dado que para la prueba de concepto sólo utilizamos 4 células P20, este código es enviado por medio de una llamada GET [8] a un servidor hecho en Kotlin que está instalado en una placa de multipropósito Raspberry PI, ese carácter es traducido a su correspondiente en Braille y enviado por medio de la librería PI4J a los pines de la Raspberry donde tenemos conectado el panel de células, al final del proceso, el panel lee la señal enviada y le dice a cada célula que muestre los caracteres enviados [9]:

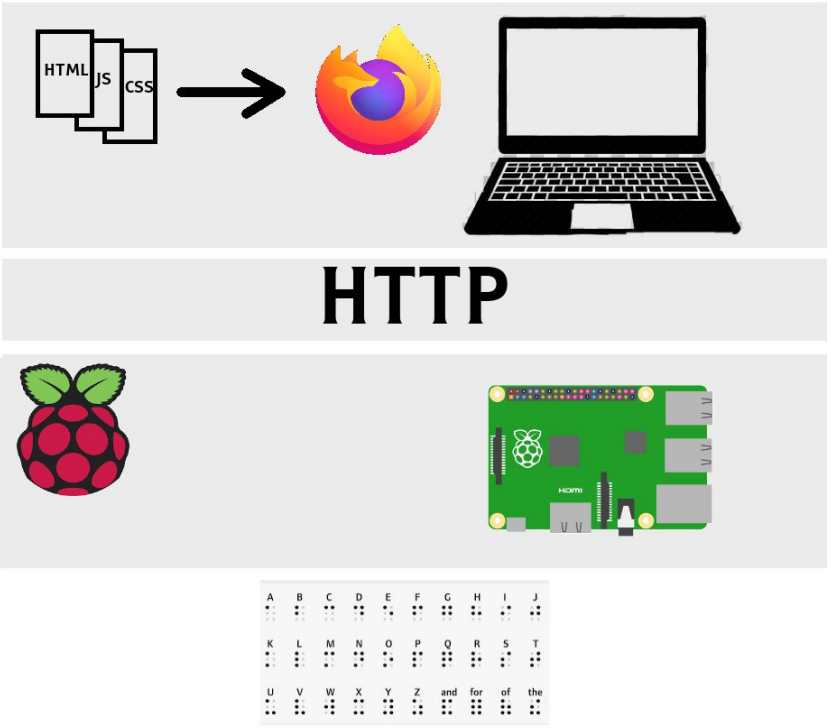


Fig. 1. Diseño de capas o layering utilizado en la prueba

A continuación listamos el *hardware* requerido y utilizado para llevar a cabo nuestra prueba de concepto:

P20 Metec[6]: Este producto es importado desde Alemania y tiene un precio dentro de todo alto, con lo cual si bien nuestro panel es de 4 células, nosotros solo vamos a utilizar 4.

[6] https://www.metec-ag.de/en/produkte-braille-module.php?p=p20

Panel Metec para 6 células P20: Este panel es importante, dado que es dock donde montamos las células, se pueden conseguir en la misma compañía donde se compran las P20. Como una aclaración las células poseen un sentido de conexión, **por favor no tratar de conectarlas "al revés"** dado que eso ocasiona un cortocircuito en la placa y nos arruinaría el slot.

Panel Metec transformador 5V. - 200V.: Este elemento es necesario para convertir el voltaje de salida del Raspberry a 200 Volts, el cual es el input necesario para las celular.

Raspberry 3B+[7]: Plaqueta Raspberry diseñada para correr un sistema operativo y poder controlar los pines de salida. Hasta el momento del diseño era una de las mejores plaquetas que se podían conseguir, el trabajo puede ser realizado con cualquiera, o incluso con un Arduino. Hoy en día existe una versión 4 que es mucho más potente, aunque como dije antes, es posible utilizar cualquier versión [10].

Sistema Operativo Raspbian[8]: Raspbian es el sistema operativo recomendado para una Raspi, en un principio trate de descargar un Ubuntu Mate adaptado (https://ubuntu-mate.org/ports/raspberry-pi/) pero lamentablemente este esta mas orientado al uso doméstico de una PC, con lo cual carecía de muchas librerías necesarias para el manejo de pines y el uso del hardware de la raspberry. Por eso instale Raspbian, el cual viene con todo instalado "ready to use".

Pi4J[9]: Librería que nos permite controlar los pines GPIO desde código Java, esto es completamente necesario para nosotros, dado que el servidor está desarrollado en Kotlin.

3 Desarrollo

Como se mencionó anteriormente el desarrollo cuenta con dos partes, una en Javascript y formato de plugin para Firefox, mientras que el otro es un servidor Kotlin corriendo sobre una RaspberryPI.

Se utiliza Kotlin dado que es un lenguaje potente, que compila a .class al igual que Java, desarrollado por el equipo de Jetbeans, es una alternativa interesante para desarrollar aplicaciones que interactúan a nivel de JVM con código Java, sin la necesidad de programar en el lenguaje de Java [9]. Kotlin está ganando mucha fuerza y me pareció interesante para esta prueba, dando grandes resultados no solo a nivel algorítmico sino en la velocidad de desarrollo y el poco *overflow* que agrega a un código similar desarrollado en Java.

[7] https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/
[8] https://www.raspberrypi.org/downloads/
[9] https://pi4j.com/1.2/usage.html

Para el *plugin* Firefox solamente desarrollamos unas pocas líneas de código, y nuestro enfoque apunta a tener una página limpia y preparada para poder utilizar con cualquier lector de código. Este desarrollo fue solo para realizar las pruebas iniciales del *plugin* y del dispositivo.

Para el plugin de Firefox lo único que necesitamos es crear una carpeta con los archivos javascript y un archivo JSON, el cual va a tener información de configuración en nuestra aplicación [3]. La figura 2 muestra cómo queda nuestra estructura de directorios.

Fig. 2. Estructura de archivos Java Script para el plugin de Firefox

El archivo manifest.json contiene toda la configuración necesaria para que Firefox reconozca nuestro *plugin*:

```
{
  "description": "Este plugin lee el contenido de una pagina web y lo envia a
una Raspberry que esta conectado a unas celulas de Braille P20",
  "manifest_version": 2,
  "name": "BraillePlugin",
  "version": "1.0",
  "icons": {
            "48": "icons/braille.png"
  },

  "content_scripts": [
            {
            "matches": ["*://*.com/**", "*://*.com.ar/**"],
            "js": ["brailleFunctions.js", "brailleplugin.js"]
            }
  ]
}
```

brailleplugin.js es la primera función que se ejecuta al cargar el plugin y solo posee una llamada:

```
/*
Control Braille P20 Cells.
*/
searchBody();
```

Por último, el archivo brailleFunctions.js posee toda la lógica, donde se corta el *body* del documento en partes y se envía de a 4 dígitos al servidor Raspberry, las funciones son simples y como dijimos antes, el enfoque adoptado en este documento es poder comunicar código Javascript con un servidor HTTP hecho en Kotlin para enviar señales a un conjunto de células Braille P20 [9]. Nuestro ejemplo es algo bastante sencillo que busca resolver el problema de mostrar en Braille físico la información en una pantalla, no busca generar un *plugin* robusto que remueva el código innecesario del DOM para solo mostrar el texto. Esta funcionalidad será incluida en las futuras versiones de este trabajo. El archivo brailleFunctions.js se ve de este modo:

```
/*
Control Braille P20 Cells.
*/

var firstChar = 0;
var lastChar = 4;
var totalLenght = 0;
var bodyContent = "";
var urlServer = "http://192.168.0.14:8001/test?command=";

function searchBody(){
   console.log("Braille: Starting setup");
   totalLenght = document.body.children[0].textContent.lenght;
   bodyContent = document.body.children[0].textContent;
   document.body.style.border = "5px solid green";
   document.body.onkeydown = function(e){
           if(e.keyCode == 17){
                   document.body.children[0].innerHTML = "<p style='display:
inline'>" + bodyContent.substr(0, firstChar)
                   + "</p><p style='background-color: green; color: white;
display: inline'>"
```

```
                + bodyContent.substr(firstChar, 4)
                + "</p><p style='display: inline'>"
                + bodyContent.substr(lastChar, totalLenght) + "</p>";
                get(urlServer+bodyContent[firstChar]);
                get(urlServer+bodyContent[firstChar+1]);
                get(urlServer+bodyContent[firstChar+2]);
                get(urlServer+bodyContent[firstChar+3]);
                firstChar = lastChar;
                lastChar += 4;
        }
    };
    console.log("Braille Add-on Loaded");
}

function get(url)
{
        var xmlHttp = new XMLHttpRequest();
        xmlHttp.open( "GET", url, false );
        xmlHttp.send( null );
        return xmlHttp.responseText;
}
```

El resultado de nuestro *plugin* ejecutándose es el que podemos ver en la figura 3:

Fig. 3. Página de prueba que contiene solo text

3.1 Raspberry server y Kotlin

Para el servidor lo que hicimos fue crear un proyecto Kotlin [9] usando Gradle:

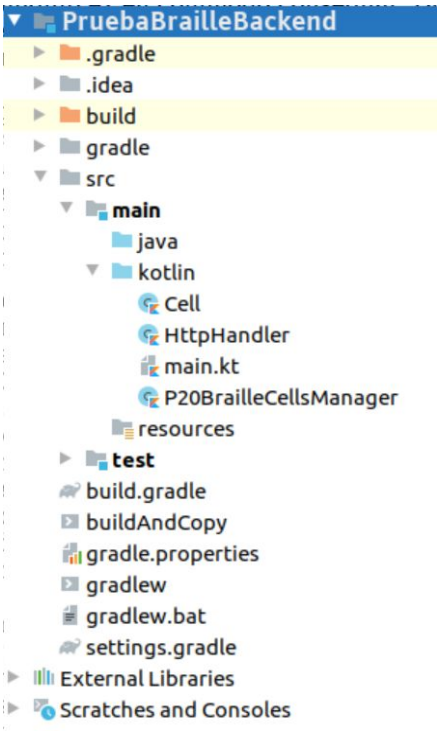


Fig. 4. Estructura de archivos para el servidor

Lo importante a destacar es que tenemos que agregar la librería de Pi4J en las dependencias de Gradle para poder ejecutar nuestro .jar con todo ya incluido:

```
dependencies {
  compile "org.jetbrains.kotlin:kotlin-stdlib-jdk8"
  compile group: 'com.pi4j', name: 'pi4j-core', version: '1.2'
  compile group: 'com.pi4j', name: 'pi4j-device', version: '1.2'
  compile group: 'com.pi4j', name: 'pi4j-gpio-extension', version: '1.2'
  compile group: 'com.pi4j', name: 'pi4j-native', version: '1.2'
  testCompile group: 'junit', name: 'junit', version: '4.12'
}
```

Nota: Quiero destacar esto porque es muy importante, Pi4J nos trae un instalador .deb que podemos ejecutar e instala bajo la carpeta /opt/ de nuestro sistema operativo Raspbian todas las librerías que necesitamos, también, agrega 1 ejecutable que compila y otro ejecutable que funciona como sandbox para correr nuestros proyectos. Lo malo de esto es que si o si debemos tener el .class descomprimido, no es posible correr un .jar completo, y esto para los que trabajamos o hemos alguna vez trabajado con Java es bastante engorroso.

La clase Cell.kt contiene representa una celda de Braille con la posición de cada dot:

```
/**
* Dots representation:
*
*       ||----------||
*       || P1    P4 ||
*       || P2    P5 ||
*       || P3    P6 ||
*       || P7    P8 ||
*       ||----------||
* */

class Cell(val name: String, var p1:Int, var p2:Int, var p3:Int, var p4:Int, var
p5:Int, var p6:Int, var p7:Int, var p8:Int)
```

El archivo main.kt se encarga de crear todos los componentes:

```
fun main() {
  val server = HttpServer.create(InetSocketAddress("192.168.0.14", 8001), 1)
  val threadPoolExecutor = Executors.newFixedThreadPool(1) as
ThreadPoolExecutor
  val p20BrailleCellsManager = P20BrailleCellsManager()
  p20BrailleCellsManager.init()
  server.createContext("/", HttpHandler(p20BrailleCellsManager))
  server.setExecutor(threadPoolExecutor);
  server.start()
  println("Server started at port 8001")
}
```

La clase HttpHandler posee dos cosas importantes, 1na es el *handler* que atiende cada petición Http, y la otra es un mapa donde tenemos cada letra en su representación en Braille. Esta última la voy a recortar para que el método no sea tan largo y sea más difícil de leer el código:

```
class HttpHandler(val p20BrailleCellsManager: P20BrailleCellsManager) :
HttpHandler {

  val commands = mapOf(
        "command=empty" to Cell("empty",0,0,0,0,0,0,0,0),
        "command=a" to Cell("char A",1,0,0,0,0,0,0,0),
              ...
        "command=9" to Cell("cell 9",0,0,1,0,1,0,0,0)
  )
  override fun handle(exchange: HttpExchange?) {
     val outputStream: OutputStream? = exchange?.responseBody
     val htmlResponse = exchange?.getRequestURI()?.getQuery()
     exchange?.sendResponseHeaders(200,
htmlResponse?.length?.toLong() ?: 0L)
     p20BrailleCellsManager.digitalWrite(commands[htmlResponse])
     println("GET Received: $htmlResponse")
     outputStream?.let { it.write(htmlResponse?.toByteArray()) }
     outputStream?.flush()
     outputStream?.close()
  }
}

class P20BrailleCellsManager {
  var gpioController: GpioController? = null
  var gpinDin: GpioPinDigitalOutput? = null
  var gpinStrobe: GpioPinDigitalOutput? = null
  var gpinClk: GpioPinDigitalOutput? = null
  val allDown = mutableListOf<Cell>()
  val delay = 25L
  fun init() {
     allDown.add(Cell("cell 1",0,0,0,0,0,0,0,0))
     allDown.add(Cell("cell 2",0,0,0,0,0,0,0,0))
     allDown.add(Cell("cell 3",0,0,0,0,0,0,0,0))
     allDown.add(Cell("cell 4",0,0,0,0,0,0,0,0))
     allDown.add(Cell("cell 5",0,0,0,0,0,0,0,0))
     allDown.add(Cell("cell 6",0,0,0,0,0,0,0,0))
     gpioController = GpioFactory.getInstance()
     gpinDin = gpioController!!.provisionDigitalOutputPin(RaspiPin.GPIO_07,
"PinDin", PinState.LOW)
     gpinStrobe =
gpioController!!.provisionDigitalOutputPin(RaspiPin.GPIO_00, "PinStrobe",
PinState.LOW)
```

```
    gpinClk = gpioController!!.provisionDigitalOutputPin(RaspiPin.GPIO_02,
"PinClk", PinState.LOW)
  }
  fun digitalWrite(cell: Cell?) {
    cell?.let {
      gpinStrobe?.state = PinState.LOW
      gpinClk?.state = PinState.LOW;gpinDin?.state = if (cell.p7 == 0)
PinState.HIGH else PinState.LOW; Thread.sleep(delay); gpinClk?.state =
PinState.HIGH; Thread.sleep(delay)
      gpinStrobe?.state = PinState.HIGH
      gpinStrobe?.state = PinState.LOW
      gpinClk?.state = PinState.LOW;gpinDin?.state = if (cell.p8 == 0)
PinState.HIGH else PinState.LOW; Thread.sleep(delay); gpinClk?.state =
PinState.HIGH; Thread.sleep(delay)
      gpinStrobe?.state = PinState.HIGH
      gpinStrobe?.state = PinState.LOW
      gpinClk?.state = PinState.LOW;gpinDin?.state = if (cell.p3 == 0)
PinState.HIGH else PinState.LOW; Thread.sleep(delay); gpinClk?.state =
PinState.HIGH; Thread.sleep(delay)
      gpinStrobe?.state = PinState.HIGH
      gpinStrobe?.state = PinState.LOW
      gpinClk?.state = PinState.LOW;gpinDin?.state = if (cell.p2 == 0)
PinState.HIGH else PinState.LOW; Thread.sleep(delay); gpinClk?.state =
PinState.HIGH; Thread.sleep(delay)
      gpinStrobe?.state = PinState.HIGH
      gpinStrobe?.state = PinState.LOW
      gpinClk?.state = PinState.LOW;gpinDin?.state = if (cell.p1 == 0)
PinState.HIGH else PinState.LOW; Thread.sleep(delay); gpinClk?.state =
PinState.HIGH; Thread.sleep(delay)
      gpinStrobe?.state = PinState.HIGH
      gpinStrobe?.state = PinState.LOW
      gpinClk?.state = PinState.LOW;gpinDin?.state = if (cell.p6 == 0)
PinState.HIGH else PinState.LOW; Thread.sleep(delay); gpinClk?.state =
PinState.HIGH; Thread.sleep(delay)
      gpinStrobe?.state = PinState.HIGH
      gpinStrobe?.state = PinState.LOW
      gpinClk?.state = PinState.LOW;gpinDin?.state = if (cell.p5 == 0)
PinState.HIGH else PinState.LOW; Thread.sleep(delay); gpinClk?.state =
PinState.HIGH; Thread.sleep(delay)
      gpinStrobe?.state = PinState.HIGH
      gpinStrobe?.state = PinState.LOW
      gpinClk?.state = PinState.LOW;gpinDin?.state = if (cell.p4 == 0)
PinState.HIGH else PinState.LOW; Thread.sleep(delay); gpinClk?.state =
PinState.HIGH; Thread.sleep(delay)
```

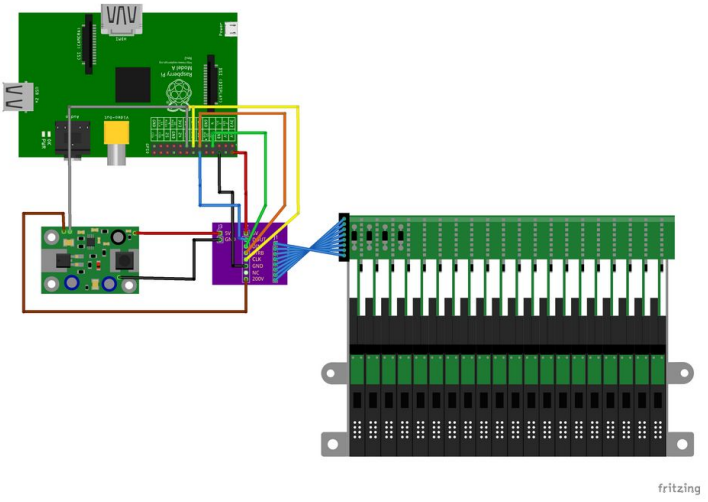
```
            gpinStrobe?.state = PinState.HIGH
        }
    }
}
```

Este código nos permite enviar señales al panel de control de las células P20, el servidor HTTP se queda escuchando posibles peticiones de algún carácter en particular. Todos los componentes en conjunto dan como resultado que si alguien hace una petición GET enviando un carácter en particular como por ejemplo: localhost:8001?command=j tengamos como resultado el carácter J impreso en la celula Braille P20.

Lo siguiente a realizar es el cableado, dado que es algo importante para poder abordar la instalación y puesta en marcha.

3.2 Hardware y cableado

Tanto para la codificación como el cableado del panel de las P20 y la raspberry nos basamos en un proyecto[10] open source encontrado en GitHub, donde el autor nos muestra el modo de instalación y el código fuente necesarios para poder conectarnos con el panel de las celular P20:



[10] https://github.com/bertrandmartel/metec-braille-driver

Fig. 5. Imagen tomada del repositorio github donde podemos ver el orden de los cables y su conexión con el panel de celular P20.

Desde la Raspberry usamos los siguientes pines:

cable rojo: 5V.
cable negro: ground.
cable amarillo: clock
cable naranja: strobe.
cable verde: out.
cable gris: ground.
cable marrón: 5V.

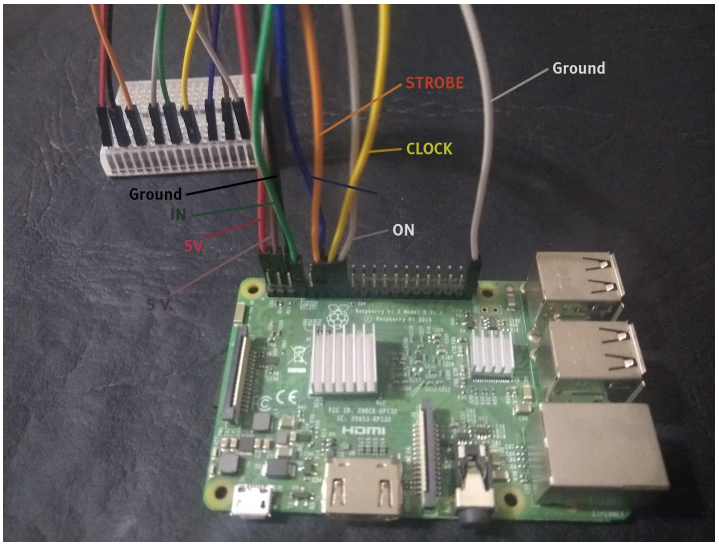


Fig. 6. Ejemplo de cómo debe quedar la instalación en nuestra Raspberry PI

utilizamos 2 pines de 5V y 2 pines de ground dado que tenemos que alimentar la placa que convierte 5V en 200V, además de que también le tenemos que mandar 5V a la placa de las células P20. Los pines STROBE y CLOCK son utilizados para bloquear la señal que vamos a enviar o no, antes de enviar un pulso o no enviarlo por el pin IN debemos apagar tanto el STROBE como el CLOCK, de ese modo le indicamos al panel de células P20 que estamos queriendo levantar o bajar un dot de cada celular.

Por el lado del transformador lo que hicimos fue soldar 4 cables de los cuales usamos los colores:

cable blanco: es la salida de 200V. este cable se conecta con el panel de las P20
cable naranja: es la entrada de 5V. que viene desde la Raspberry
cable azul: es una salida que puede ser conectada a un pin GPIO de la Raspberry con el propósito de ver si tenemos señal y con el cual podemos determinar si esta encendido el panel o no, a fines de este experimento no lo usamos.
cable celeste: es el negativo proveniente de la raspberry, ground.

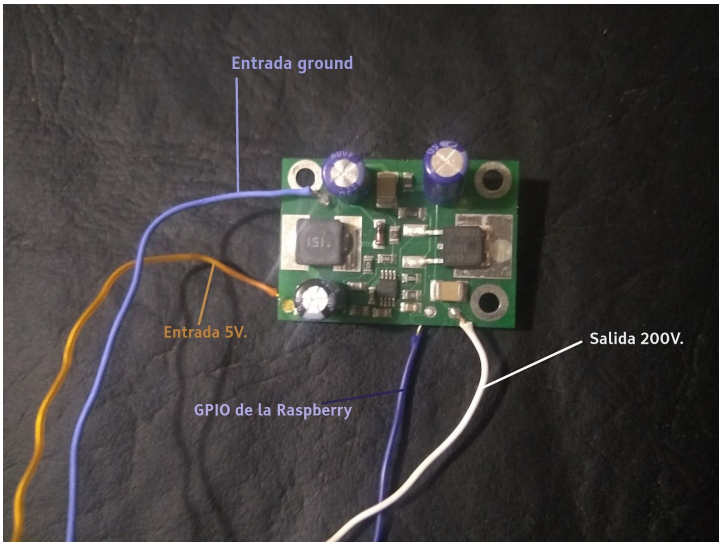


Fig. 7. Ejemplo de conexión de cables en el panel de alimentación, esta placa transforma 5VC en 200 VC.

Por último, el cableado del panel P20 se ve de la siguiente manera:

cable blanco: es el cable de 200 V que sale de la placa transformadora
cable verde oscuro: no es utilizado
cable verde claro: ground
cable azul oscuro: Clock
cable marron oscuro: Strobe
cable marron claro: IN
cable naranja: 5V provenientes desde la Raspberry

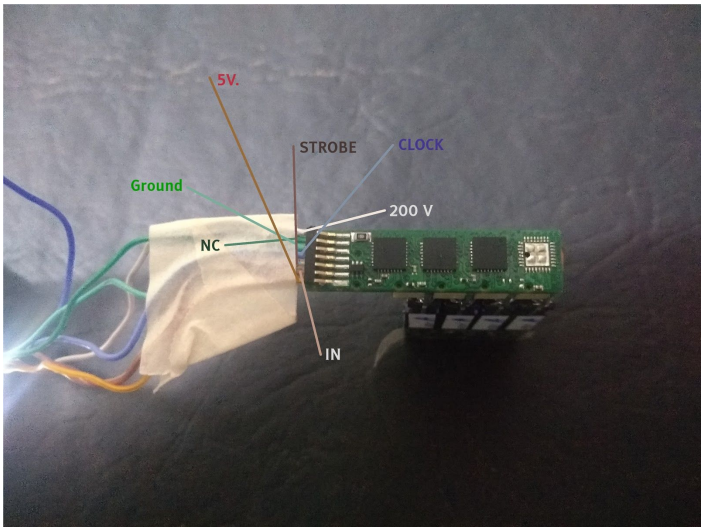


Fig. 8. Cableado de nuestro panel de celular P20

4 Pruebas, Conclusión y trabajo a futuro

En las pruebas realizadas se utilizó una página web creada por nosotros, donde el contenido era el “lorem ipsum”, esto nos ayudó a ver que sin importar del largo del contenido el *plugin* era capaz de imprimir en Braille todo el sitio. Obviamente una de las limitantes más grandes e importantes que tuvimos fue la cantidad de caracteres a mostrar, 4 caracteres es muy poco para cualquier palabra y muchas veces terminamos cortando la información en dos o tres grupo de caracteres de 4, algo que claramente fue de utilidad simplemente como una estrategia inicial de prueba. Como resultado, podemos ver que los elementos que están con *highlight* en la página web creada para el testeo, al presionar el botón de control (esa fue la tecla que utilizamos en el *plugin* de Firefox como ejemplo, pero puede ser cualquiera) se seleccionan los próximos 4 caracteres, y así se puede continuar avanzando en el documento y leyendo todo lo necesario.

Nos gustaría destacar que llevar el texto a una célula de Braille en lugar de un lector de voz nos permite mostrar símbolos que son difíciles o hasta imposibles de pronunciar, esto abre un gran camino a que todo el contenido de un sitio sea perfectamente accesible por una persona no vidente sin mucho esfuerzo por lado de los programadores.

En la siguiente imagen, se puede ver la salida en Braille de las primeras 4 letras que forman mi nombre F, R,A,N:

Fig. 9. Ejemplo de salida

4.1 Conclusión

En primer medida nos gustaría destacar que la problemática propuesta en este artículo y los objetivos propuestos pudieron cumplirse. Es posible generar una herramienta que interactúe con un sitio web, con una página de escritorio o con el mismo sistema operativo, que esa herramienta, envíe datos sobre un protocole existente, confiable y estandarizado como lo es el HTTP y que su receptor sea una placa RaspberryPI customizada por nosotros y conectada a una tira de células Braille. A modo de prueba de concepto quedó demostrado que la integración de estos elementos es posible y se pueden realizar muchas mejoras en ambos extremo con el fin de ofrecer una mejor experiencia a los usuarios finales. Así mismo, sabemos que hoy en día existen muchos productos similares al que realizamos nosotros, y que lamentablemente no podemos romper el ruido del código HTML o Javascript metido en muchos de los datos que estamos analizando. Pero podemos agregar, que es mucho más fácil interpretar código sobre una tira de celular P20 que en un lector de voz, esto abre un terreno nuevo para experimentar a los que deseen adentrarse en el mundo de la accesibilidad.
Es importante destacar que los comandos difíciles de pronunciar por una autómata de voz son sencillamente reproducibles en una tira de Braille, este tipo de enfoque abre una puerta para desarrollos más inclusivos para personas invidentes.

4.2 Trabajo a futuro

Como un trabajo a futuro sería interesante el poder mostrar esos datos en la tira de celular Braille P20 en lugar de filtrarlos, pero para lo cual se necesitaría extender el lenguaje de braille de modo que sea posible analizar dichos caracteres.
La propuesta acá mencionada implicaría el poder crear un nuevo código de Braille capaz de soportar caracteres especiales, en especial todos los utilizados en la programación, de esta manera podremos mostrar todo el contenido de una página sin filtrar absolutamente nada.
Otro caso interesante a probar, sería el de utilizar las células de braille 2D de Metec, para poder así mostrar el contenido en distintas posiciones de la pantalla, dando un sensación más real a lo que se está queriendo transmitir en el sitio web.
Uno de los puntos más importantes que nos gustaría dejar como trabajo a futuro es la posibilidad de testear lo propuesto en este artículo con los usuarios finales, nosotros sabemos que es muy importante la necesidad de asegurarnos que la herramienta está brindando una mejora a lo existente, que es fácil de utilizar y que se adapta a todo tipo de sitio web o usuario. De este modo quedaría no sólo demostrado que es posible realizar dicha integración entre herramientas, como quedó demostrado con este análisis y prueba de concepto, sino que efectivamente este trabajo tiene un valor agregado en la comunidad, entendiendo que nuestros desarrollos tecnológicos deben responder a las demandas de la sociedad mejorando su calidad de vida.

Referencias

[1] O'Connor, Joshue. Pro HTML5 Accessibility. APress, 2012.
[2] Cunningham, Katie. Accessibility Handbook. O'Reilly Media, 2012.
[3] Shane Hudson. JavaScript Creativity: Exploring the Modern Capabilities of JavaScript and HTML5. Springer Science+Business Media New York, 2014.
[4] Picabea, Juan. Tecnología e inclusión. National Scientific and Technical Research Council. Buenos Aires, Argentina., 2015.
[5] Líneas Braille. https://www.metec-ag.de/en/produkte-braille-zeilen.php
[6] Traducción gráfico 2D a Braille. https://www.metec-ag.de/en/produkte-graphik-display.php.
[7] McEwen, Adrian. Designing the Internet of Things. Editorial Chichester : Wiley. Reino Unido, 2014.
[8] Stallings, William. Foundations of modern networking. Editorial Pearson. Estados Unidos, 2016
[9] Dmitry Jemerov, Svetlana Isakova. Kotlin in Action. Manning, 2017.
[10] Rodríguez Eguren, Sebastián. Análisis del uso de un cluster de Raspberry Pi para cómputo de alto rendimiento. Congreso Argentino de Ciencias de la Computación. Tandil, Argentina, 2018.