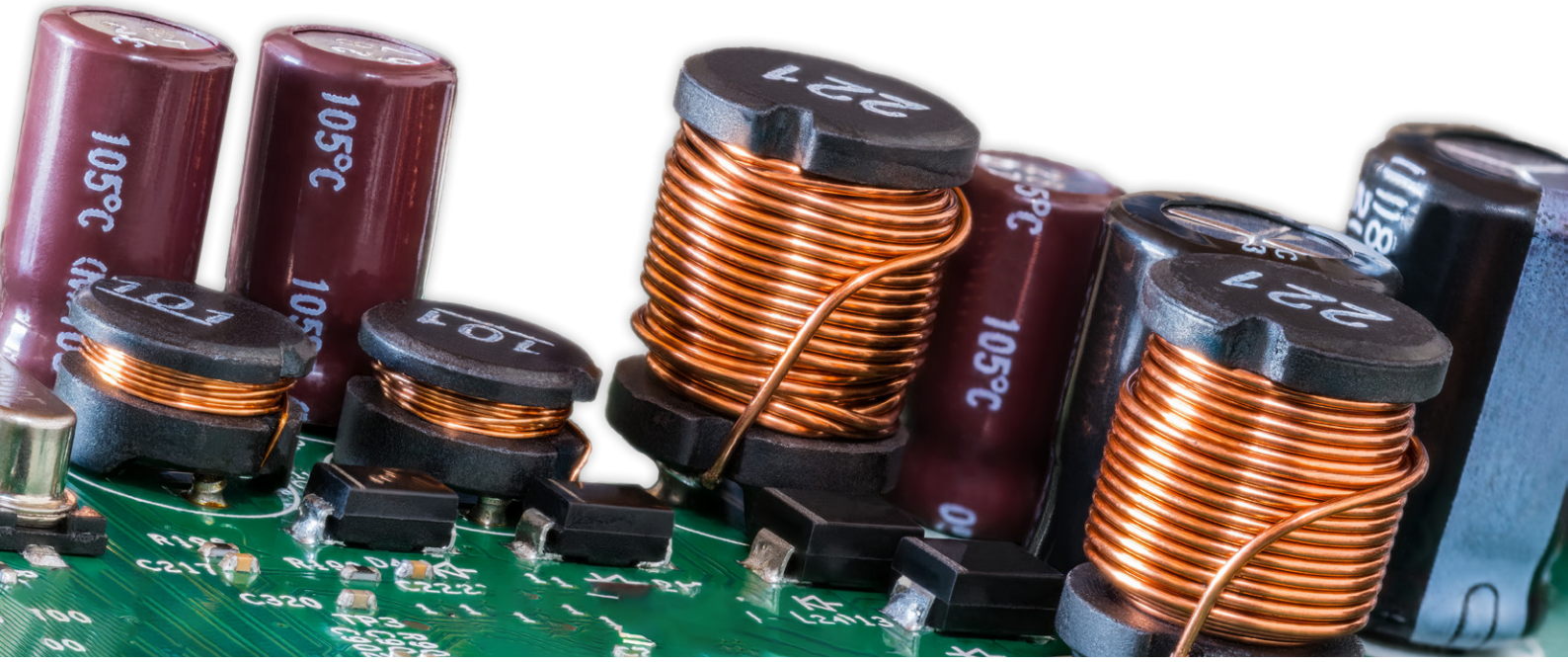
analyseur d'impédance **basé sur un ESP32**

simple, comportant peu de composants et de faible coût !



Volker Ziemann (Suède)

Un ESP32 analyseur d'impédance, pouvant également vérifier la fréquence de résonance d'un réseau LC ? Oui, ce microcontrôleur peu couteux, associé à une poignée de composants externes peut le faire, tout en offrant une interface basée sur le Web.

Commençons par un peu de théorie. L'impédance d'un composant électronique $Z = U/I$ est le rapport de la tension U aux bornes du composant par l'intensité I qui le traverse. La loi d'Ohm écrite sous la forme $R = U/I$ est un exemple particulier où la résistance R est la forme la plus simple d'une impédance. Pour un condensateur de valeur C, l'impédance dépend de la fréquence f dont la pulsation est $\omega = 2\pi f$, est donnée par $-i/\omega C$; l'unité imaginaire i est une forme compacte indiquant que la phase de la tension est décalée de 90° par rapport au courant. De même, l'impédance d'une inductance L est donnée par $i\omega L$. Ici encore, l'unité imaginaire i indique que la tension précède le courant. Nous pouvons donc déterminer l'impédance d'un composant en examinant la dépendance en fréquence de celle-ci et la relation de phase entre la tension et le courant qui le traverse.

Ces mesures deviennent intéressantes dans le cas de plusieurs composants interconnectés. Si une inductance et un condensateur sont reliés en parallèle, le circuit à l'impédance la plus élevée à la fréquence de résonance pour laquelle les résistances des deux composants en courant alternatif sont égales. Pour une connexion en série, l'impédance est la plus faible à ce point. Avec l'analyseur d'impédance basé sur un ESP32 d'Espressif, on peut enregistrer la courbe de l'impédance

en fonction de la fréquence d'un réseau RLC et ainsi déterminer ses caractéristiques.
Pour cela il est nécessaire de disposer d'un moyen de produire un signal sinusoïdal d'amplitude constante, de faire varier sa fréquence dans une plage préférablement importante, puis de mesurer rapidement le courant qui traverse le(s) composant(s) que l'on qualifie de DUT (*Device Under Test*). J'ai remarqué que le microcontrôleur ESP32 que j'avais reçu de la part d'Elektor pour ma participation au concours de design de 2018 pouvait réaliser la plupart de ces opérations. Il possède un générateur de fréquence intégré et un convertisseur Numérique Analogique (CNA ou DAC (*Digital Analog Converter*) qui génère des tensions de sortie dont la fréquence peut atteindre plusieurs centaines de kilohertz. De plus, il intègre un Convertisseur Analogique Numérique (CAN ou ADC *Analog Digital Converter*) qui peut lire des tensions à une rapidité pouvant atteindre un million de fois par seconde, certes en trichant un peu ; mais nous y parviendrons. L'ESP32 ne pouvant lire qu'un seul canal CAN à haute vitesse et les convertisseurs CAN et CNA fonctionnant indépendamment, on ne pourra déterminer que l'amplitude de l'impédance avec un seul ESP32, mais pas la phase. La possibilité de déterminer l'écart de phase produit par le circuit en test nécessite un dispositif spécial, ce sera le sujet d'un article séparé.

Circuit analogique d'entrée

Nous devons nous assurer que l'amplitude de la tension du CNA est constante et ne dépend pas de l'impédance du circuit en test. C'est le rôle de l'amplificateur opérationnel de la **figure 1**. Il est utilisé comme tampon de gain 1x pour le signal issu de la broche 25 de l'ESP32. Sa faible impédance de sortie pilote le DUT, circuit en test. L'autre terminaison du circuit en test est reliée à l'entrée négative d'un second amplificateur opérationnel qui est configuré en amplificateur à transimpédance. Un courant fourni à son entrée négative est converti, grâce à la résistance de contre-réaction R_{fb} en une tension de sortie qui est appliquée à la broche 32 du CAN de l'ESP32. Ce CAN n'accepte qu'une tension positive. Ainsi, la composante continue du signal est portée à la moitié de la valeur de la tension d'alimentation. Grâce à ce dispositif, le circuit en test est alimenté par une tension alternative d'amplitude constante ; le courant est mesuré à l'aide du CAN.
La **figure 2** montre l'interface d'entrée réalisée sur une plaque d'essais avec une petite platine perforée munie d'un condensateur comme circuit en test (DUT) en bas et à gauche. La résistance R_{fb} à droite de l'amplificateur opérationnel est facile à changer. Les quatre fils reliant ce petit circuit à l'ESP32 sont, en noir la masse GND, en rouge, la tension d'alimentation 3,3 V, en vert la sortie du CNA (broche 25), et en bleu la broche 32 du CAN.

CAN rapide

Le mode rapide du CAN de l'ESP32 fait partie du sous-système I2S qui est normalement utilisé pour le traitement des signaux audio. En outre, l'ESP32 supporte un mode dans lequel les mots générés par le CAN sont copiés dans une mémoire tampon à accès direct (DMA) qui ne nécessite pas l'utilisation du processeur. Le CAN fonctionne de façon asynchrone (librement) et la mémoire DMA collecte les données à un rythme déterminé. Tant que ce rythme est inférieur à celui de la rapidité maximum de conversion du CAN, soit environ 250 kC/s, toutes

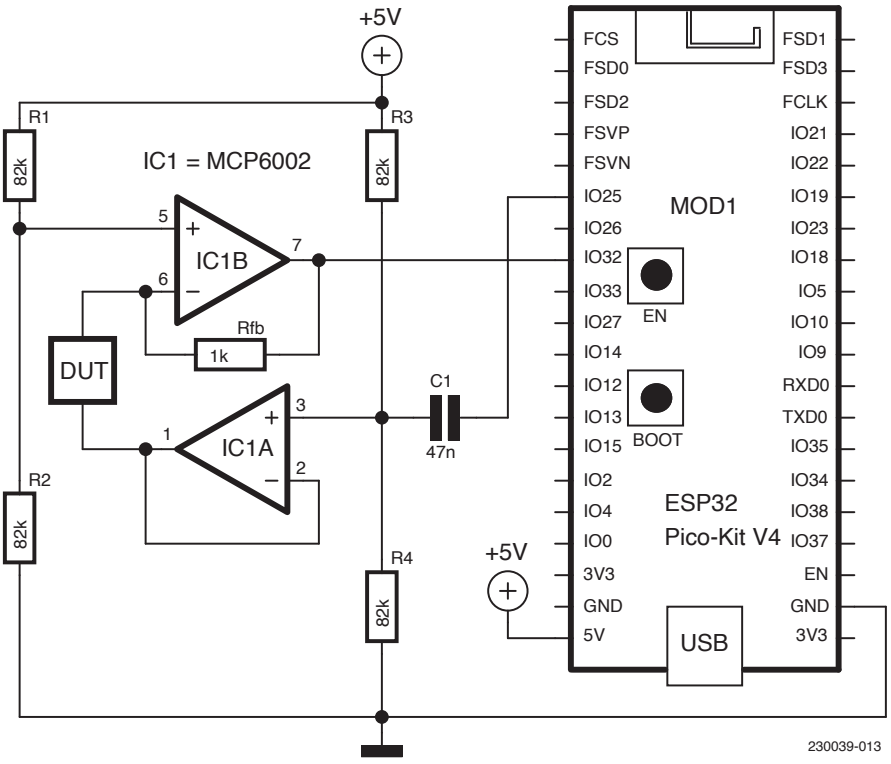


Figure 1. Interface analogique d'entrée.

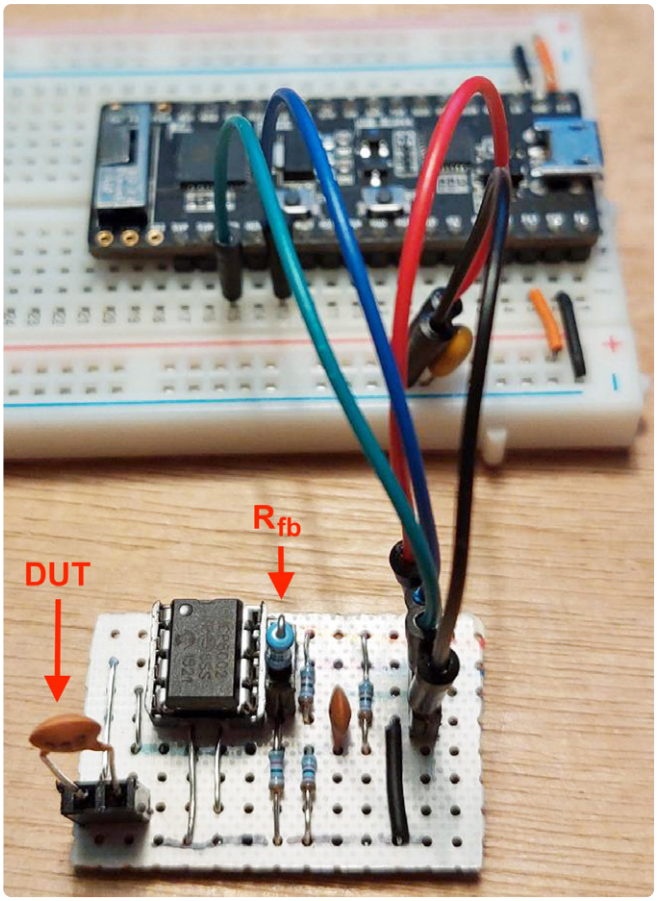


Figure 2. Seuls quelques composants externes sont nécessaires pour permettre à l'ESP32 de mesurer l'impédance d'un circuit en test (DUT).

les données sont « fraîches ». En revanche, si le taux est trop élevé, des échantillons identiques sont copiés dans la mémoire tampon, ce qui se traduit par des escaliers dans les données capturées.
La vitesse d'échantillonnage du CAN est paramétrée par la fonction `i2sinit()`, qui est dérivée du croquis *HiFreq_ADC.ino* présent dans la bibliothèque de support de l'ESP32 de l'EDI Arduino. La majeure partie de la configuration consiste à entrer dans la structure `i2s_config`, le mode d'opération, le taux d'échantillonnage et quelques paramètres indicateurs qui ont été trouvés après quelques recherches en [1]. Avant de quitter la fonction, on doit régler le niveau d'atténuation de façon à ce que l'amplitude du canal du CAN corresponde à la valeur de la tension d'alimentation de 3,3 V, choisir le canal du CAN et autoriser la sortie.

Génération de fréquence

L'ESP32 possède un bloc fonctionnel interne qui délivre les valeurs successives d'une fonction cosinus à un rythme qui peut être ajusté. La sortie de ce bloc peut être dirigée, en positionnant plusieurs bits de configuration, vers l'entrée du registre du CNA, ce qui produit une tension de sortie de forme sinusoïdale. La fréquence de ce générateur est directement contrôlée en paramétrant les registres `SENS_SAR_DAC_CTRL1_REG` et `SENS_SAR_DAC_CTRL2_REG`, qui sont décrits en [1] et plus en détail en [2]. En suivant les explications fournies en [2], la fonction `cwDACinit()` est paramétrée pour remplir ces registres avec les valeurs permettant de configurer la fréquence de sortie et l'amplitude. Après avoir inclus les fichiers d'en-tête qui permettent l'utilisation des noms mnémoniques, la fonction `SET_PERI_REG_MASK(reg,bits)` est utilisée pour définir les bits du registre `reg`. Par exemple, la première des commandes suivantes :

```
SET_PERI_REG_MASK(SENS_SAR_DAC_CTRL1_REG,
  SENS_SW_TONE_EN);
SET_PERI_REG_BITS(SENS_SAR_DAC_CTRL1_REG, SENS_SW_FSTEP,
  frequency_step, SENS_SW_FSTEP_S);
```

active le fonctionnement du générateur à haute fréquence interne. La seconde détermine la fréquence du signal de sortie en spécifiant l'entier `frequency_step`, dont la valeur est dérivée de la rapidité de l'horloge interne. La fréquence de sortie désirée est déterminée approximativement par l'équation figurant en [1].

Croquis ESP32 contrôlé par l'interface série

J'ai programmé l'ESP32 en utilisant la version 1.8.19 de l'EDI Arduino disponible en [3], et suivi les instructions d'installation disponibles. À la date d'écriture de cet article, la nouvelle version 2 de l'EDI ne supportait pas encore un supplément fonctionnel nécessaire par la suite. J'ai également dû installer les fonctions de support de l'ESP32 depuis [4] en suivant les instructions d'installation.
Les croquis suivants sont basés sur [5] où se trouve une discussion détaillée sur de multiples aspects. L'objectif est de contrôler la fréquence générée et l'acquisition des données à partir de l'interface série selon un protocole simple dans lequel une seule chaîne est échangée de façon bidirectionnelle entre l'ESP32 et un programme fonctionnant sur un ordinateur PC pouvant supporter une communication série ; cela peut être Python, Octave ou LabView. Ce protocole ressemble au protocole SCPI utilisé dans les oscilloscopes ou d'autres appareils de mesure, dans lequel un point d'interrogation final indique une commande qui attend une réponse de l'ESP32. Par exemple, l'envoi de `STATE?`, déclenche l'envoi de la réponse `STATE fmin fmax fstep`, dont les trois valeurs numériques indiquent la gamme et le pas de variation du balayage en fréquence.
Analysons maintenant les différentes parties du sketch. En premier, deux fichiers en-têtes sont inclus ; l'un pour les fonctions de support du CAN et du CNA que j'ai développées, l'autre pour l'accès au système de fichiers *SPIFFS* de la mémoire flash de l'ESP32. Ce dernier est utilisé pour mémoriser les données de calibration. Les valeurs par défaut de la gamme de fréquences balayées et quelques autres valeurs pertinentes sont également spécifiées. La fonction `setup()` du sketch initialise la communication série, fixe la fréquence de sortie et l'amplitude de sortie du CNA, et lit les données de calibration à partir des fichiers SPIFFS s'ils sont disponibles.
La fonction `loop()` vérifie si une demande de la part de l'ordinateur hôte est reçue par l'appel à `Serial.available()`, lit une ligne (terminées par le caractère retour à la ligne `\n`) et convertit ce qui est reçu dans la chaine de caractères `line`. Quelques vérifications sont alors faites, selon le contenu spécifique du début de la chaine. Par exemple, si `line` contient `FREQ 20000`, la valeur numérique est extraite par `atoi(&line[5])`. La conversion de la chaine commençant à la cinquième position de `line` en un entier, est effectuée par la fonction intégrée `atoi()`. Cette valeur est ensuite affectée à la variable `dac25freq` et initialise le générateur de fréquence par `cwDACinit()`. De façon identique, toutes les variables importantes sont accessibles depuis l'ordinateur hôte.
`SWEEP?` est la commande la plus intéressante. Après avoir reçu cette commande et initialisé les variables utilisées dans cette section, une boucle `for` incrémente la variable `f` de `fmin` à `fmax` avec un pas de

valeur `fstep`. À l'intérieur de la boucle, la valeur de la fréquence du CNA est tout d'abord définie, puis, après un court délai, les valeurs mesurées par le CAN sont récupérées à l'aide de `is2_read()`.

```
for (float f=freqmin; f<freqmax;f+=freqstep) {
 dac25freq=f;
 cwDACinit(dac25freq,dac25scale,0);
 delay(50);
 i2s_read(I2S_NUM_0, &buffer,
     sizeof(buffer), &bytes_read, 15);
...
}
```

La taille de son argument d'entrée `buffer`, définie au début du sketch, est utilisée pour déterminer le nombre d'échantillons, ici 1024. La fonction renvoie également la valeur `bytes_read` indiquant le nombre d'octets lus. Chaque échantillon contenant deux octets, nous devons diviser par deux lors du prélèvement des échantillons afin de déterminer les valeurs minimum et maximum. Rappelons-nous que la tension mesurée par le CAN est proportionnelle au courant qui traverse le circuit en test, nous utilisons donc les variations des valeurs de pointes pour déterminer l'amplitude du courant.

Calibration

En parcourant les échantillons, les sommes `q1` et `q2` sont également accumulées. Utilisées conjointement avec les variables `S0`, `S1` et `S2`, elles permettent de définir un segment de droite joignant les points des données, ces paramètres sont nécessaires à la calibration. Les détails de l'algorithme utilisé sont indiqués dans l'annexe B de [5]. La commande `SAVECALIB` sauvegarde ces paramètres dans la mémoire persistante SPIFFS. La commande `GETCALIB?` les récupère depuis le PC. Les constantes de calibration `calib_slope` et `calib_offset` sont nécessaires pour prendre en compte les variations de certains facteurs d'atténuation du système inconnus, par exemple, due au condensateur de couplage CA (transmission des tensions alternatives) de l'entrée de l'amplificateur opérationnel inférieur. Cette ambiguïté est résolue en calibrant le système à l'aide d'une résistance de valeur résistive connue, insérée comme DUT. Toutes les autres impédances sont alors déterminées en référence à cette résistance de calibration. Celle-ci devrait idéalement être insensible à la fréquence utilisée, mais de faibles variations systématiques sont prises en compte en utilisant la pente comme fonction de la fréquence, en plus de l'écart (offset). La valeur de la résistance de calibration devrait avoir sensiblement la même valeur que la résistance de contre-réaction de l'amplificateur à transimpédance afin de correspondre à la gamme opérationnelle du CAN.

Avant d'utiliser l'analyseur, le système de fichiers SPIFFS de l'ESP32 doit être initialisé en créant un répertoire vide nommé `data` contenu dans le répertoire où est stocké le sketch. Ensuite, le plugin de chargement du système de fichiers Arduino ESP32 doit être installé depuis [6] en suivant les instructions. Quand cela est fait, un nouveau choix *ESP32 sketch Data Upload* apparait dans le menu *Outils* de l'EDI Arduino. Après s'être assuré que le moniteur série est fermé, un clic sur ce menu va créer le système de fichiers de l'ESP32.

Contrôle par Octave ou Python

Le contrôle de l'acquisition de données depuis le PC nécessite l'ouverture d'un port série dont le nom diffère selon le système d'exploitation utilisé. Il est habituellement nommé `COMn` sous Windows, `/dev/tty.usbserial-n` avec un MAC ou `/dev/ttyUSBn` sous Linux. L'envoi d'une commande, par exemple pour définir la fréquence de départ du balayage, se fait simplement en envoyant `FMIN 10000` par le port série. De même, la lecture des valeurs des paramètres se fait en envoyant la commande `STATE?`, puis en attendant un court instant avant de lire les caractères qui sont retournés, jusqu'à la réception du caractère retour à la ligne (*line feed*) `\n`. La réponse contient `STATE fmin fmax fstep`. Le déclenchement du balayage de la fréquence est initialisé par l'envoi de la commande `SWEEP?` et la réception des valeurs se fait ensuite ligne par ligne. Lorsque toutes les données ont été mises à disposition, on peut fermer le port série et préparer le tracé des courbes.

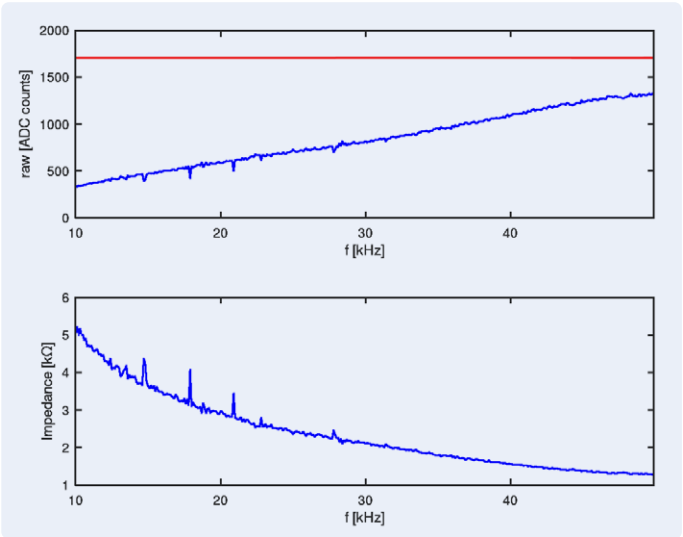


Figure 3. Valeurs brutes et impédance d'un condensateur utilisé en test.

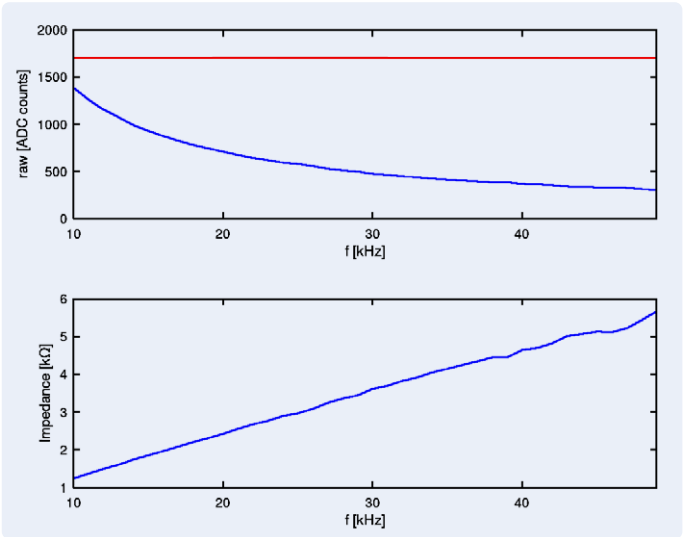


Figure 4. Données brutes et impédance dans le cas de test d'une inductance.

Les procédures scripts pour Octave sont incluses dans la *toolbox instrument* qui doit-être installée et pour Python dans le *Python-serial package* se trouvent dans l'archive logiciel accompagnant cet article. Notez que pour réaliser les tests préliminaires, un simple programme terminal tel que Putty peut être utilisé.

Utilisation du système

Pour commencer à utiliser le système, une résistance de calibration de 1 kΩ est utilisée comme composant de test et le balayage en fréquence est exécuté. Ensuite, la commande `SAVECALIB` qui est décrite en détail dans le script Octave `nwa.m` sauvegarde les données de calibration dans l'ESP32. Lorsque la commande `SAVECALIB` est exécutée, la résistance est remplacée par un condensateur de 2,2 nF, et `nva.m` est à nouveau exécuté, le tracé des données correctement calibrées de la **figure 3** apparait. Le tracé montre la dépendance inverse de la fréquence de laquelle on déduit la capacitance à l'aide de la commande Octave :

```
capacitance_nF=
     mean(1./(2*pi*Zabs*1e3.*xx*1e3))*1e9
```

Selon la formule :

$C = 1/(2\pi f Z_{abs})$

La moyenne de toutes les valeurs est réalisée par la fonction `mean()`. Les puissances de dix sont nécessaires pour prendre en considération le multiplicateur kilo de kΩ et de nano dans nF. Par exemple, `1e9` à la fin de la formule convertit la valeur du condensateur de Farad en nanoFarad.
Le tracé de la **figure 4** est obtenu en réitérant le balayage après avoir remplacé le condensateur by une inductance de 22 mH. Comme on s'y attendait, l'impédance de l'inductance sur le tracé inférieur augmente avec la fréquence, l'inductance peut être déterminée par la formule :

$L = Z_{abs}/2\pi f$

Ou obtenue en Octave par :

```
inductance_mH=
     mean(1e3*Zabs./(2*pi*xx*1e3))*1e3
```

qui réalise également la moyenne de tous les points obtenus, à l'aide de `mean()`. Les puissances de dix sont nécessaires pour tenir compte des multiplicateurs kilo de kΩ et khz et milli de mH.

Interface utilisateur Web

Jusqu'à présent, les mesures d'impédance ont été faites depuis Octave ou Python, nous allons maintenant utiliser un navigateur web pour contrôler et afficher les mesures. Pour cela, l'ESP32 est configuré afin d'exécuter un serveur web qui crée la page web de la **figure 5**. Quand le navigateur reçoit cette page web, un code javascript intégré ouvre un deuxième canal de communication série de l'ESP32, basé sur les websockets. Un websocket assume le rôle d'une communication série, il est ici utilisé pour assurer la transmission et la réception de messages. L'envoi de ces messages est déclenché par un clic sur les boutons situés en haut de la page web. Ils provoquent un balayage de la fréquence, l'effacement des courbes affichées et sauvegardent les données de calibration. Sur la ligne en-dessous, les paramètres de contrôle de l'ESP32 incluant la gamme du balayage, le taux d'échantillonnage du CAN peuvent être modifiés à partir des menus de sélection. Les boutons de la troisième ligne réalisent le calcul de la capacitance, de la résistance et de l'inductance, et en indiquent le résultat sur la ligne en-dessous des tracés. La ligne inférieure affiche les valeurs reçues de l'ESP32. Sur le tracé, l'axe horizontal indique la fréquence selon la gamme spécifiée dans les menus en haut de la page. L'axe vertical indique Z_{abs}, relativement à la résistance de calibration dont la valeur est affichée par la ligne horizontale bleue. En cliquant sur le tracé, les valeurs de la fréquence et de Z_{abs} apparaissent dans la ligne d'état.

Croquis de l'ESP32

Après avoir spécifié le nom du réseau wifi (SSID) et son mot-de-passe, les fichiers de support pour le wifi, websocket et webserver doivent être inclus, on déclare alors le serveur web `server2` afin de communiquer avec le réseau par le port 80, et par le port 81 pour le `websocket`. Les messages sous forme de texte échangés par les navigateurs sont formattés selon JSON, sous la forme `{"INFO":"Yada yada"}`. L'analyse grammaticale de ces messages et la génération de messages plus complexes sont réalisées par la bibliothèque *ArduinoJson*, tandis que le support du CNA et du CAN (ADC et DAC) est fourni par *ESP32_I2Sconfig.h* comme précédemment.

```
const char* ssid     = "YOUR_SSID";
const char* password = "YOUR_PASSWORD";
#include <WiFi.h>
#include <WebSocketsServer.h>
WebSocketsServer webSocket = WebSocketsServer(81);
#include <WebServer.h>
#include <SPIFFS.h>
WebServer server2(80);
#include <ArduinoJson.h>
#include "ESP32_I2Sconfig.h"
```

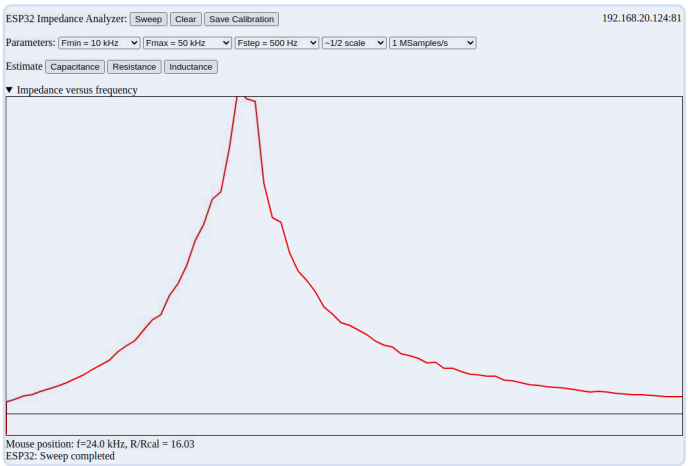


Figure 5. La page web générée par l'ESP32 montre la résonance obtenue sur la courbe d'impédance d'un condensateur de 2,2 nF, d'une inductance de 22 mH et d'une résistance de 33 kΩ connectés en parallèle.

Après avoir spécifié diverses variables, les fonctions de support sont définies, parmi lesquelles `webSocketEvent()` est la plus importante. Elle est appelée à chaque fois qu'un message envoyé par le navigateur arrive. S'il s'agit d'un message au format JSON, la fraction de code suivante extrait la commande `cmd` et la valeur `val` à l'aide des fonctions de la bibliothèque *ArduinoJson*. Selon la commande reçue, les actions appropriées sont alors exécutées. Par exemple, la réception de la commande `SWEEP` renverra des informations au navigateur par la fonction `sendMSG()` et positionnera la variable `mmode` à la valeur 1 qui est utilisée dans la boucle principale. `webSocketEvent()` étant appelée de façon asynchrone, elle interrompt les autres activités et doit être maintenue le plus court possible ; par conséquent, le balayage en fréquence est confié au programme principal. D'autre part, la définition de la fréquence de départ du balayage `freqmin` n'utilise que très peu de cycles de la CPU et se trouve dans la fonction `webSocketEvent()`.

```
DynamicJsonDocument root(300);
deserializeJson(root,payload);
const char *cmd = root["cmd"];
const long val = root["val"];
if (strstr(cmd,"SWEEP")) {
 sendMSG("INFO","ESP32: Received Sweep command");
 mmode=1;
} else if (strstr(cmd,"FMIN")) {
 freqmin=val;
 Serial.printf("FreqMin = %g\n",freqmin);
} else

...
```

Une suite de commandes similaires à celles utilisées précédemment est traitée de façon identique.
L'exemple de code suivant, extrait de la fonction `setup()`, connecte d'abord le réseau WLAN selon les paramètres fournis précédemment. Elle envoie des points sur la liaison série jusqu'à ce que cela réussisse, puis elle affiche l'adresse IP, démarre la communication avec le websocket, et paramètre `WebSocketEvent` de façon à traiter les messages provenant du websocket. Notez que la ligne de communication série n'est pas absolument nécessaire à ce stade, mais elle permet de mieux observer ce qui se passe dans l'ESP32, en particulier durant la phase de développement du système.

```
WiFi.begin(ssid, password);
while(WiFi.status() != WL_CONNECTED)
     {Serial.print("."); delay(500);}
Serial.print("\nConnected to ");
Serial.print(ssid);
Serial.print(" with IP address: ");
Serial.println(WiFi.localIP());
webSocket.begin();
webSocket.onEvent(webSocketEvent);
```

Ensuite, toujours dans `setup()`, on vérifie que le système de fichier SPIFFS a bien été créé, comme dans la première partie puis on en extrait les données de calibration. Cependant, à cet instant, le système SPIFFS contient également le fichier esp32_impedance_analyser.html qui décrit le contenu de la page web. Après avoir démarré `server2`, ce fichier HTML est utilisé par défaut lorsqu'un navigateur web se connecte comme indiqué par le premier argument dans le code suivant :

```
server2.serveStatic()
server2.begin();
server2.serveStatic("/",SPIFFS,
     "/esp32_impedance_analyzer.html");
```

La suite du code de la fonction `setup()` ressemble beaucoup au contenu du sketch précédent.
Dans la fonction `loop()`, ce qui concerne les serveurs http et websocket est tout d'abord traité, avant la vérification de la variable `info_available`. Cela est réalisé par `sendMSG()` qui informe que `info_buffer` contient un message au format JSON qui est traité immédiatement transmis au navigateur par un appel à `webSocket.sendTXT()`.

```
server2.handleClient();   // handle http server
webSocket.loop();         // handle websocket server
if (info_available==1) {
 info_available=0;
 webSocket.sendTXT(websock_num,
     info_buffer,strlen(info_buffer));
}
```

La valeur de `mmode` est ensuite vérifiée, et l'action appropriée est initialisée. Par exemple, `mmode==1` active le balayage de la fréquence par un code très similaire à celui précédemment utilisé. Dans ce cas, un message au format JSON nommé `WFO` (pour waveform ou forme d'onde) est créé et enregistré dans la variable `doc` par la commande :

```
doc["WF0"][nn-1]=floor((512/16)*Z);
```

dans laquelle la variable `nn` permet de reboucler sur toutes les valeurs de la fréquence et `Z` est la valeur absolue de l'impédance à cette fréquence. Notez que la variable est calibrée pour une couverture de 0 à 16 kΩ sur 512 pixels et ses valeurs sont converties en un entier par le fonction `floor()`. Lorsque la boucle est terminée, la courbe est envoyée au navigateur par :

```
serializeJson(doc,out);
webSocket.sendTXT(websock_num,out,strlen(out));
sendMSG("INFO","ESP32: Sweep completed");
```

et informe de la fin du balayage par `sendMSG()`. Les autres valeurs de `mmode` permettent de sauvegarder les données de calibration et envoient au navigateur les valeurs estimées du condensateur, de la résistance, ou de l'inductance.
Le fichier *esp32_impedance_analyser.html* décrit la page web et contient le code javascript qui la rend dynamique.

La page web

La plupart des pages web interactives utilisent les balises `<style>` pour décrire les aspects génériques de l'apparence des objets affichés par la page, suivies par des commandes HTML décrivant la page web et les instructions javascript qui assurent son interactivité.

Les instructions de style suivantes qui apparaissent au début du fichier *esp32_impedance_analyzer.html* définissent un cadre entourant la zone affichée et l'affichage de deux tracés rouge et noir. La dernière instruction permet l'affichage de l'adresse IP en haut et à droite de la page :

```
<style>
#displayarea { border: 1px solid black; }
#trace0 { fill: none; stroke: red; stroke-width: 2px;}
#trace1 { fill: none; stroke: black; stroke-width: 1px;}
#ip {float: right;}
</style>
```

La partie principale de la description de la page est insérée entre les balises `<BODY>` et `</BODY>`. Au début de cette section se trouvent la définition des boutons. La définition du premier bouton encadrée par des balises `button` est la suivante :

```
<button id="sweep" type="button"
     onclick="sweep();">Sweep</button>
```

`Sweep` est inscrit sur le bouton, un clic provoque l'exécution de la fonction `sweep()`. Ce type d'actions liées à un évènement sont baptisées fonctions de rappel. L'assignation de l'identificateur `sweep` au bouton permet de changer ultérieurement ses propriétés, par exemple le texte affiché ou l'action déclenchée. La définition de l'autre bouton suit les mêmes règles.
Le menu de sélection de la deuxième ligne utilise une syntaxe légèrement différente. La définition de ce menu est encadrée par des balises `SELECT`. Si l'une des entrées est choisie, elle appelle `setDataFreqMin(thisvalue)`, où `thisvalue` est la valeur spécifiée dans les différentes balises `OPTION`. Les balises `OPTGROUP` sont uniquement présentes pour agrémenter la clarté du code.

```
<SELECT onchange="setDacFreqMin(this.value);">
<OPTGROUP label="Sweep start frequency">
<OPTION value="1000">Fmin = 1 kHz</OPTION>
<OPTION value="2000">Fmin = 2 kHz</OPTION>
<OPTION value="5000">Fmin = 5 kHz</OPTION>
<OPTION selected="selected" value="10000">
    Fmin = 10 kHz</OPTION>
<OPTION value="20000">Fmin = 20 kHz</OPTION>
<OPTION value="50000">Fmin = 50 kHz</OPTION>
  </OPTGROUP>
</SELECT>
```

Pour terminer, la zone affichant l'impédance est de type SVG (*Scalable Vector Graphic* c'est-à-dire affichage graphique vectoriel adaptable), elle possède une taille de 1024 x 512 pixels et affiche deux courbes dont les identificateurs sont `trace0` et `trace1`. L'identificateur `id` permet leur modification ultérieure. La propriété d décrit la forme d'onde. L'initialisation est faite en déplaçant le curseur sur le pixel (0,200) avec `M0 200`. Par la suite, `d` est redéfinie par la courbe `WFO` en provenance de l'ESP32.

```
<svg id="displayarea" width="1024px" height="512px">
<path id="trace0" d="M0 200" />
<path id="trace1" d="M0 200" />
</svg>
```

À la fin, les deux lignes d'état sont définies par :

```
<div id="status">Status window</div>
<div id="reply">Reply from ESP32</div>
```

Elles sont identifiées par l'identificateur `id`, qui permet la modification ultérieure du texte affiché. De façon identique, la zone affichant l'adresse IP est identifiée par l'identificateur `ip` en haut à droite.

JavaScript

Toutes les commandes JavaScript sont insérées entre des balises `SCRIPT`. Après la balise d'ouverture, plusieurs variables sont définies, et l'adresse IP de l'ESP32 est déterminée par :

```
var ipaddr=location.hostname + ":81";
document.getElementById('ip').innerHTML=ipaddr;
```

Ici, le numéro de port du websocket est ajouté à l'ESP32 et le texte de la balise, identifié `ip`, est immédiatement mis à jour. L'espace alloué à l'affichage du texte est accédé à la deuxième ligne par une commande de construction assez longue, dans laquelle document se réfère à la page web elle-même. La partie suivant le point donne accès à l'élément nommé `ip` et `innerHTML` se réfère au texte affiché qui est alors modifié pour afficher `ipaddr`. Cette construction de commande est utilisée de façon extensive pour accéder à des éléments nommés et modifier leurs propriétés. Les fonctions `toStatus()` et `toReply()` suivent cette partie pour copier le texte dans les lignes d'état en bas de la page web.
Connaissant maintenant l'adresse du websocket de l'ESP32, ce dernier est ouvert par la commande new `WebSocket()` puis les fonctions de rappel sont ajoutées aux évènements `onopen`, `onclose` et quelques autres. Un court message est souvent envoyé à la console JavaScript par la commande `console.log ()`. La console est accessible par les outils de développement du navigateur, en utilisant par exemple, le raccourci clavier Ctrl-Shift dans Chrome.

```
var websock = new WebSocket('ws://' + ipaddr);
websock.onopen = function(evt)
  {console.log('websock open');};
```

La fonction de rappel la plus intéressante réagit à l'arrivée des messages provenant de l'ESP32. La fonction raccourcie `websock.onmessage()` analyse le message au format JSON dont le contenu est enregistré dans `event.data` et place la paire de valeurs de la commande dans la structure `stuff`. Si `stuff` contient la commande INFO, la valeur associée est mémorisée dans `val` et affichée sur la page web par `toReply()`. Si la commande est `WFO`, elle contient la courbe des données d'impédance. Le nombre de points transmis est déterminé par `val.length`. Pour s'y adapter, l'axe horizontal est mis à l'échelle afin d'utiliser la totalité des 1024 pixels. La propriété `d` du tracé est initialisée et les points y sont ajoutés, un pour chaque entrée de `val`. Le pixel correspondant à la valeur `0,0` étant en haut à gauche, la position verticale doit être inversée en affichant le pixel `512-val[i]`.

Pour terminer, la ligne de référence de calibration par la résistance, de couleur noire, est affichée.

```
websock.onmessage=function(event) {
var stuff=JSON.parse(event.data);
var val=stuff["INFO"]; // info
if (val != undefined) {toReply(val);}
var val=stuff["WF0"]; // waveform0
if (val != undefined) {
nstep=Math.floor(0.5+1024/val.length)
pixmax=nstep*val.length;
var d="M0 511";
for (i=0; i<val.length; i++)
{d += ' L' + (nstep*i) + ' ' + (512-val[i]);}
document.getElementById('trace0').setAttribute('d',d);
d="M0 480 L1024 480";
document.getElementById('trace1').setAttribute('d',d);
}
}
```

La suite du code JavaScript est principalement constituée de l'affectation des fonctions de rappel aux boutons menus. La fonction `sweep()`, déclenchée par le bouton correspondant est la suivante :

```
function sweep() {
websock.send(JSON.stringify
     ({ "cmd" : "SWEEP", "val" : -1 }));
}
```

La fonction `JSON.stringify()` encapsule ses arguments dans un message correctement formaté et `websocket.send()` le transmet à l'ESP32. Toutes les autres fonctions de rappel utilisent une structuration identique.
Juste avant la fin de la section JavaScript, `showCoordinates()` est définie pour afficher la fréquence et l'impédance correspondant au pixel du graphique dans la zone que l'on peut choisir en cliquant. Cette fonction répond à un évènement `mousedown` en attachant `addEventListener()` à `displayarea`:

```
document.getElementById("displayarea")
 .addEventListener('mousedown', showCoordinates, false);
```

Cette possibilité est très pratique pour déterminer directement les fréquences et les impédances correspondantes depuis l'affichage du tracé.

Sur la **figure 5**, on peut voir que la résonance d'un circuit parallèle RLC utilisé en test, indique un pic de résonance proche de la fréquence 24 kHz où cette impédance est supérieure à 16 kΩ.
Le microprogramme et les autres fichiers peuvent être téléchargés depuis [7]. L'analyseur d'impédance est maintenant autonome dans le sens où aucun programme de contrôle n'est nécessaire à son fonctionnement. Tous les échanges se font entre l'ESP32 et un navigateur, même dans le cas d'un smartphone.

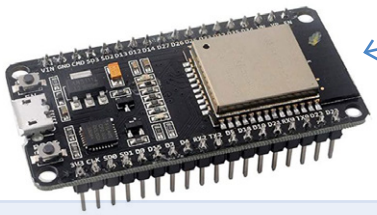
VF : Jean Boyer — 230039-04

Des questions, des commentaires ?

Contactez Elektor (redaction@elektor.fr).

À propos de l'auteur

L'intérêt de Volker Ziemanns pour l'électronique a commencé avec l'amplificateur Edwin de 40 W (Elektor au milieu des années 1970), mais il a suivi une orientation différente et étudié la physique, puis a travaillé sur des accélérateurs de particules – au SLAC aux États-Unis, au CERN à Genève et maintenant à Uppsala en Suède. L'électronique ayant un rôle primordial dans le contrôle et l'acquisition de données des accélérateurs, son intérêt précoce lui a été utile au cours de sa carrière. Il enseigne maintenant à l'Université d'Uppsala. Un de ses livres traitant de l'acquisition de données avec Arduino et Rapsberry Pi traite du sujet de cet article.



Produits

- **ESP32-DevKitC-32D (SKU 18701)**
  www.elektor.fr/18701
- **OWON HDS1021M-N oscilloscope à 1 voie (20 MHz) + multimètre (SKU 18778)**
  www.elektor.fr/18778

LIENS

[1] ESP32 Technical Reference Manual (Version 4.7), disponible à l'adresse : https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf
[2] Descritpion du générateur sinusoidal ESP32 : https://github.com/krzychb/dac-cosine
[3] Site web Arduino : https://www.arduino.cc
[4] Fonctions de support Arduino : https://github.com/espressif/arduino-esp32
[5] V. Ziemann, A Hands-on Course in Sensors Using the Arduino and Raspberry Pi, 2nd ed., CRC Press, Boca Raton, 2023:
[6] ESP32fs Plugin : https://github.com/me-no-dev/arduino-esp32fs-plugin
[7] Code source de ce project sur GitHub : https://tinyurl.com/4awvvemc