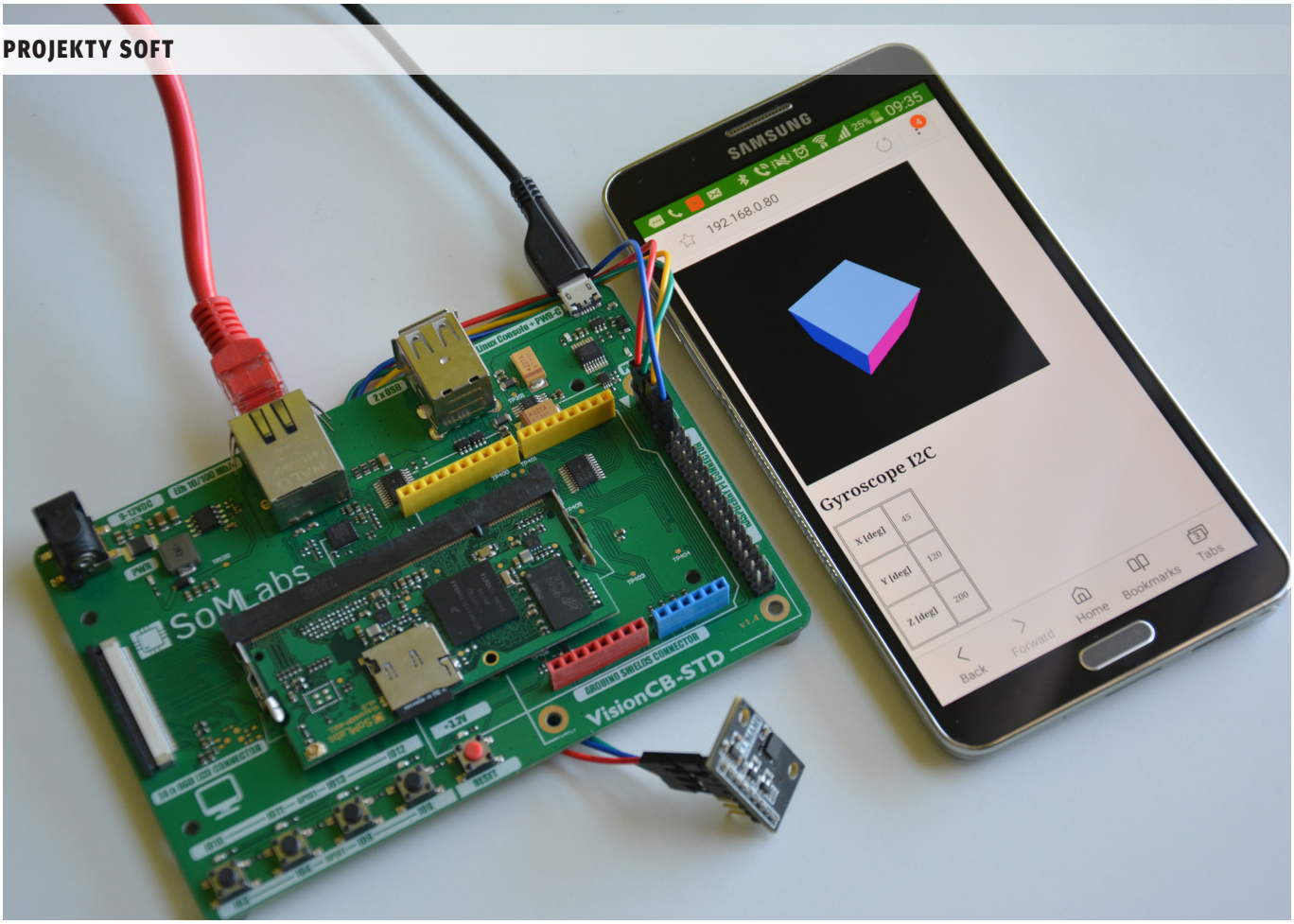


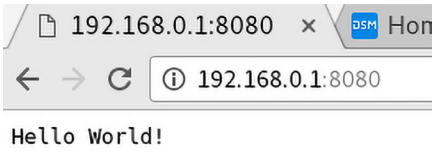
Serwer WWW z elementami grafiki 3D (1)

Praktyczne wykorzystanie pakietów Node.js oraz Three.js w systemach wbudowanych

Jedną z niewątpliwych zalet wykorzystania systemu operacyjnego Linux w procesie projektowania urządzeń wbudowanych jest szybki i łatwy dostęp do otwartych, darmowych i wolnych (w sensie wolności) repozytoriów oprogramowania implementujących m.in. rozbudowane stosy graficzne, protokoły sieciowe czy złożone algorytmy obliczeń. Fakt ten odgrywa szczególnie ważną rolę, kiedy zadanie stawiane przed naszym urządzeniem może zostać chociaż częściowo zrealizowane za pomocą istniejących już pakietów oprogramowania. Samodzielna implementacja stosów obsługi USB, Ethernet, Bluetooth czy bardziej złożonych interfejsów graficznych użytkownika może być czasochłonna i podatna na błędy. Wykorzystując gotowe sterowniki i pakiety oprogramowania (o ile są one udostępnione na dogodnej licencji), zyskujemy nie tylko czas, ale i pewność, że oprogramowanie zostało przetestowane przez tysiące innych użytkowników Linuksa.

W artykule, na przykładzie komputera jednopłytkowego *VisionSOM* firmy *SoMLabs* [1], przedstawiono przykład prostego i szybkiego tworzenia bardziej rozbudowanych projektów sprzętowo-programowych, z wykorzystaniem bibliotek gotowego i darmowego oprogramowania. Wykorzystując wyłącznie minimalną funkcjonalność środowiska uruchomieniowego *Node.js* [2] oraz biblioteki *Three.js* [3], przygotujemy prosty serwer WWW, prezentujący wyniki danych pomiarowych (odczytanych z modułu żyroskopu komunikującego się po magistrali I²C) w postaci animowanej kostki 3D. Proces budowy kompletnego projektu został przedstawiony etapowo – od najprostszej implementacji serwera WWW wyświetlającej napis „Hello World", poprzez podział serwera na funkcje front-end/back-end oraz implementację odczytu danych z procesu obsługującego moduł żyroskopu (tematyka pierwszej części artykułu), aż do przygotowania właściwej obsługi modułu żyroskopu i prezentacji wyników pomiarów w postaci animowanej kostki 3D (przedstawimy w drugiej części artykułu).

W artykule wykorzystano komputer jednopłytkowy VisionSOM pracujący pod kontrolą dystrybucji Debian. Autor celowo pominął opisy przygotowania komputera do pracy – przygotowanie karty SD z systemem Debian oraz konfiguracja i kompilacja jądra systemu i plików Device Tree została przedstawiona w ramach artykułu „Emulator konsoli NES w systemie Linux na komputerze VisionSOM" publikowanego na łamach Elektroniki Praktycznej 03/2018. Komplet informacji został również udostępniony na stronie Wiki producenta komputera [4].



Rysunek 1. Implementacja erwera WWW

Node.js – systemy wbudowane i JavaScript?

Czym jest *Node.js*? Jest to wieloplatformowe środowisko uruchomieniowe JavaScript udostępnione na licencji *open-source*. Inaczej ujmując ogólnodostępną definicję – platforma *Node.js* umożliwia uruchomienie kodu JavaScript poza przeglądarką internetową. Należy podkreślić, że samo *Node.js* nie jest serwerem, umożliwia jednak proste i szybkie utworzenie serwera HTTP oraz bardziej złożonych aplikacji internetowych. Ponieważ kod programu jest uruchamiany poza przeglądarką, programista ma możliwość tworzenia typowych rozwiązań "*server-side*", czyli takich w których aplikacja może realizować odczyt/zapis systemu plików, baz danych oraz co równie istotne pod kątem realizacji urządzeń wbudowanych – wchodzić w interakcję z dołączonym do systemu urządzeniami peryferyjnymi – czujnikami, aktuatorami, itp.

Ze względu na dużą popularność platformy *Node.js* (warto nadmienić, że korzystają z niej takie serwisy jak *Netflix*, *PayPal*, *LinkedIn* czy *Uber*), gotowe pakiety oprogramowania są obecnie dostępne w niemal wszystkich dystrybucjach linuksowych. Dla dystrybucji *Debian*, instalacja pakietu nodejs przebiega w sposób standardowy dla narzędzia apt-get:

```
root@localhost:~# apt-get install nodejs
...
Selecting previously unselected package nodejs.
Preparing to unpack .../nodejs_4.8.2~dfsg-1_armhf.deb ...
Unpacking nodejs (4.8.2~dfsg-1)
...
```

Aby przetestować poprawność instalacji, wywołajmy komendę nodejs -v:

```
root@localhost:~# nodejs -v
v4.8.2
```

Jeśli w wyniku powyższego polecenia został wyświetlony numer wersji oprogramowania *Node.js*, możemy przystąpić do przygotowania najprostszej implementację serwera WWW.

Prosta implementacja serwera WWW

Implementację serwera WWW rozpoczynamy od utworzenia pliku *main.js*. W pierwszej linii kodu zaimportujmy wbudowany w *Node.js* moduł http:

```
var http = require ('http');
```

oraz zdefiniujmy numer portu na którym będzie nasłuchiwał tworzony serwer:

```
var PORT = 8080;
```

Kolejnym krokiem jest utworzenie właściwego serwera poprzez wywołanie metody createServer() na module http:

```
var server = http.createServer (/*...*/);
```

Metoda createServer() jako argument przyjmuje funkcje zwrotną, której zadaniem jest obsługa zapytań przychodzących do serwera. Funkcja ta przyjmuje dwa argumenty:

- request – argument zawiera informacje o szczegółach zapytania,
- response – obiekt zawierający metody i własności do obsługi odpowiedzi.

W naszej prostej implementacji serwera, w odpowiedzi na zapytanie klienta prześlemy: informację z kodem odpowiedzi (*200–OK*), typ zwracanego dokumentu (*text/plain* – czyste dane tekstowe) oraz napis *„Hello World!"*. Uzupełniony kod metody createServer() o obsługę zapytania został przedstawiony poniżej:

```
var server = http.createServer (function handler (request, response) {
  response.writeHead (200, {'Content-Type': 'text/plain'});
  response.end ('Hello World!');
});
```

Ostatnim etapem, jest wywołanie metody listen() wraz z przekazaniem numeru portu, na którym serwer będzie nasłuchiwał nadchodzących połączeń:

```
server.listen (PORT);
```

Kompletną zawartość pliku *main.js* zamieszczono na **listingu 1**. Po zapisaniu zmian w pliku *main.js*, możemy uruchomić skrypt za pomocą polecenia:

```
nodejs main.js
```

Jeżeli próba uruchomienia skryptu została zakończona sukcesem, wpisując w oknie przeglądarki adres IP podłączonego do sieci komputera *VisionSOM*, zobaczymy pierwszy rezultat działania serwera WWW, jak pokazano na **rysunku 1**.

Listing 1. Skrypt main.js z implementacją prostego serwera WWW

```
var http = require ('http');
var PORT = 8080;

var server = http.createServer (function handler (request, response) {
  response.writeHead (200, {'Content-Type': 'text/plain'});
  response.end ('Hello World!');
});
server.listen (PORT);
```

Listing 2. Serwer WWW z podziałem na funkcje fron-end/back-end

```
var http = require ('http');
var fs = require ('fs');
var index = fs.readFileSync (__dirname + '/index.html');
var PORT = 8080;

var server = http.createServer (function handler (request, response) {
  response.writeHead (200, {'Content-Type': 'text/html'});
  response.end (index);
});

server.listen (PORT);
```

Serwer WWW z podziałem na funkcje front-end/back-end

Bezpośrednie umieszczenie „kodu strony" – w postaci napisu *„Hello World!"* – w funkcji response.end() nie wpływa znacząco na czytelność kodu, jednak nietrudno wyobrazić sobie sytuację, że budowany przez nas serwis zaczyna się rozrastać, a wprowadzane znaczniki HTML znacznie zwiększają objętość kodu strony, powodując że skrypt *main.js* może stać się mało czytelny i trudny w zarządzaniu.

W takiej sytuacji niezbędne jest wprowadzenie jasnego podziału na *front-end* (czyli właściwą stronę udostępnianą użytkownikowi) oraz *back-end* (czyli kod realizujący zadania stawiane przed serwerem).

Wprowadzenie podziału na *front-end* i *back-end* wymaga jedynie kosmetycznych zmian w skrypcie *main.js* z listingu 1. Zmodyfikowany skrypt *main.js* przedstawiono na **listingu 2** (pogrubioną czcionką wyróżniono zmiany wprowadzone w stosunku do poprzedniej wersji skryptu).

Z wykorzystaniem wbudowanego modułu fs (pozwalającego na przeprowadzanie szeregu operacji I/O na plikach) w sposób synchroniczny wczytujemy zawartość pliku *index.html*, umieszczonego w tym samym folderze jak skrypt *main.js*. Ponieważ wczytany plik jest prostą stroną HTML, zmieniamy zawartość pola Content-Type na text/html. W wywołaniu response.end() przesyłamy użytkownikowi zawartość pliku *index.html*. Dla kompletności zadania, utwórzmy również najprostszy plik HTML – *index.html* – jak przedstawiono to na **listingu 3**. Po zakończonej edycji plików *main.js* oraz *index.html*, sprawdźmy poprawność naszego kodu poprzez ponowne uruchomienie serwera:

```
nodejs main.js
```

Komunikacja front-end<–>back-end z wykorzystaniem socket.io

Wprowadzenie wyraźnego podziału na sekcje *front-end* i *back-end* stawia przed nami kolejne zadanie do wykonania – zapewnienie sprawnej komunikacji i wymiany danych w „czasie rzeczywistym" pomiędzy tymi modułami. Dlaczego w czasie rzeczywistym? Protokół HTTP jest typowym protokołem typu żądanie-odpowiedź, w którym to rolę żądającego pełni klient/przeglądarka internetowa. Rozwiązanie to spełnia swoje zadanie w przypadku gdy to klient chce przesłać dane do serwera. Niestety w sytuacji gdy serwer chce poinformować odbiorcę o aktualizacji danych (np. nowych odczytach z czujników temperatury, których wyniki powinny być wyświetlane w czasie rzeczywistym w interfejsie przeglądarkowym), nie może on zainicjować połączenia z klientem. Modyfikacja „w locie" pliku *index.html* przez kod serwera oraz cykliczne odświeżanie strony przez klienta nie brzmią jak idealne rozwiązanie problemu. W takiej sytuacji pomocną dłoń wyciąga do nas biblioteka *socket.io* [5] zapewniająca połączenie pomiędzy stroną WWW (front-endem) a skryptem uruchomionym na serwerze (back-endem). *Socket.io* jest biblioteką języka JavaScript, której zadaniem jest ułatwienie pracy z protokołem *WebSocket* (który to natomiast jest częścią specyfikacji HTML5, umożliwiającą dwustronną komunikację klient-serwer w czasie rzeczywistym). Biblioteka *socket.io* składa się z tzw. części serwerowej (będącej modułem dla platformy *Node.js*) oraz klienckiej (dla przeglądarek internetowych). Bazując na kodzie skryptu *main.js* oraz strony *index.html* z poprzedniego podrozdziału, przejdźmy do praktycznej implementacji.

Rozbudowę skryptu *main.js* rozpoczynamy od zaimportowania modułu *socket.io* (szczegóły dotyczące instalacji zewnętrznego pakietu *socket.io* przedstawiono w ramce:

```
var io = require ('socket.io').
listen(server);
```

Pakiet socket.io nie jest częścią platformy Node.js i wymaga dodatkowej instalacji. Do instalacji dodatkowych pakietów można wykorzystać dystrybuowany wraz z Node.js, manager pakietów npm [6]: npm install socket.io.

W następnym kroku utwórzmy *event-handler* dla zdarzenia connection (które jest wywoływane każdorazowo, gdy do serwera podłączony zostanie nowy klient), wyświetlający krótki komunikat na standardowym wyjściu:

```
io.on ('connection', function
(socket) {
  console.log ('We have
new connection!');
});
```

W docelowym rozwiązaniu aplikacja serwera będzie przesyłała do przeglądarki użytkownika informacje odczytane z modułu żyroskopu. Sposób, w jaki zostanie zrealizowana komunikacja pomiędzy procesem obsługującym żyroskop a serwerem WWW, zostanie omówiony w kolejnym podrozdziale artykułu. Na potrzeby obecnego etapu prac, przygotujmy prostą funkcję send_time(), która z interwałem jednej sekundy, prześle do wszystkich podłączonych klientów aktualny czas:

```
function send_time() {
  io.emit ('time', {message: new
Date().toISOString()});
}
setInterval (send_time, 1000);
```

W ciele funkcji send_time() wysyłamy rozgłoszeniową wiadomość time z aktualnym czasem serwera, skierowaną do wszystkich aktualnie podłączonych klientów. Pełny kod skryptu *main.js* wraz z wyszczególnieniem wprowadzonych zmian (w odniesieniu do *listingu 2*), został przedstawiony na **listingu 4**.

Ostatnim etapem zadania jest integracja biblioteki *socket.io* z udostępnianą przez serwer stroną *index.html*. Integrację biblioteki rozpoczniemy od dołączenia w sekcji <head> biblioteki *socket.io*:

```
<script src='/socket.io/socket.
io.js'></script>
```

Również w sekcji <head> utwórzmy prosty skrypt realizujący nawiązanie połączenia z serwerem oraz odbiór komunikatów (należy pamiętać, że kod zawarty w tagach <script></script> zostanie uruchomiony przez przeglądarkę, a więc komputer PC użytkownika):

```
var socket = io();
socket.on ('time', function (data)
{
  /* TODO */
});
```

Zanim przystąpimy do uzupełnienia kodu *event-handler'a* dla zdefiniowanego przez nas zdarzenia time, w sekcji <body> strony HTML utwórzmy akapit z identyfikatorem test, w miejscu którego wyświetlone zostaną dane otrzymane z serwera WWW:

```
<p id="test">JavaScript can
change HTML content.</p>
```

Mając określone pole w którym otrzymane dane będą wyświetlane, możemy uzupełnić implementację *event-handler'a* dla zdarzenia time:

```
socket.on ('time', function (data)
{
  document.getElementById("test").
innerHTML = data.message;
});
```

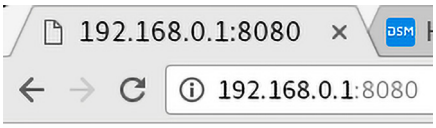
Pełna zawartość pliku *index.html* (wraz z wyróżnieniem zmian wprowadzonych w stosunku do *listingu 3*) została przedstawiona na **listingu 5**. Przy ponownym uruchomieniu serwera poleceniem *nodejs main.js* oraz odświeżeniu zawartości strony WWW, powinniśmy uzyskać efekt przedstawiony na **rysunku 2**.

Listing 3. Zawartość pliku index.html

```
<!DOCTYPE html>
<html>
  <head>
  </head>
  <body>
    <h1>Hello World!</h1>
  </body>
</html>
```

Listing 4. Skrypt main.js zintegrowany z biblioteką socket.io

```
var http = require ('http');
var fs = require ('fs');
var index = fs.readFileSync (__dirname + '/index.html');
var PORT = 8080;

var server = http.createServer (function handler (request, response) {
  response.writeHead (200, {'Content-Type': 'text/html'});
  response.end (index);
});

var io = require ('socket.io').listen(server);

io.on ('connection', function (socket) {
  console.log ('We have new connection!');
});

function send_time() {
  io.emit ('time', {message: new Date().toISOString()});
}
setInterval (send_time, 1000);

server.listen (PORT);
```



Hello World!

2017-10-01T19:44:59.317Z

Rysunek 2. Przykład komunikacji między serwer WWW a przeglądarką internetową

Serwer WWW – odczyt danych z procesu obsługi modułu żyroskopu

Na obecnym etapie realizacji projektu wiemy już jak nawiązać prostą komunikację pomiędzy serwerem a klientem. Do pełnej realizacji celu brakuje wciąż informacji w jaki sposób „poinformować" serwer o aktualnych danych pomiarowych, które będą zwracane przez podłączony do komputera za pomocą magistrali I²C moduł żyroskopu. Do najprostszych rozwiązań tego problemu możemy zaliczyć np. bezpośrednią implementacją obsługi żyroskopu w kodzie serwera – z wykorzystaniem operacji na plikach lub gotowych modułów *Node.js*, instalowanych poprzez menadżer pakietów *npm*. Przykładem takiego modułu może być pakiet i2c, instalowany poleceniem:

```
npm install i2c,
```

który udostępnia proste API do realizacji niskopoziomowych operacji zapisu/odczytu danych na magistrali, np.:

```
var i2c = require('i2c');
var wire = new i2c(address,
{device: '/dev/i2c-1'});
wire.writeByte(byte, function(err)
{});
wire.writeBytes(command, [byte0,
byte1], function(err) {});
wire.readByte(function(err, res)
{});
```

Pomimo tego, że API modułu i2c jest bardzo czytelne, a ewentualna reimplementacja istniejących kawałków kodu lub bibliotek z języka C nie powinna być problematyczna, do realizacji projektu użyjemy alternatywnego podejścia. Całość obsługi modułu żyroskopu zostanie przygotowana w języku C i skompilowana do postaci pliku wykonywalnego gyro-i2c (patrz ramka poniżej). Dlaczego? Po pierwsze, większość programistów związanych z niskopoziomowymi systemami wbudowanymi nie zaryzykuje implementacji obsługi sprzętu w JavaScript (język C jest tutaj bardziej naturalnym wyborem), a po drugie – metoda ta może być przydatna, gdy nie posiadamy dostępu do kodów źródłowych aplikacji obsługujących sprzęt – wówczas jedyną możliwością jest odczyt danych ze standardowego wyjścia procesu.

Do uruchomienia i komunikacji z procesem gyro-i2c z poziomu *Node.js* wykorzystamy wbudowany moduł child_process. Za pomocą metody spawn() utworzymy nowy proces potomny oraz zdefiniujemy dla niego funkcje zwrotną obsługi standardowego wyjścia (wywoływaną w chwili gdy program gyro-i2c zwróci kolejną porcję danych z wynikami pomiarów).

Analogicznie, jak w poprzednich podpunktach, do realizacji tego etapu wykorzystamy pliki z **listingu 4** oraz **listingu 5**. Edycję rozpoczniemy od skryptu *main.js* w którym zaimportujemy wbudowany moduł child_process:

```
var spawn = require('child_
process').spawn;
```

W kolejnym kroku, za pomocą wywołania spawn() utworzymy nowy proces potomny realizujący kod programu gyro-i2c (skopiowany uprzednio do folderu */tmp*):

```
var child = spawn ('/tmp/
gyro-i2c');
```

Ostatnią modyfikacją w skrypcie *main.js* jest dodanie funkcji zwrotnych do obsługi kanałów stdout (funkcja przesyła odczytane dane do przeglądarki w postaci komunikatu xyz) oraz stderr (funkcja wypisuje w konsoli dane odczytane ze standardowego strumienia błędów):

```
child.stdout.on ('data', function
(data) {
  io.emit ('xyz', {message: data.
toString().split('\n')[0]});
});
child.stderr.on ('data', function
(data) {
  console.log ('stderr: ' + data);
});
```

Warto również zaimplementować obsługę zdarzenia close, która poinformuje o zakończeniu procesu potomnego i zwróconym przez niego kodzie wyjścia:

```
child.on ('close', function (code)
{
  console.log ('exit: ' + code);
});
```

Pełna zawartość pliku *main.js* (wraz z wyróżnieniem zmian wprowadzonych w stosunku do listingu 4) została przedstawiona na **listingu 6**.

Przystosujmy również plik *index.html* do nowych wymagań projektu, tj. wyświetlenia wartości trzech pomiarów dla osi X, Y oraz Z. W tym celu w sekcji <body> utwórzmy prostą tabelę zawierającą identyfikatory pól x_val, y_val oraz z_val – **listing 7**.

W sekcji <head> zmodyfikujmy kod obsługi wiadomości xyz. Odczytana linia danych zostanie podzielona względem separatora ' ' (spacja), a wyniki pomiarów przypisane do poszczególnych identyfikatorów pól – **listing 8**.

Listing 5. Strona index.html zintegrowana z biblioteką socket.io

```
<!DOCTYPE html>
<html>
  <head>
    <script src='/socket.io/socket.io.js'></script>
    <script>
      var socket = io();
      socket.on ('time', function (data) {
        document.getElementById("test").innerHTML = data.message;
      });
    </script>
  </head>
  <body>
    <h1>Hello World!</h1>
    <p id="test">JavaScript can change HTML content.</p>
  </body>
</html>
```

Jak wspomniano na wstępie artykułu, szczegóły związane z przygotowaniem obsługi modułu żyroskopu w języku C zostaną przedstawione w kolejnej części artykułu. Na potrzeby aktualnego etapu, przygotujmy najprostszą aplikację generującą losowe wyniki dla trzech osi (X, Y, Z). Dane będą wypisywane na standardowe wyjście z interwałem 1 sekundy w formacie: <pomiarX><spacja><pomiarY><spacja>pomiarZ>

[1] Przykładowe implementacja:

```
/* … */
  while (1) {
    sleep (1);
    printf („%d %d %d\n", rand()%360, rand()%360, rand()%360);
    fflush (stdout);
  }
/* … */
```

[2] Kompilacja:

```
gcc main.c -o /tmp/gyro-i2c
```

Listing 6. Skrypt main.js z zaimplementowaną obsługą procesu potomnego

```
var http = require ('http');
var fs = require ('fs');
var spawn = require('child_process').spawn;
var index = fs.readFileSync (__dirname + '/index.html');
var PORT = 8080;

var server = http.createServer (function handler (request, response) {
  response.writeHead (200, {'Content-Type': 'text/html'});
  response.end (index);
});
var io = require ('socket.io').listen(server);

io.on ('connection', function (socket) {
  console.log ('We have new connection!');
});

var child = spawn ('/tmp/gyro-i2c');

child.stdout.on ('data', function (data) {
  io.emit ('xyz', {message: data.toString().split('\n')[0]});
});

child.stderr.on ('data', function (data) {
  console.log ('stderr: ' + data);
});

child.on ('close', function (code) {
  console.log ('exit: ' + code);
});

server.listen (PORT);
```

Listing 8. Parsowanie otrzymanych danych i przypisanie do identyfikatorów pól

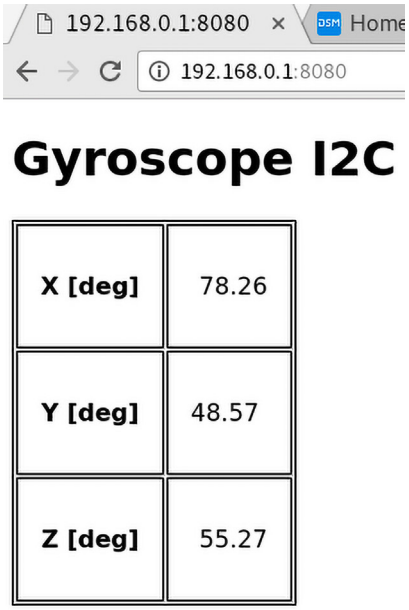
```
<script>
      var socket = io();
      socket.on ('xyz', function (data) {
        var arr = data.message.split(" ");
        document.getElementById("x_val").innerHTML = arr[0];
        document.getElementById("y_val").innerHTML = arr[1];
        document.getElementById("z_val").innerHTML = arr[2];
      });
</script>
```

Listing 9. Wyświetlanie wyników pomiarów postaci tabeli – plik index.html

```
<!DOCTYPE html>
<html>
  <head>
    <style>
      table, th, td {
        border: 1px solid black;
      }
      th, td {
        border: 1px solid black;
        padding: 15px;
      }
    </style>
    <script src='/socket.io/socket.io.js'></script>
    <script>
      var socket = io();
      socket.on ('xyz', function (data) {
        var arr = data.message.split(" ");
        document.getElementById("x_val").innerHTML = arr[0];
        document.getElementById("y_val").innerHTML = arr[1];
        document.getElementById("z_val").innerHTML = arr[2];
      });
    </script>
  </head>

  <body>
    <h1>Gyroscope I2C</h1>
    <table>
      <tr>
        <th>X [deg]</th>
        <td><p id="x_val">---</p></td>
      </tr>
      <tr>
        <th>Y [deg]</th>
        <td><p id="y_val">---</p></td>
      </tr>
      <tr>
        <th>Z [deg]</th>
        <td><p id="z_val">---</p></td>
      </tr>
    </table>
  </body>
</html>
```

Listing 7. Tabela z wynikami pomiarów żyroskopowych – plik index.html

```
<table>
  <tr>
    <th>X [deg]</th>
    <td><p id="x_val">---</p></td>
  </tr>
  <tr>
    <th>Y [deg]</th>
    <td><p id="y_val">---</p></td>
  </tr>
  <tr>
    <th>Z [deg]</th>
    <td><p id="z_val">---</p></td>
  </tr>
</table>
```



Rysunek 3. Prezentacja wyników pomiarów w oknie przeglądarki internetowej

Dla poprawienia estetyki utworzonej strony, w sekcji head dodajmy wpis formatujący wygląd tabeli. Pełna zawartość pliku *index.html* (wraz z wyróżnieniem zmian wprowadzonych w stosunku do *listingu 5*) została przedstawiona na **listingu 9**.

Po ponownym uruchomieniu serwera oraz odświeżeniu zawartości strony internetowej, generowane przez aplikację gyro-i2c, losowe wyniki pomiarów powinny zostać wyświetlone i na bieżąco aktualizowane – **rysunek 3**.

Łukasz Skalski
contact@lukasz-skalski.com

Bibliografia:

[1] *https://somlabs.com*
[2] *https://nodejs.org/en*
[3] *https://threejs.org*
[4] *http://wiki.somlabs.com*
[5] *https://socket.io*
[6] *https://www.npmjs.com*