gestion du temps avec l'ESP32 et Toggl

Pratiquer le kit *ESP32 Basic Core* de M5Stack

Mathias Claußen (Elektor)

Il peut s'avérer utile de suivre les heures passées sur des projets électroniques. Différents services permettent de le faire. Nous utilisons ici Toggl.com pour vous montrer comment effectuer des requêtes web en HTTPS et comment mettre en œuvre les fonctions de base d'une interface graphique. Le tout est réalisé sur un kit de développement *ESP32 Basic Core* de M5Stack, une plate-forme de prototypage rapide pour les montages à base d'ESP32.

Figure 1. Le kit ESP32 Basic Core de M5Stack.

De nos jours, surtout avec le travail à domicile, le suivi de vos tâches et du temps que vous y consacrez peut vous aider à rester concentré. Cela vous permet aussi de discuter plus facilement de votre travail, que ce soit avec vos collègues ou vos clients. Il y a de nombreuses solutions sur le marché – comme openTimetool, Toggl et Kimai – qui proposent une gestion du temps et des projets et permettent aussi de produire les factures pour les clients. J'ai utilisé pour cet article le service de suivi en ligne Toggl, qui fournit également pour la gestion du temps, un greffon pour votre navigateur ou une appli pour votre téléphone. Son API ouverte permet de s'interfacer avec un système de suivi des temps, et de créer ainsi, avec ces services, votre propre application ou dispositif matériel pour suivre les temps.

KISS

« *Keep it simple stupid* » (KISS) : ne pas compliquer les choses. Pour un système de suivi des temps, c'est le mieux que vous puissiez offrir à un utilisateur. Pour certains d'entre nous, le suivi des temps est plus une corvée qu'un plaisir. Juste appuyer sur un bouton quand on se met au travail et sur un autre quand on s'arrête serait assez simple. Comme Toggl est un service en ligne, il nous faut quelque chose qui puisse se connecter à l'internet (avec ou sans fil) pour soumettre notre requête. L'ESP32 nous vient à l'esprit pour cette tâche. Pour éviter un fouillis de fils et de composants sur une platine d'expérimentation, nous suivons le principe KISS et nous prenons un kit de développement *ESP32 Basic Core* de M5Stack (**fig. 1**), dont la documentation est accessible en ligne [1]. En plus de l'EPS32, nous disposons de trois boutons, un écran, une batterie et quelques autres périphériques dans un joli boîtier compact. Il ne manque plus que quelques lignes de code.

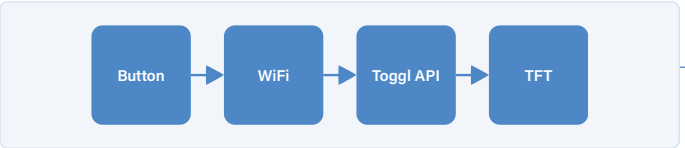

Figure 2. Flux de données vers le service Toggl.

Figure 3. Flux de données pour obtenir l'état actuel.

Idée de base

Il suffit d'un appareil qui puisse accéder au service web Toggl et réagir à au moins un bouton pour signaler la présence ou l'absence au travail. Avec le kit M5Stack, cela signifie utiliser le wifi pour la connexion à l'internet, puisqu'un point d'accès est à portée et que nous pouvons considérer être en ligne en permanence. Comme vous pouvez le voir sur la figure 1, nous avons le choix entre trois boutons. Et puisqu'il y a un écran LCD, pourquoi ne pas afficher quelques graphiques et indiquer l'état d'occupation (au travail ou absent du bureau) ? Est-ce difficile d'accéder à Toggl ? En théorie, pas trop, car l'API est bien documentée. En clair, il faut appuyer sur un bouton et envoyer une commande appropriée au service web Toggl. Cette commande est suivie d'un changement d'état qui sera affiché pour l'utilisateur. Cela paraît simple au départ. La **figure 2** montre le flux de données entre la pression d'un bouton et l'API Toggl. La **figure 3** montre le cheminement pour une mise à jour de l'écran LCD avec les données courantes, avec un flux vers le serveur Toggl, quelque part sur le réseau, et un flux de retour vers l'ESP32. C'est aussi simple que cela en a l'air.
La partie wifi a déjà été traitée plusieurs fois et nous récupérons le code des nombreux projets d'Elektor comprenant un ESP32. Ceci nous permet d'avoir un serveur web modulaire, un mécanisme intégré pour gérer le temps et les mises à jour OTA, c'est plus qu'il n'en faut. On utilise une pile de protocoles pour la communication avec le service Toggl. La **figure 4** montre cette pile et les bibliothèques concernées. La bibliothèque `WIFIClient` incluse dans le canevas Arduino ESP32 permet de se connecter à des serveurs accessibles par le réseau wifi auquel l'ESP32 est connecté. Au sommet, vous avez la bibliothèque `HTTPClient` qui gère le protocole et les requêtes HTTP, également inclus dans le canevas Arduino ESP32. Comme Toggl exige que les données soient échangées par HTTP au format JSON, nous avons besoin d'une bibliothèque supplémentaire pour gérer JSON : ce sera la bibliothèque `ArduinoJSON`.
Outre la communication, nous avons également besoin d'une bibliothèque pour piloter l'écran du kit M5Stack. `TFT_eSPI` est une bibliothèque bien connue qui fonctionne avec l'écran du kit M5Stack et qui fournira un large éventail de fonctions prêtes à l'emploi pour le dessin de graphiques. Comme elle est compatible, qu'elle fournit toutes les fonctions nécessaires pour dessiner du texte et des graphiques et qu'elle a été utilisée dans des projets antérieurs, ce serait dommage de s'en passer. Un bref aperçu de la pile graphique qu'offre `TFT_eSPI` est présenté à la **figure 5**.
Pour commencer, une brève introduction à plusieurs sujets sera utile. Je commencerai par le protocole HTTP, indispensable pour accéder au service Toggl.

HTTP

Le protocole *Hypertext Transfer Protocol* (HTTP), développé en 1989 au CERN, est utilisé par un client (par ex. votre navigateur web) pour demander une ressource et obtenir une réponse d'un serveur web. Par exemple, si vous voulez ouvrir www.elektor.com, votre client envoie une requête au serveur web. Pour ce faire, une connexion TCP/IP est établie et une chaîne de requête est envoyée comme indiqué dans le **listage 1**.

Listage 1. Requête HTTP

```
GET / HTTP/1.1
Host: www.elektor.com
User-Agent: curl/7.55.1
Accept: */*
```

La première ligne, un en-tête HTTP avec `GET`, indique au serveur que nous demandons quelque chose. Le « / » indique au serveur que nous voulons que la page web par défaut soit émise. Avec `HTTP/1.1`, nous disons au serveur que nous parlons dans la version 1.1 de HTTP. Le travail a commencé sur HTTP/3 (la troisième génération du protocole), mais pour la plupart de nos cas d'usage, HTTP/1.1 suffit.
Pour la deuxième ligne, l'hôte, il faut savoir que les noms que vous saisissez dans votre navigateur sont traduits par un service DNS en une adresse IP. Cela signifie que elektor.com sera traduit en 83.96.255.227, l'adresse IP utilisée pour la connexion TCP/IP. Derrière cette adresse IP peut se trouver un serveur web qui traitera non seulement elektor.com, mais aussi elektor.de ou elektor.nl. Pour permettre au serveur web d'identifier la page que vous voulez, la deuxième ligne avec `Host` est nécessaire. Avec `User-Agent` à la ligne 3, nous disons au serveur web quel type de client HTTP nous sommes, ici *curl*. Cela permet à un serveur web, par exemple, de présenter des pages optimisées pour votre navigateur web, car Safari se comporte différemment de Firefox. Le dernier en-tête HTTP de la ligne 4 indique à notre client d'accepter toutes sortes de contenus. Chaque requête HTTP se termine par une ligne vide.
À ce stade, le serveur web confectionnera une réponse à notre requête, en commen-

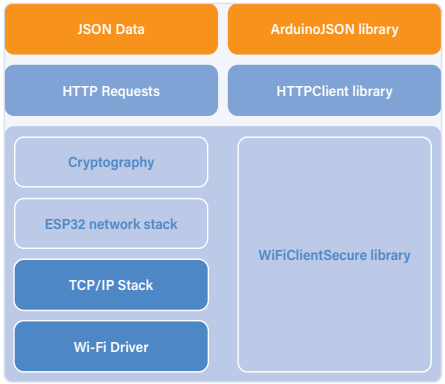

Figure 4. Pile de protocoles et bibliothèques.

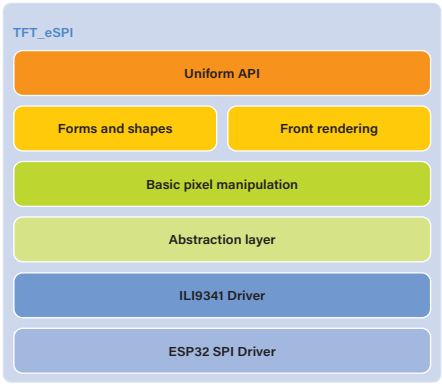

Figure 5. Fonctions de dessins empilés pour l'écran LCD.

Listage 2. Réponse HTTP

```
HTTP/1.1 200 OK
Server: unknown
Date: Tue, 17 Nov 2020 13:34:04 GMT
Content-Type: text/html; charset=UTF-8
Transfer-Encoding: chunked
Connection: keep-alive
X-Content-Type-Options: nosniff
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN
Strict-Transport-Security: max-age=63072000
Pragma: no-cache
Expires: -1
Cache-Control: no-store, no-cache, must-revalidate, max-age=0
Accept-Ranges: bytes
Strict-Transport-Security: max-age=63072000
```

indice	binaire	codé
0	000000	A
1	000001	B
2	000010	C
3	000011	D
4	000100	E
5	000101	F
6	000110	G
7	000111	H
8	001000	I
9	001001	J
10	001010	K
11	001011	L
12	001100	M
13	001101	N
14	001110	O
15	001111	P
16	010000	Q
17	010001	R
18	010010	S
19	010011	T
20	010100	U
21	010101	V
22	010110	W
23	010111	X
24	011000	Y
25	011001	Z
26	011010	a
27	011011	b
28	011100	c
29	011101	d
30	011110	e
31	011111	f
32	100000	g

indice	binaire	codé
33	100001	h
34	100010	i
35	100011	j
36	100100	k
37	100101	l
38	100110	m
39	100111	n
40	101000	o
41	101001	p
42	101010	q
43	101011	r
44	101100	s
45	101101	t
46	101110	u
47	101111	v
48	110000	w
49	110001	x
50	110010	y
51	110011	z
52	110100	0
53	110101	1
54	110110	2
55	110111	3
56	111000	4
57	111001	5
58	111010	6
59	111011	7
60	111100	8
61	111101	9
62	111110	+
63	111111	/
PADDING		=

Tableau 1. Tableau de codage en Base64

çant également par un ensemble d'en-têtes HTTP, comme indiqué dans le **listage 2**. La première ligne indique la version du protocole et un code de réponse. Comme nous avons demandé du HTTP/1.1, le serveur répondra également dans cette version du protocole. Le code de réponse que nous obtenons est `200`, signalant que notre requête peut être traitée. Cette ligne est suivie d'autres en-têtes qui donnent des informations supplémentaires.

Si nous sommes uniquement intéressés par le contenu demandé, nous pouvons sauter l'en-tête. L'en-tête est séparé du contenu demandé par une ligne vide (contenu ne figurant pas dans le listage 2). Pour votre information, jusqu'à 8 Ko de données, voire plus, peuvent se trouver juste dans l'en-tête lui-même. On peut définir le corps et sa longueur avec `Transfer-Encoding: chunked` pour les contenus de longueur inconnue, comme les pages web créées dynamiquement, ou `Content-Length:<length>` pour les contenus de taille connue à l'avance, comme les images.

Il existe d'autres méthodes HTTP comme le POST, où les données sont transférées du client au serveur, pour soumettre des informations avec un formulaire web. La méthode PUT peut également être utilisée pour transférer des données vers un serveur web. Pour plus d'informations sur les autres méthodes définies, consultez le « Mozilla Developer Network » [2].

De HTTP à HTTPS

Lors du développement de http, la sécurité intrinsèque du protocole de communication a été oubliée. Les informations sont transmises en texte clair et toute personne avec un peu de connaissance sur les réseaux peut les extraire ou les modifier. HTTPS, où le « S » signifie sécurisé, a rajouté une couche supplémentaire à la pile de communication. Le serveur et le client continuent à se parler en utilisant le protocole HTTP, mais au lieu d'avoir une communication en texte clair sur une connexion TCP/IP, tout le trafic entre les deux parties est d'abord traité par la couche *Transport Layer Security* (TLS) qui gère le chiffrement et l'échange de ses clés.

Il faudra aussi plus de puissance de calcul, de flash et de RAM si nous utilisons HTTPS, car il faut maintenant effectuer le chiffrement et le déchiffrement, et inclure dans notre logiciel les routines supplémentaires pour TLS. La puissance de calcul d'un ordinateur moderne suffira. Sur les

systèmes embarqués les plus simples, on observera des ralentissements dans la communication si le chiffrement et le déchiffrement, ainsi que la gestion et l'échange des clés, sont effectués uniquement dans le logiciel. C'est pourquoi, sur les UC modernes, vous trouverez des accélérateurs de chiffrement qui la déchargent de ce travail. Pour l'ESP32, on dispose d'une accélération matérielle pour effectuer rapidement ce processus.

Requête et réponse

Le service Toggl propose une API ouverte et documentée [3] pour l'échange de données. L'API fonctionne en soumettant des requêtes POST, PUT et GET à une URL, avec un en-tête et une charge utile définis. Toutes les actions nécessitent une clé d'authentification qui peut être obtenue sur la page web de Toggl. Grâce aux informations fournies, la construction d'une requête peut commencer. Nous utilisons la classe `WiFiClientSecure` en conjonction avec l'objet `HTTPClient`. C'est une méthode éprouvée pour communiquer avec le serveur et éviter de réinventer la roue lorsqu'il s'agit de traiter des données HTTP. Nous constituons à la main le jeton d'autorisation, une combinaison codée en Base64 du nom d'utilisateur et du mot de passe. Pour le codage en Base64, il faut une clé API et la chaîne `:api_token`. On peut obtenir la clé API auprès du service Toggl. Comme elle ne sera pas envoyée en texte clair, il peut y avoir des caractères qui interfèrent avec les caractères réservés pour l'en-tête lui-même. Pour inclure tout type de données dans ces requêtes, la solution est un encodage en Base64. D'habitude, les valeurs d'un octet sont comprises entre 0 et 255. La Base64 n'utilisera que 64 valeurs mises en correspondance avec les lettres ASCII équivalentes `A-Z`, `a-z`, `0-9`, `+`, `/` et `=` pour le remplissage (**tableau 1**). Par exemple, si nous encodons Elektor comme une chaîne de caractères en Base64, nous obtiendrons RWxla3Rvcg== (**fig. 6**). Sachez que les pièces jointes des courriers électroniques sont encodées en Base64 pour des raisons similaires, et comme le montre l'exemple, la taille de la pièce jointe augmentera d'environ 37% par rapport à la taille originale.
Une fois les informations d'identification transmises au client HTTP pour autorisation, la requête peut maintenant être faite. L'exemple suivant montre comment mettre en œuvre une requête GET et traiter les informations renvoyées. L'extrait du

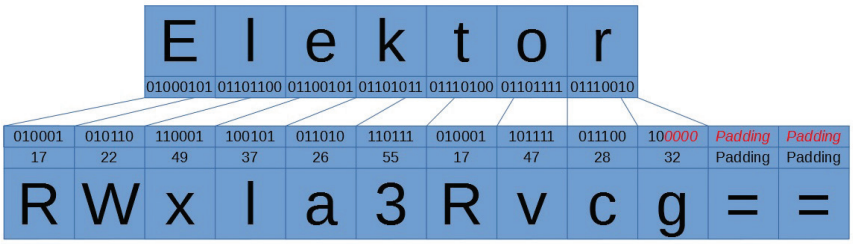

Figure 6. Elektor codé en Base64.

listage 3 vous donnera quelques indications. La fonction `GET()` renverra le code de réponse du serveur pour notre requête, ou si nous avons un problème de communication, une valeur négative. On peut voir en [4] les différentes valeurs possibles du code de réponse. Le code le plus connu est 404, *Resource not found*. Ici, le code 200, *Response okay*, est attendu et vérifié. Si des données sont reçues du serveur, elles peuvent être récupérées avec `getString()` pour un traitement ultérieur.
Il existe un code similaire pour PUT et POST, avec quelques différences mineures, mais importantes, comme le montre le **listage 4**. La fonction `addHeader()` utilisée avant `PUT()` ajoutera deux en-têtes supplémentaires à la requête. Le premier est *Content-Type* qui indique au serveur web le type de données que nous lui enverrons, le second est *Content-length*, et c'est ici que HTTP devient soudainement un peu différent de ce que nous avons vu auparavant. D'habitude, nous transmettons après l'en-tête des données, comme du texte, qui commencent et se terminent par deux sauts de ligne. Cela fonctionne pour le texte, mais si nous envoyons maintenant une image

Listage 3. Extrait de la requête HTTP

```
httpclient.begin(client,link);
   httpclient.setReuse(true);
   httpclient.setAuthorization((const char*)b64.c_str());
   int httpCode = httpclient.GET();
    if (httpCode > 0) { //Check for the returning code
        if(httpCode!=200){
            Serial.println("Response Error");
            String payload = httpclient.getString();
            Serial.println(httpCode);
            Serial.println(payload);
            *error = 1;
        } else{
            *Result = httpclient.getString();
        }
    } else {
        Serial.print("Request Error");
        Serial.println(httpclient.errorToString(httpCode));
    }
    httpclient.end();
```

Listage 4. En-tête HTTP additionnel

```
httpclient.setAuthorization((const◦char*)b64.c_str());
   httpclient.addHeader("Content-Type","application/json");
   httpclient.addHeader("Content-length","0");
   int httpCode = httpclient.PUT((uint8_t*)(NULL),0);
      if (httpCode > 0) { //Check for the returning code

      . . .
```

Listage 5. Réutilisation de la connexion

```
httpclient.begin(client,link);
httpclient.setReuse(true);
httpclient.setAuthorization((const char*)b64.c_str());
httpclient.addHeader("Content-Type","application/json");
int httpCode = httpclient.POST(payload);
   if (httpCode > 0) {

. . .
```

Listage 6. Exemple de chaîne JSON

```
{ "name": "Test", "value": 1351824120,"array": [ 123.45, 321.89 ] }
```

Listage 7. Sérialisation JSON

```
DynamicJsonDocument doc(capacity);
JsonObject time_entry = doc.createNestedObject("time_entry");
   if(description.length()>0){
       time_entry["description"] = description.c_str();
   }
time_entry["created_with"] = "ESP32";
serializeJson(doc, payload);
```

Listage 8. Désérialisation JSON

```
DeserializationError er = deserializeJson(doc, Result.c_str());
 if (er) {
      //something went wrong
         } else {
         if(false == doc["data"].isNull() ){
             JsonObject data = doc["data"];
             uint32_t data_id = data["id"];
             const char* data_start = data["start"];
            if(false == data["description"].isNull()){
                 const char* data_description = data["description"];
                  Element->description = String(data_description);
             } else {
                 Element->description = "";
             }
```

Listage 9. Chaîne JSON Toggl

```
{
"data":
   {
           "id":436694100,
           "pid":123,
           "wid":777,
           "billable":false,
           "start":"2013-03-05T07:58:58.000Z",
           "duration":1200,
           "description":"Brew some coffee",
           "tags":["billed"]
   }
}
```

JPEG au serveur, elle peut être constituée de n'importe quelle combinaison d'octets arbitraires. C'est là que *Content-length* entre en jeu. Les en-têtes sont disposés et séparés de nos données par deux sauts de ligne. Avec *Content-length*, vous pouvez envoyer n'importe quel type de caractère ou de données binaires. Le serveur web a été informé du nombre d'octets qu'il doit s'attendre à recevoir avec *Content-length*. Pour POST, il suffit de fournir le type de contenu qui sera transféré. Aucune charge utile réelle ne sera transmise. Si nous devions fournir une charge utile ou un contenu supérieur à 0 octet, le client HTTP ajouterait un en-tête *Content-Length* de son propre chef. Dans ce cas particulier, avec une longueur fixée à zéro, puisque nous n'envoyons pas de contenu, on n'ajoute pas du tout l'en-tête *Content-Length*. Comme le serveur se plaindra s'il manque la longueur du contenu, nous devons l'ajouter nous-mêmes avec la fonction `addHeader()`, comme indiqué dans le listage 4. On peut voir dans le **listage 5** que `setReuse()` est réglé sur *true*. Après cette requête, que ce soit POST, PUT ou GET, la connexion au serveur sera normalement fermée par le client. Si `setReuse()` est réglé sur *true*, cette connexion restera ouverte et sera réutilisée pour la requête suivante. On gagne du temps, car on évite de rétablir la connexion s'il faut effectuer une autre requête. En outre, au moment de la rédaction de ce document, cela contourne un bogue quelque part dans la pile réseau, où la mémoire n'est pas correctement libérée. Tôt ou tard, cela peut entraîner des échecs de connexion, car l'ESP32 n'a plus de mémoire disponible.

Échange de données avec JSON

JSON signifie *JavaScript Object Notation* et décrit un format d'échange de petites quantités de données entre deux systèmes. Toutes les données sont regroupées dans une chaîne de caractères qui peut ressembler à celle du **listage 6**. Cette chaîne doit être constituée et analysée. La plupart des environnements qui traitent et échangent des données pour des applications web utilisent JSON et disposent de fonctions intégrées ou de bibliothèques supplémentaires toutes faites pour la constitution et l'analyse de données au format JSON. C'est aussi vrai pour les petits systèmes comme un Arduino ou un ESP32.
`ArduinoJSON` est une bibliothèque compatible avec le canevas Arduino et l'ESP32.

Elle peut analyser et constituer des chaînes JSON à échanger avec d'autres systèmes. Comme Toggl utilise JSON pour échanger des données, cette bibliothèque est adaptée à nos besoins. L'API de Toggl nous permet de déterminer les informations qui devraient se trouver à l'intérieur des chaînes JSON. Cette documentation décrit également comment organiser les données pour envoyer des informations. Pour une nouvelle saisie de temps, nous devons créer une chaîne JSON qui contient un objet `time_entry`. À l'intérieur de cet objet `time_entry`, nous avons besoin d'une chaîne `description` et d'une chaîne optionnelle `created_with` pour la nouvelle entrée.

Le code du **listage 7** va créer un nouveau document `DynamicJsonDocument` avec une capacité définie dans la pile de l'ESP32. Cela signifie que la mémoire est allouée avec `malloc()` puis libérée avec `free()` à partir de la mémoire non utilisée statiquement par l'ESP32. Par opposition à `StaticJsonDocument` qui sera alloué de manière statique dans la RAM ou, s'il est utilisé localement dans une fonction, sera placé dans la pile. `StaticJsonDocument` est plus rapide, car localement lié à la pile, mais sa taille est inférieure à celle qu'on peut utiliser avec `DynamicJsonDocument`. La fonction `serializeJson()` créera la chaîne de caractères souhaitée qui pourra être envoyée.

`deserializeJson()` permet de décoder les données JSON et signalera toute erreur rencontrée lors de l'analyse. Cette partie du code est présentée dans le **listage 8**. On y voit comment accéder aux éléments dans la chaîne JSON. La chaîne de caractères renvoyée par Toggl ressemble à celle du **listage 9**. Le code vérifiera après « désérialisation » si les données (`data`) de l'objet existent, c'est-à-dire si nous avons un résultat valide. Ensuite, le code accède aux éléments à l'intérieur de `data` et ne récupère que les valeurs nécessaires plus tard pour l'affichage et le traitement, à savoir `id`, `start` et `description`. Ce dernier champ est spécial, il n'est intégré dans `data` que s'il contient une valeur de la base de données Toggl. Cela nécessite de vérifier avec `data[«description»].isNull()` que l'entrée existe bien avant de tenter d'y accéder ; sinon, la bibliothèque provoquera une exception.

L'échange de données complet est encapsulé dans la bibliothèque `toggleClient`. Ici, elle prend en charge le strict nécessaire : afficher l'entrée courante, démarrer une nouvelle entrée et terminer l'entrée courante. Le code n'est pas parfait et chacun est invité à contribuer aux améliorations et aux corrections d'erreurs, car il s'agit plus d'un point de départ que d'une bibliothèque finalisée. Outre la collecte et la soumission de données, il faut aussi les afficher, et c'est l'objet de ce qui suit.

Dessiner des pixels

Pour le dessin, vous vous souvenez peut-être d'un projet d'Elektor plus ancien, « GUI tactiles » [5], qui utilisait la bibliothèque `TFT_eSPI`, compatible avec diverses combinaisons d'écrans et d'ESP32. Comme cette bibliothèque prend en charge l'écran du kit M5Stack, on pourra réutiliser du code provenant d'autres projets. Dans le dossier de la bibliothèque, on doit modifier le fichier *User_Setup_Select.h* pour utiliser la configuration du kit M5Stack. Dans le fichier TogglButtonM5.ino, la ligne `TFT_eSPI tft = TFT_eSPI() ;` créera un nouvel objet TFT_eSPI qui servira ultérieurement pour le dessin.

Pour initialiser l'affichage, nous n'avons besoin que de quatre lignes de code à l'intérieur de la fonction `setup()` (**listage 10**). On doit régler `inverted display` sur *true* pour transmettre les couleurs dans le bon ordre pour l'écran du kit M5Stack. Avec `setRotation(1)`, le bas de l'écran sera là où se trouvent les boutons du kit M5Stack. Après cette ligne de code, on peut utiliser toutes les manipulations de pixels possibles qu'offre la bibliothèque.

Une petite astuce pour accélérer le tracé, en particulier pour un afficheur basé sur l'ILI9341. Ces afficheurs sont optimisés pour traiter des images en plein écran ou de plus gros volumes de données. L'accès pixel par pixel ralentira le dessin d'au moins sept fois, voire plus. Ceci est dû au mode d'émission des commandes et d'accès aux pixels individuels. Déjà optimisée pour le dessin avec l'ESP32, la bibliothèque `TFT_eSPI` optimisera, dans la mesure du possible, les temps d'accès. Outre le placement de pixels ou le remplissage de l'écran avec une seule couleur, la bibliothèque fournira aussi d'autres fonctions de dessin comme les lignes, les rectangles et l'utilisation des fonctions `print` pour imprimer des chaînes de caractères sur l'écran. Si on vous demande pourquoi utiliser une bibliothèque pour ça au lieu d'écrire un pilote, vous pouvez répondre : ne jamais réinventer la roue.

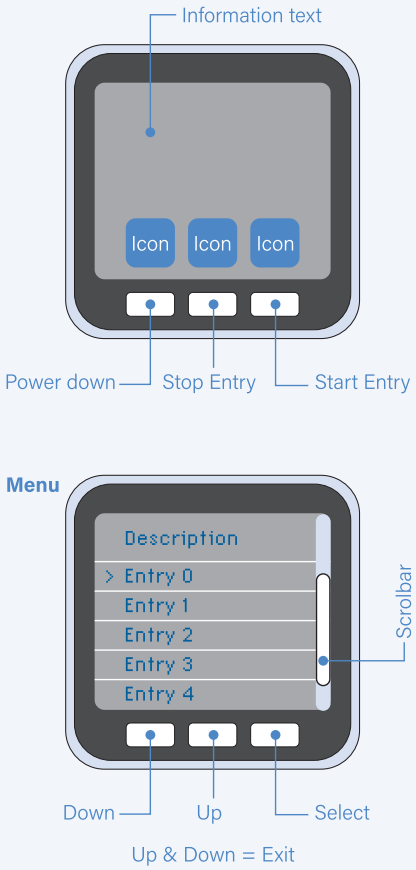

Figure 7. Schéma conceptuel de la GUI.

Listage 10. Initialisation de l'écran LCD

```
tft.init();
tft.invertDisplay( true );
tft.setRotation(1);
tft.fillScreen(TFT_WHITE);
```

Interface graphique (GUI) et menu

Le kit M5Stack dispose d'un écran et de trois boutons. Les solutions de suivi des temps ne fournissent généralement qu'un seul bouton pour démarrer et terminer une entrée. Comme le kit M5Stack peut aussi fonctionner sur batterie, il peut être très pratique d'avoir un bouton d'arrêt. La **figure 7** montre le principe de l'interface graphique et les positions approximatives des quelques éléments à afficher.

Comme il est assez simple de dessiner du texte, la question se pose : qu'allons-nous afficher ? Les éléments essentiels à afficher sont la description, si nous en avons une, et l'heure de début. Pour les icônes, nous utiliserons celles de la bibliothèque Open Icon [6] et on peut voir l'écran principal qui

Figure 8. Menu principal de la GUI.

en résulte sur la **figure 8**. Il nous permet de démarrer une nouvelle entrée, de l'arrêter et de mettre l'appareil hors tension. L'appareil reste ainsi facile à utiliser.

Nous pouvons démarrer une nouvelle entrée Toggl, mais il n'y a pas de description de l'activité. On n'a pas besoin de plus, mais ce serait bien de pouvoir au moins choisir parmi quelques activités prédéfinies. Même s'il est possible de saisir la description des activités avec seulement trois boutons, ce n'est pas très convivial. Une refonte totale basée sur le projet d'horloge à trois affichages [7] a permis d'introduire un système de menu simple à utiliser avec la bibliothèque `TFT_eSPI`. La résolution de l'écran et l'orientation du système de menu correspondent également à notre configuration du bouton Toggl. On peut récupérer le code de ce projet pour présenter un menu comme celui de la **figure 9**.

Le menu n'est pas aussi universel et flexible que celui d'un canevas pour GUI. Il est léger, fonctionne avec `TFT_eSPI` et a été conçu pour être utilisé avec seulement quelques boutons. Si j'avais eu plus de temps, j'aurais utilisé la bibliothèque LVGL (*Light and Versatile Graphics Library*) [8], car elle fonctionne également pour les écrans non tactiles. Voyons comment fonctionne notre système de menu fait maison et comment les graphiques sont dessinés. La logique du menu réside dans une fonction `RenderMenu()`. Selon l'état, elle dessine le menu et retourne *true*. Si

Figure 9. Sélection de l'activité dans la GUI.

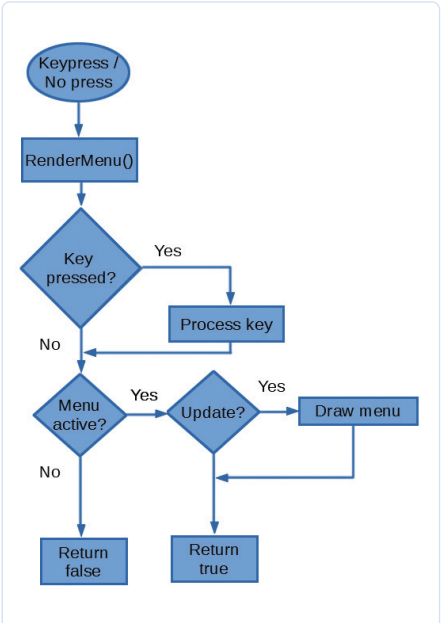

Figure 10. Organigramme du système de menu.

Listage 11. ShowMenuSettingsList

```
void Menus::ShowMenuSettingsList(uint8_t selected_idx, bool refresh){

  /* Draw Headline */
  if(true == refresh){
    _lcd->fillRect(0,0,320,40,_lcd->color565( 45,47,50 ) );
    _lcd->setTextColor(_lcd->color565( 112,116,122 ),TFT_BLACK);
    _lcd->setFreeFont(FSB18);
    _lcd->setCursor(160- ( _lcd->textWidth("Description") / 2) ,30);
    _lcd->print("Description");
  }
  /* Headline done */

  /* Draw scrollbar */
  _lcd->fillRect(303,43,15,197,TFT_WHITE);
  _lcd->drawRect(300,40,20,200,_lcd->color565( 112,116,122 ));
  _lcd->drawRect(301,41,18,198,_lcd->color565( 112,116,122 ));
  _lcd->drawRect(302,42,16,196,_lcd->color565( 112,116,122 ));
  _lcd->fillRect(305,45 +(196/GetUsedEntryCount())*selected_idx
   ,10,(196/GetUsedEntryCount())-2 , TFT_DARKGREY);
  /* Scrollbar done */

  /* Menu entries */
  uint8_t startidx=0;
  uint8_t endindex=0;
  if(selected_idx>4){ //We can display 5 Elements
    startidx = selected_idx -4 ;
  }
  if((startidx+5)>GetUsedEntryCount()){ //Limit last drawn item
      endindex=GetUsedEntryCount();
  }else{
      endindex=startidx+5;
  }
  /*Draw up to five elements*/
  for(uint8_t i=startidx;i<endindex;i++){
          DrawSettingsMenuEntry( (40*(i-startidx))
   ,DescriptionArray[i].c_str() ,( i==selected_idx ) );
  }
  /* Menu entries drawn */
}
```

aucun menu n'est dessiné, la fonction retournera false, ce qui indique que le contenu de l'affichage n'a pas été modifié. Si *true* est renvoyé, il faut éviter de faire un autre tracé par-dessus. Voir l'organigramme simplifié de la **figure 10**.

La logique du menu sera assurée par `RenderMenu()`, mais la partie dessin sera faite avec `ShowMenuSettingsList()`. Cette fonction dessinera l'entête du menu et jusqu'à cinq items de menu. La logique du menu garantit que l'élément sélectionné reste dans une plage valide. Ainsi, la routine de dessin calcule la position de départ et affiche cinq items sous forme de liste à l'écran. En outre, une barre de défilement, comme dans de nombreux autres systèmes Windows, sera calculée et dessinée avec un petit indicateur de la position dans la liste.

Le **listage 11** montre le code de `ShowMenuSettingsList()`, structuré en trois parties. La première étant le titre. Pour gagner du temps de traitement, elle n'est redessinée que lorsque nécessaire.

Le centrage du texte du titre se fait manuellement en calculant la moitié de la longueur du texte en pixels et en soustrayant la moitié de la largeur de l'écran. Cela fonctionne si le texte fait moins de 160 pixels et peut produire des résultats imprévisibles s'il est plus long. La seconde partie est la barre de défilement, dessinée à droite du menu. Elle se compose de quelques rectangles. Sa longueur est calculée à partir du nombre d'éléments utilisés dans le menu. De même, cela fonctionne s'il y a moins de 196 items dans la liste, sinon les résultats peuvent être imprévisibles.

Dans la troisième partie, on calcule les indices de début et de fin des items de liste à dessiner. Comme nous ne pouvons dessiner que cinq éléments, nous devons déplacer cette plage en conséquence par rapport à l'indice sélectionné par l'utilisateur. Les trois parties se chargent de la liste affichée. Comme tous les items de la liste se ressemblent, chacun est dessiné avec sa propre fonction qui n'a besoin que d'un décalage pour dessiner à la bonne hauteur et d'un drapeau si l'item est sélectionné. On obtient un système de menu sommaire qui remplit sa fonction et qui illustre quelques notions de base des menus.

Configuration à partir du Web

Le logiciel n'a pas été écrit à partir de zéro. J'ai réutilisé des parties de projets antérieurs. C'est aussi vrai pour la configuration à partir du web. Si vous appuyez sur les deux boutons de gauche pendant le démarrage, le logiciel démarre un point d'accès auquel vous pouvez vous connecter avec un ordinateur ou un appareil portable compatible wifi. Sinon, le logiciel essaiera de se connecter au réseau wifi qui a été configuré, le cas échéant. Il est facile de démarrer un serveur web sur l'ESP32 : après le démarrage du wifi le logiciel nous présente une interface web, qui nous permet de configurer le réseau wifi (**fig. 11**) et d'activer une clé d'API Toggl pour l'authentification de l'utilisateur (**fig. 12**).

Prolongements et enseignements

Bien que le bouton Toggl fonctionne et permette un efficace suivi des temps, il est encore possible de l'améliorer. Dans cet article, je n'ai abordé que quelques-uns des nombreux aspects des dispositifs IdO modernes, en particulier les interac-

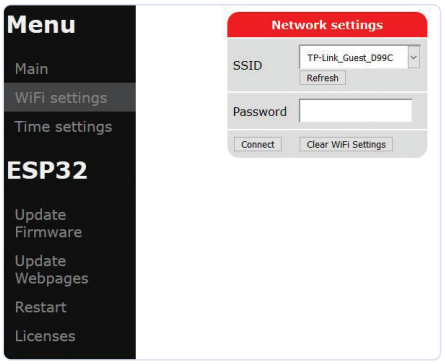

Figure 11. Page de configuration du wifi.

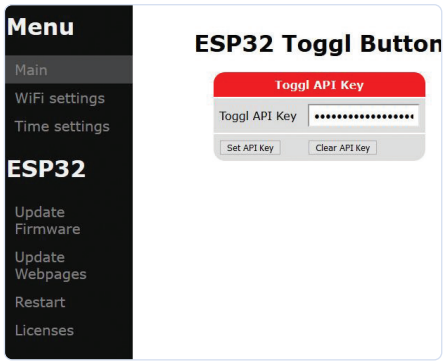

Figure 12. Clé d'API de Toggl.

tions avec tout type de serveur ou de navigateur web. Des termes tels que wifi, serveur HTTP, client HTTP, TLS, JSON, JavaScript, HTML, GUI et C/C++ peuvent sembler être des mots à la mode, mais ils sont en réalité les ingrédients essentiels de ce modeste projet. Le fait de s'occuper de HTTP et HTTPS, JSON d'un côté et de la conception de l'interface graphique (GUI) de l'autre, tout en fournissant une interface de configuration à partir du web, était peu courant aux premiers temps de l'IdO. Mais aujourd'hui, c'est le strict minimum pour un appareil connecté. Ce mélange de différents domaines – dont le développement web de base, JSON, HTML, C/C++ et JavaScript – pourrait remplir un livre entier. Et si un tel livre est publié, Elektor en rendra compte.

En écrivant cet article, j'ai trouvé quelques idées pour rendre le bouton encore plus convivial et ajouter des fonctions qui amélioreront la comptabilité. La première amélioration serait de traiter la GUI avec LVGL. En parlant de la GUI, passons rapidement aux enseignements. Si vous voulez ajouter une GUI à votre projet, cherchez quelque chose qui a fait ses preuves et qui est disponible. Il y a peut-être même des solutions avec une licence amiable pour votre projet. Développer votre propre système de menus pour le plaisir d'apprendre peut être génial, mais réinventer la roue ne l'est pas.

Pour l'ESP32, si vous utilisez le canevas et les bibliothèques Arduino, pensez à consulter son dépôt GitHub et les anomalies signalées. C'est une bonne ressource pour éviter des problèmes ou contourner des limitations connues. On trouve le code de ce bouton Toggl sur la page GitHub d'Elektor [9] ce qui est un moyen pratique de le cloner et de l'examiner, car son évolution y est conservée. Si vous trouvez des bogues ou avez des suggestions, vous pouvez utiliser la fonction issues de GitHub. Comme ce projet touche de nombreux sujets, j'attends vos retours pour savoir s'il faudrait revenir plus en détail sur certains d'entre eux dans un prochain article.

(200631-04)

PRODUITS

- **Kit de développement *ESP32 Basic Core* de M5Stack**
  www.elektor.fr/m5stack-esp32-basic-core-development-kit
- **Kit de développement ESP32 Arduino MPU9250 de M5Stack**
  www.elektor.com/m5stack-esp32-arduino-mpu9250-development-kit
- **W. Gay, *FreeRTOS pour ESP32-Arduino*, Elektor 2020 (livre en anglais)**
  www.elektor.fr/freertos-for-esp32-arduino

Contributeurs
Développement, texte et diagrammes : **Mathias Claußen**
Illustrations : **Patrick Wielders**
Mise en page : **Giel Dols**
Traduction : **Denis Lafourcade**

Des questions, des commentaires ?
Si vous avez des questions ou des commentaires sur son article, envoyez un courriel à l'auteur (mathias.claussen@elektor.com) ou contactez Elektor (redaction@elektor.fr).

LIENS

[1] **M5Core Basic** : https://docs.m5stack.com/#/en/core/basic
[2] **HTTP | MDN** : https://developer.mozilla.org/en-US/docs/Web/HTTP
[3] **Documentation de l'API de Toggl.com** : https://github.com/toggl/toggl_api_docs
[4] **Codes de retour d'état HTTP | MDN** : https://developer.mozilla.org/en-US/docs/Web/HTTP/Status
[5] **« GUI tactiles – pour ESP32, RPi & Co. », Elektor** : www.elektormagazine.fr/news/guitactilespouresp32rpico
[6] **Bibliothèque Open Icon** : https://sourceforge.net/projects/openiconlibrary/
[7] **Horloge à 3 affichages avec écran TFT** : https://bit.ly/37KBhJC
[8] **Bibliothèque Light and Versatile Graphics** : https://lvgl.io/
[9] **Page GitHub du projet** : https://github.com/ElektorLabs/200631_ESP32_Toggl_Button