Rapport de Synthèse
N°75
Objet connecté à base de microcontrôleur ARM

Auteurs : CHEN Muyao & YE Jing
Encadrant : ABIB Idir Ghalid

Table des matières

Introduction

Les applications embarquées et les objets connectés nécessitent souvent une connexion sans fil type WiFi ou Bluetooth afin d'interagir avec un système distant. Ceci permet de récupérer des données de capteurs via une connexion directe ou via des points d'accès.

L'enjeu principal de l'Internet des objets de nos jours est l'interopérabilité entre différents protocoles de communication. Ce projet est dédié au développement d'un point d'accès pour gérer à la fois la liaison Internet (WiFi) et la communication avec les capteurs (Bluetooth), qui permettra aux utilisateurs de facilement contrôler leurs objets connectés via une seule application (Android).

Pour des raisons de faible coût et de faible consommation électrique, certains systèmes embarqués intègrent un microcontrôleur de type ARM et disposent même d'un contrôleur WiFi Intégré. Vu la mémoire (Programme et données) disponible dans un microcontrôleur, l'optimisation du programme principal et des bibliothèques TCP/IP (Ethernet) ou Bluetooth est cruciale. En outre, le développement de l'application Android nécessite une bonne connaissance de la programmation orientée objet, et aussi un savoir-faire du génie logiciel.

Ce projet est divisé en deux parties. Les objectifs de la partie dite "Hardware" sont d'une part, la mise en place d'un point d'accès (Gateway) WiFi à l'aide d'un microcontrôleur ARM Cortex M4 de Texas Instruments permettant la communication avec un smartphone; et d'autre part, l'ajout à ce même microcontrôleur d'une liaison Bluetooth afin de communiquer avec des objets connectés tels que des capteurs. Pour la partie "logicielle", une application Android sera développée pour établir la communication entre le smartphone et le point d'accès via WiFi. Cette application a pour but de recevoir les données, de les visualiser et de contrôler des objets tel qu'un interrupteur de lumière.

Partie Hardware

Matériel utilisé

Afin de créer une passerelle IoT qui peut à la fois gérer plusieurs connexions en Bluetooth et la connexion Ethernet, nous avons choisi un microcontrôleur ARM cortex M4 (TIVA C TM4C1294) proposé par Texas Instruments.

Par rapport aux périphériques, nous avons utilisé deux cartes Arduino UNO, une carte Wemos D1 (ESP8266 Wi-Fi intégré) et deux modules DHT11 (détection)

L'IDE utilisé pour le développement en C est CCS7.3.0 (Code Composer Studio) de TI.

Explication des programmes

"Project" en détail

Le projet CCS comprend des fichiers nécessaires pour programmer les fonctionnalités sur la carte. D'abord, il existe trois répertoires qui s'appellent "Binaires", "Includes" et "Debug". Ces deux répertoires n'existent qu'après avoir construit (Build) le projet. Dans le répertoire "Binaires", on trouve le fichier généré (compilé et construit) par l'IDE pour enfin programmer le microcontrôleur. Le répertoire "Includes" contient toutes les librairies utilisées par nos programmes. Dans le répertoire "Debug", il y a tous les fichiers intermédiaires, y compris le Makefile.

Ensuite, nous trouvons tous les fichiers pour construire notre projet. Dans "drivers", le fichier "pinout.c" initie tous les ports GPIO par déclarer leurs types et aussi les configure selon nos besoins. Dans "utils", le fichier "ustdlib.c" offre des fonctions pratiques et standardise afin de faciliter la programmation. Le fichier "uartstdio.c" nous fournit les fonctions utilisant UART0 pour envoyer les textes qu'on veut afficher sur l'écran d'ordinateur par la liaison série. Les deux autres fichiers C concernent la gestion de LwIP. La partie TCP sera expliquée par la suite.

Puis, “enet_lwip.c” contient la fonction “main” du programme et aussi quelques sous-fonctions. Le fichier “tcp_server.c” et “tcp_server.h” sont des fonctions pour la gestion du protocole TCP. Le fichier “lwipopts.h” définit les options pour utiliser la pile LwIP. Le fichier “startup_css.c” est pour démarrer (bootloader) la carte pour qu’il puisse charger la fonction “main” et ainsi les autres fichiers dépendants. Enfin, les fichiers restants sont nécessaires pour le compilateur.

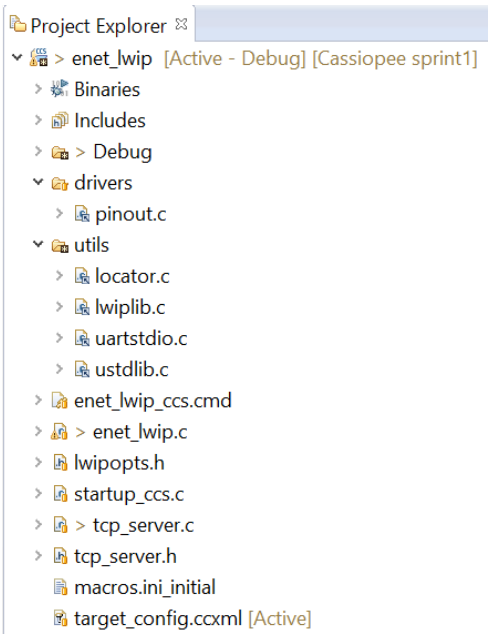

Fig 1 Fichiers du projet

Fonction “Main.c”

Nous aborderons dans cette partie le principe de la fonction “main”. L’explication détaillée du code est commentée dans le programme.

La fonction “Main” contient trois étapes. La première phase et celle d’initialisation, où on initialise la fréquence de microcontrôleur, la configuration des broches, les liaisons UART, et l’EEPROM.

```
630 //*****************************************************************************
631 //
632 // Main fonction
633 //
634 //*****************************************************************************
635 int
636 main(void)
637 {
638     uint32_t ui32User0, ui32User1;
639     uint8_t pui8MACArray[8];
640
641     //
642     // Make sure the main oscillator is enabled because this is required by
643     // the PHY.  The system must have a 25MHz crystal attached to the OSC
644     // pins. The SYSCTL_MOSC_HIGHFREQ parameter is used when the crystal
645     // frequency is 10MHz or higher.
646     //
647     SysCtlMOSCConfigSet(SYSCTL_MOSC_HIGHFREQ);
648
649     //
650     // Run from the PLL at 120 MHz.
651     //
652     g_ui32SysClock = MAP_SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ |
653                                              SYSCTL_OSC_MAIN |
654                                              SYSCTL_USE_PLL |
655                                              SYSCTL_CFG_VCO_480), 120000000);
656     //
657     // Configure the device pins.
658     //
659     PinoutSet(true, false);
```

Fig 2 Initialisation

La deuxième phase concerne les premières manipulations essentielles. On configure le module Ethernet intégré sur la carte pour d'abord se connecter à Ethernet et obtenir une adresse IP locale et ensuite mettre en place un serveur TCP à l'aide des fonctions spécifiques dont je parlerais dans la partie suivante. En plus, on ajoute les adresses des deux appareils Bluetooth dans la liste des capteurs Bluetooth pour les futures interrogations.

```
752     UARTprintf("Waiting for IP.\n");
753
754     //
755     // Convert the 24/24 split MAC address from NV ram into a 32/16 split MAC
756     // address needed to program the hardware registers, then program the MAC
757     // address into the Ethernet Controller registers.
758     //
759     pui8MACArray[0] = ((ui32User0 >>  0) & 0xff);
760     pui8MACArray[1] = ((ui32User0 >>  8) & 0xff);
761     pui8MACArray[2] = ((ui32User0 >> 16) & 0xff);
762     pui8MACArray[3] = ((ui32User1 >>  0) & 0xff);
763     pui8MACArray[4] = ((ui32User1 >>  8) & 0xff);
764     pui8MACArray[5] = ((ui32User1 >> 16) & 0xff);
765
766     //
767     // Initialize the lwIP library, using DHCP.
768     //
769     lwIPInit(g_ui32SysClock, pui8MACArray, 0, 0, 0, IPADDR_USE_DHCP);
770     //
771     // Setup the device locator service.
772     //
773     LocatorInit();
774     LocatorMACAddrSet(pui8MACArray);
775     LocatorAppTitleSet("EK-TM4C1294XL enet_io");
776
777
778     //
779     // Initialize a tcp server
780     //
781     tcp_server_test();
```

Fig 3 Connexion Ethernet

Dans la boucle infinie (while(1)), on essaie de se connecter avec chaque capteur Bluetooth en changeant le pair du module Bluetooth lié à la carte maître. En envoyant un

message de demande, on récupère ensuite dans l'interruption d'URAT7 la donnée renvoyée par le capteur connecté. Toute la communication sur Internet sera traitée dans l'interruption d'Ethernet.

```
797     //
798
799     addBluetooth(1, "2016,11,282119",14);
800     addBluetooth(2, "2016,11,295802",14);
801
802
803     while(1)
804     {
805
806         if(consultBluetooth(1)){
807             int delay_t = 30;
808             while(delay_t--){
809                             SysCtlDelay(g_ui32SysClock / 3); // delay 1s
810                             UARTprintf(".");
811                         }
812             if(!RequestBluetooth()) UARTprintf("\n No Response from Bluetooth 1 \n");
813         }
814
815
816         if(consultBluetooth(2)){
817             int delay_t = 30;
818             while(delay_t--){
819                             SysCtlDelay(g_ui32SysClock / 3); // delay 1s
820                             UARTprintf(".");
821                         }
822             if(!RequestBluetooth()) UARTprintf("\n No Response from Bluetooth 2\n");
823         }
824
825     }
826 }
```

Fig 4 Boucle infinie

Gestion du protocole TCP

Il existe plusieurs manières pour mettre en place la communication en protocole TCP/IP, y compris les sockets, LwIP (Light-Weight IP), et uIP(Micro IP). La programmation en sockets est facile à manipuler, mais elle demande un système d'exploitation (RTOS). Vu que notre programme n'est pas trop complexe, et qu'en plus, nous voulons réduire la consommation du système embarqué, nous avons donc choisi d'intégrer LwIP, une pile de protocole TCP dédiée aux systèmes embarqués.

La communication Ethernet s'établit en deux phases, celle d'initialisation et celle de traitement. Dans l'initialisation, on crée d'abord un identifiant (sous forme d'une structure) pour s'identifier dans le réseau local. Au niveau du réseau, on définit principalement notre adresse MAC et la manière d'adressage, soit par le protocole DHCP, ou par une adresse IP statique.

Après l'initialisation, on arrive à la phase critique du projet, où on créé un serveur TCP pour échanger de données. Afin d'établir un serveur, il faut d'abord créer un tcp_pcb (protocol control block) pour mettre la configuration du serveur (adresse IP locale, Port

local, adresse IP du client, Port du client, etc.), et demander une allocation de mémoire qu'on va utiliser comme un buffer de données à envoyer et à recevoir. En utilisant la fonction "tcp_bind()", on peut relier notre PCB avec un port spécifique. Ensuite, on accepte la connexion venant d'un client.

```
//TCP Server test
void tcp_server_test(void)
{
    err_t err;
    struct tcp_pcb *tcppcbnew;
    struct tcp_pcb *tcppcbconn;

    uint8_t res=0;
    tcppcbnew=tcp_new();
    if(tcppcbnew)
    {
        err=tcp_bind(tcppcbnew,IP_ADDR_ANY,TCP_SERVER_PORT);
        if(err==ERR_OK)
        {
            tcppcbconn=tcp_listen(tcppcbnew);
            tcp_accept(tcppcbconn,tcp_server_accept);
        }else res=1;
    }else res=1;

    if(res == 1){
        tcp_server_connection_close(tcppcbnew,0);
        tcp_server_connection_close(tcppcbconn,0);
        memset(tcppcbnew,0,sizeof(struct tcp_pcb));
        memset(tcppcbconn,0,sizeof(struct tcp_pcb));
    }

}
```

Fig 5 tcp_server_test

Les traitements qu'on effectue lorsqu'une demande de connexion arrive sont décrits dans la fonction "tcp_server_accept()". En connexion, on va mettre à jour les informations du client et aussi l'état de connexion. Nous relions aussi les fonctions qu'on a créées pour manipuler les données avec les modèles d'handler tcp proposés par LwIP.

```
65 err_t tcp_server_accept(void *arg,struct tcp_pcb *newpcb,err_t err)
66 {
67     err_t ret_err;
68     struct tcp_server_struct *es;
69     LWIP_UNUSED_ARG(arg);
70     LWIP_UNUSED_ARG(err);
71     tcp_setprio(newpcb,TCP_PRIO_MIN);
72     es=(struct tcp_server_struct*)mem_malloc(sizeof(struct tcp_server_struct));
73     if(es!=NULL)
74     {
75         es->state=ES_TCPSERVER_ACCEPTED;
76         es->pcb=newpcb;
77         es->p=NULL;
78
79         tcp_arg(newpcb,es);
80         tcp_recv(newpcb,tcp_server_recv);
81         tcp_err(newpcb,tcp_server_error);
82         tcp_poll(newpcb,tcp_server_poll,1);
83         tcp_sent(newpcb,tcp_server_sent);
84
85         UARTprintf("\nPhone connected\n");
86         //tcp_server_flag|=1<<5;
87
88         ret_err=ERR_OK;
89     }else ret_err=ERR_MEM;
90     return ret_err;
91 }
```

Fig 6 tcp_server_accept()

En recevant une donnée, nous allons la récupérer de la mémoire du paquet TCP et la mettre dans un buffer. On change l'état d'un drapeau signifiant qu'il y a une nouvelle donnée à traiter.

```
if(p!=NULL)
{
    memset(tcp_server_recvbuf,0,TCP_SERVER_RX_BUFSIZE);
    for(q=p;q!=NULL;q=q->next)
    {
        if(q->len > (TCP_SERVER_RX_BUFSIZE-data_len))
            MEMCPY(tcp_server_recvbuf+data_len,q->payload,(TCP_SERVER_RX_BUFSIZE-data_len));
        else MEMCPY(tcp_server_recvbuf+data_len,q->payload,q->len);
        data_len += q->len;
        if(data_len > TCP_SERVER_RX_BUFSIZE) break;
    }
    UARTprintf("%s",tcp_server_recvbuf);
    if(tcp_server_recvbuf[0] == '1')
        receive_flag = 1;

    tcp_recved(tpcb,p->tot_len);
    pbuf_free(p);
    //free(tcp_server_sendbuf);
    ret_err=ERR_OK;
}
```

Fig 7 tcp_server_recv()

Dans une autre fonction "tcp_server_poll()", nous testons le drapeau et s'il y a une nouvelle donnée, on va la comparer avec "1". Si c'est "1", que l'on a reçu, on va envoyer toutes les données enregistrées dans EEPROM et la vider. Enfin, on remet le drapeau à 0. Cette fonction va être appelée régulièrement (chaque 2ms) pour vérifier qu'on n'omet pas une demande.

```
if(receive_flag == 1){
    UARTprintf("\nSuper! I have received the request for data\n");

    static uint32_t eepromAddr;
    EEPROMRead(&eepromAddr,0,4);
    if(eepromAddr == 0){
        char* str_nodata = "No Data";
        strcpy(tcp_server_sendbuf,str_nodata);
    } else {
        *tcp_server_sendbuf = '\0';
        if(eepromAddr != 4294967295)
        {
            int var = 0;
            uint32_t temp;
            char bufferConvert[10];
            for (var = 4; var <= eepromAddr; var+=4) {
                EEPROMRead(&temp,var,4);
                sprintf(bufferConvert,"%d",temp);
                UARTprintf("%s",bufferConvert);
                strcat(tcp_server_sendbuf,bufferConvert);
                strcat(tcp_server_sendbuf,",");
            }
            EEPROMMassErase();
            uint32_t zero = 0;
            EEPROMProgram(&zero, 0, 4);
            EEPROMRead(&eepromAddr,0,4);
        }
    }

    es->p=pbuf_alloc(PBUF_TRANSPORT,strlen((char*)tcp_server_sendbuf),PBUF_POOL);
    pbuf_take(es->p,(char*)tcp_server_sendbuf,strlen((char*)tcp_server_sendbuf));
    tcp_server_senddata(tpcb,es);
    if(es->p!=NULL)pbuf_free(es->p);

    receive_flag = 0;
```

Fig 8 tcp_server_poll

Pour envoyer des données, il faut juste mettre des données dans le packet TCP, et appliquer la fonction "tcp_sndbuf()" fournie par LwIP.

```
215 void tcp_server_senddata(struct tcp_pcb *tpcb, struct tcp_server_struct *es)
216 {
217     struct pbuf *ptr;
218     uint16_t plen;
219     err_t wr_err=ERR_OK;
220      while((wr_err==ERR_OK)&&es->p&&(es->p->len<=tcp_sndbuf(tpcb)))
221      {
222         ptr=es->p;
223         wr_err=tcp_write(tpcb,ptr->payload,ptr->len,1);
224         if(wr_err==ERR_OK)
225         {
226             plen=ptr->len;
227             es->p=ptr->next;
228             if(es->p)pbuf_ref(es->p);
229             pbuf_free(ptr);
230             tcp_recved(tpcb,plen);
231         }else if(wr_err==ERR_MEM)es->p=ptr;
232      }
233 }
```

Fig 9 tcp_server_senddata

Finalement, on ferme la session quand le client se déconnecte et attendre une nouvelle connexion.

Gestion des connexions Bluetooth

Comme nous l'avons expliqué, les demandes venant de l'application Android vont être traitées dans l'interruption Ethernet. La communication entre la passerelle et les capteurs Bluetooth se fait dans la boucle infinie dans la fonction "main"

Nous connectons un Module HC05 avec la carte TI, et configurons ce module par commandes AT via UART (liaison série). On change entre le mode AT et le monde transmission en contrôlant la broche "EN" sur module. Si on met "EN" à "1", puis alimenter le module, il va entrer dans le mode AT. Si on désactive "EN" et redémarre le module, il va passer au mode communication. C'est de cette manière qu'on peut connecter ce module avec plusieurs capteurs Bluetooth.

Le capteur bluetooth s'identifie par une adresse Hardware unique. Ainsi, on sauvegarde les adresses du capteur dans un tableau.

```
//*****************************************************************************
//
// Add Bluetooth device into the device list
//
//*****************************************************************************
void
addBluetooth(int id,char* addr,int size){
    strncpy(Bluetooth_list[id-1].addr,addr,size);
    Bluetooth_list[id-1].id = id;
}
```

Fig 10 addBluetooth

Dans la fonction “consultBluetooth()”, on coupe la connexion Bluetooth actuelle et entre dans mode AT. Ensuite, on envoie des commandes pour redéfinir le capteur à connecter. Enfin, on redémarre le module pour se connecter avec ce capteur.

```
466 //*****************************************************************************
467 //
468 // Configure the Bluetooth module to connect with different bluetooth devices
469 // If configuration succeeds return 1
470 //
471 //*****************************************************************************
472 int
473 consultBluetooth(int id){
474
475     //
476     // Turn off the Bluetooth
477     //
478     GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_6, 0); // Key
479     GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_7, 0); // VCC
480
481     SysCtlDelay(g_ui32SysClock / (100 * 3)); // delay 10ms
482
483     flagMode = 0;
484
485     //
486     // Enter the AT Mode
487     //
488     GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_6, GPIO_PIN_6);
489     SysCtlDelay(g_ui32SysClock / (1000 * 3)); // delay 1ms
490     GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_7, GPIO_PIN_7); // VCC
491
492     int t = 1000;
493     char *BL_test = "AT";
494     UARTSend(BL_test, strlen(BL_test));
495     while(!flagcmd && t--){
496         SysCtlDelay(g_ui32SysClock / (1000 * 3)); // delay 1ms
497     }
498     if(!flagcmd){
499         UARTSend(BL_test, strlen(BL_test));
500         t=1000;
```

Fig 11 consultBluetooth

Finalement, dans la fonction “RequestBluetooth()”, on envoie une demande (“1”) à un capteur Bluetooth. On attend une seconde puis vérifier s’il y a une nouvelle donnée stockée dans EEPROM.

```
609 //*****************************************************************************
610 //
611 // Send the request message ("1") to the Bluetooth device so that
612 // the target device would send back the data
613 // Received data will be received in the BTIntHandler fonction
614 // Here we wait for several time and check if we have received the data (return 1 or 0)
615 //
616 //*****************************************************************************
617 int RequestBluetooth(){
618     uint32_t AddrToRead = 0;
619     EEPROMRead(&AddrToRead,0,4);
620     UARTSend("1",2);
621     UARTprintf("Message Requiring Data Sent\n");
622     int t = 1000;
623     while(t--) SysCtlDelay(g_ui32SysClock / (1000 * 3));
624     uint32_t NewAddrToRead = 0;
625     EEPROMRead(&NewAddrToRead,0,4);
626     if(AddrToRead == NewAddrToRead) return NULL;
627     return 1;
628 }
```

Fig 12 RequestBluetooth

Gestion de l'enregistrement de donnée

Pour gérer les données récupérées des capteurs Bluetooth, nous avons décidé d'utiliser EEPROM. L'intérêt de cette mémoire est qu'on ne perd pas de données même si la carte n'est plus alimentée. Car l'espace de la mémoire est limité, on vide la mémoire quand on envoie tous les données à l'application Android.

```
//
// Save the received data in the EEPROM
//
EEPROMRead(&eepromAddr,0,4);    // Find the current eeprom address
if(eepromAddr == 4294967295) {  // If overflow, begin from the 0 address
    eepromAddr = 0;
}
EEPROMProgram(&BLInService,eepromAddr+4,4); // Save the current device ID being interrogated
eepromAddr+=4;
EEPROMProgram(&data,eepromAddr+4,4);        // Save the Integer into the next address
eepromAddr+=4;
EEPROMProgram(&eepromAddr, 0, 4);           // Update the current address

//
// Display the data information on the screen (for debugging)
//
uint32_t temp;
EEPROMRead(&temp,eepromAddr,4);
UARTprintf("%d ",temp);
```

Fig 13 EEPROM

Périphériques

Afin de démontrer notre système IoT, nous avons aussi programmé des périphériques, y compris deux capteurs Bluetooth et un actionneur. Nous avons utilisé deux cartes Arduino UNO et deux modules DHT11 pour détecter la température et l'humidité. Ces deux cartes sont aussi liées avec un module HC05. On a programmé une carte WEMOS D1 qui intègre un module ESP8266 pour illustrer que notre application arrive aussi à contrôler un actionneur via la liaison Wi-Fi. Une LED est donc liée avec cette carte.

Partie Android

Introduction

Cette application sert à réaliser la détection de l'environnement intérieur et le contrôle des lumières en utilisant la communication par les sockets avec un serveur TCP.

IDE

Les ressources logicielles sont :

Android Studio 3.0.1, com.github.markushi:circlebutton:1.1,

com.github.jd-alexander:android-flat-button:v1.1,

com.github.zcweng:switch-button:0.0.3@aar, CSVWriter, hellocharts-library-1.5.8

Explication des programmes

Structure du programme

Notre application dispose de 2 fonctions : lecture de la température et de l'humidité; et le contrôle de lumière. Le fichier MainActivity.java correspond à la page principale de l'application. Les deux fonctions ont besoin la connexion TCP pour communiquer avec la partie Hardware grâce à tcpClientSender.

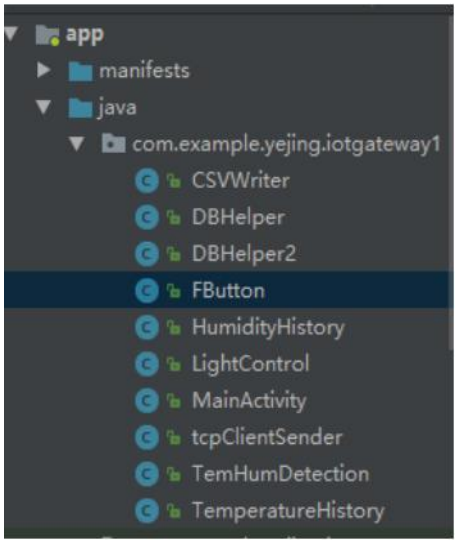

Fig 13 Page principale

La fonction "Détection de la température et de l'humidité" est réalisée par TemhumDetection.java. Il faut saisir l'adresse IP de la carte à laquelle on se connecte pour commencer la communication. Dès que c'est fait, le bouton "Synchronize" permet d'obtenir les données de la température et de l'humidité stockées dans la carte en envoyant un "1". Elles sont sauvegardées dans les bases de données correspondantes grâce à la classe DBHelper. Les deux boutons "Temperature" et "Humidity" sont respectivement liés avec TempratureHistory et HumidityHistory, par lesquelles on peut voir les données sous forme d'un graphe (20 dernières données) avec l'instant où la

valeur est détectée et d'un tableau qui présente toutes les données. Elles peuvent être exportées en fichier .CSV en utilisant CSVWriter.

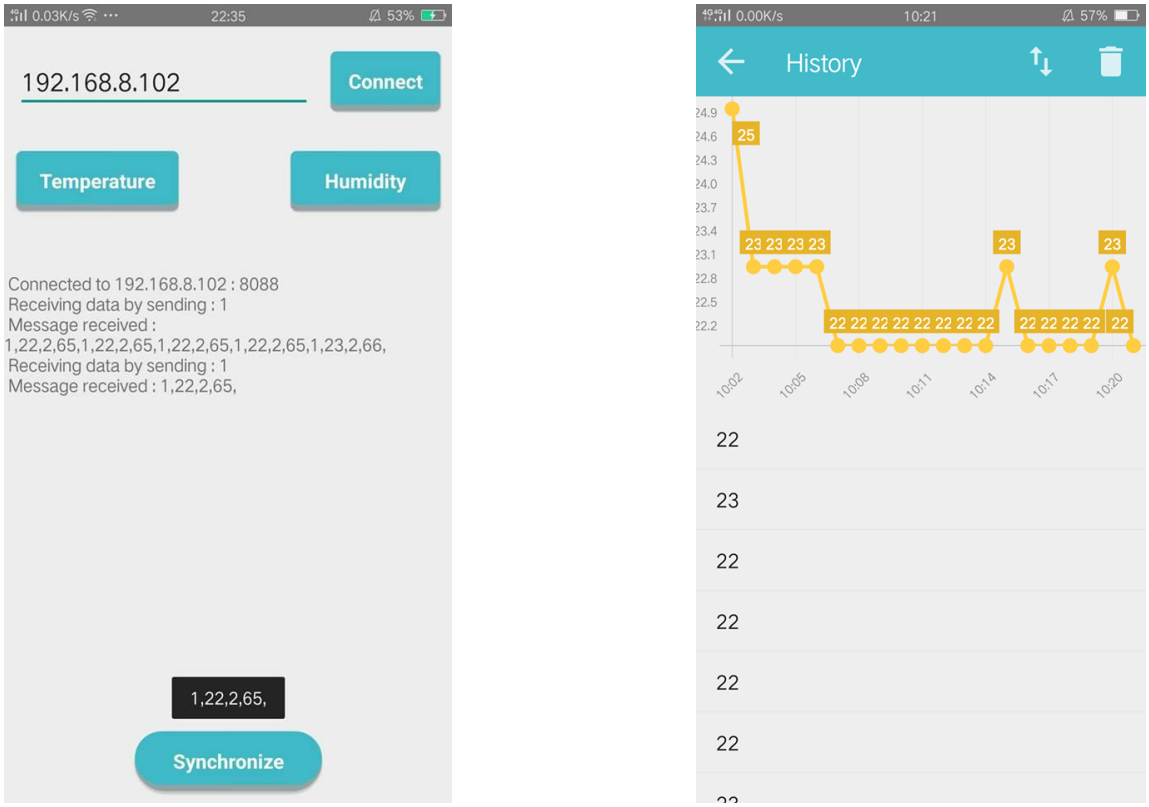

Fig 14 Détection de la température et de l'humidité

La deuxième fonction est beaucoup plus simple, elle comporte un bouton permettant d'allumer et d'éteindre la lumière. C'est la classe LightControl qui s'occupe de cette fonction.

Classe "TemhumDetection.java"

Dans TemhumDetection, on réalise la communication TCP avec la carte qui est comme un serveur et envoie des messages de notre choix. Le processus entier est présenté en bas de l'écran par la méthode « append ». Toutes les étapes de connexion sont encapsulées dans TCPClientSender (setAddress, send, receive, quit, etc) pour gérer la connexion dans un autre Thread et pour simplifier le code. Il utilise un PrintWriter et la méthode getOutputStream pour stocker le message à envoyer au serveur et la méthode getInputStream pour stocker le message reçu de serveur.

```
@Override
public void run() {
    try {
        socket = new Socket(ipAddress,targetPort);
        socket.setSoTimeout(8000);
        isRun = true;
        os = socket.getOutputStream();

        pw = new PrintWriter(new BufferedWriter(new OutputStreamWriter( socket.getOutputStream(), "UTF-8")), true);

        is = socket.getInputStream();
        dis = new DataInputStream(is);
    } catch (IOException e)
    {
        e.printStackTrace();
    }
```

Fig 15 Déroulement de TCP Client Sender

Au début, on fixe le numéro de port et on tape l'adresse IP à laquelle on veut se connecter pour créer un nouveau socket. Ensuite, on exécute un TCPClientSender(mTCPCL) afin de commencer la connexion. A ce moment-là, le paramètre « isRun » de mTCPCL est égal à « true ». Quand un socket est créé, bind est réalisé automatiquement. On peut dès maintenant échanger des données avec le serveur.

```
btnConnect.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {

        targetIpAddress = inputText.getText().toString();
        targetPort = 8088;
        mTCPCL.setIpAddress(targetIpAddress);
        mTCPCL.setTargetPort(targetPort);
        sPort = Integer.toString(targetPort);
        btnSync.setEnabled(true);

        communication.append("\r\n Connected to ");
        communication.append(targetIpAddress);
        communication.append(" : ");
        communication.append(sPort);

        exec.execute(mTCPCL);
    }
});
```

Fig 16 Conexion TCP

On clique le bouton « Synchronize » pour envoyer un "1" en appelant la méthode « send » dans TCPClientSender.

```
btnSync.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {

        communication.append("\r\n Receiving data by sending ");
        communication.append(": ");
        mTCPCL.setTextToBeSent("1");
        exec.execute(new Runnable() {
            @Override
            public void run() { mTCPCL.send(); }
        });
        communication.append("1");
    }
});
```

Fig 17 Demande de données

Quand le serveur nous donne une réponse, TCPClientSender la garde et l'associe à une action « tcpClientReceiver ». Et puis TemhumDetection peut détecter cette action et appeler « myHandler » pour traiter les données et afficher le message sur l'écran.

```
@Override
public void onReceive(Context context, Intent intent) {
    String mAction = intent.getAction();
    switch (mAction) {
        case "tcpClientReceiver":
            String msg = intent.getStringExtra("tcpClientReceiver");
            curDate = new Date(System.currentTimeMillis());
            //communication.append(msg);
            Message message = new Message();
            message.obj = msg;
            myHandler.sendMessage(message);
            Toast.makeText(getApplicationContext(), msg, Toast.LENGTH_SHORT).show();
            break;
    }
}
```

Fig 18 Réception de données

Dans MyHandler, les données sont triées selon le chiffre précédant la valeur de température ou d'humidité. Si c'est un "1", la valeur est stockée dans la base de données "Temperature.db"; si c'est un "2", la valeur est stockée dans "Humidity.db".

```
private class MyHandler extends Handler {
    private final WeakReference<TemHumDetection> mActivity;

    public MyHandler(TemHumDetection activity) {
        mActivity = new WeakReference<TemHumDetection>(activity);
    }

    @Override
    public void handleMessage(Message msg) {
        TemHumDetection activity = mActivity.get();
        if (null != activity) {
            String str = msg.obj.toString();
            String str2="";
            communication.append("\r\n Message received : ");
            communication.append(str);
            for(int i=0;i<str.length();i++){
                if(str.charAt(i)=='1'&&str.charAt(i+1)==','){
                    int j = i+2;
                    while(j<str.length()&&str.charAt(j)!=',') {
                        str2+=str.charAt(j);
                        j++;
                    }
                    addData1(str2);
                    str2="";
                    i=j;
                }
                else if(str.charAt(i)=='2'&&str.charAt(i+1)==','){
                    int j = i+2;
                    while(j<str.length()&&str.charAt(j)!=',') {
                        str2+=str.charAt(j);
                        j++;
                    }
                    addData2(str2);
                    str2="";
                    i=j;
                }
            }
        }
    }
}
```

Fig 19 Traitement de données

Classe “TemperatureHistory.java”

Cette classe est appelée quand le bouton “Temperature” est cliqué. Elle se compose de 3 parties. D’abord le toolbar en haut permet d’afficher le titre et les boutons pour exporter les données en fichier .csv et vider la base de données.

```
private Toolbar.OnMenuItemClickListener onMenuItemClick = (menuItem) -> {
        switch (menuItem.getItemId()) {
            case R.id.action_export:
                dbHelper.exportDB(getApplicationContext());
                break;
            case R.id.action_delete:
                AlertDialog.Builder builder=new AlertDialog.Builder(TemperatureHistory.this);
                builder.setMessage("Remove all?");
                builder.setTitle("Warning");
                builder.setPositiveButton("YES", (dialog, which) -> {

                        dbHelper.deleteAll();
                        adapter.notifyDataSetChanged();
                        cursor.requery();
                        _id = 0;
                });
                builder.setNegativeButton("LET ME THINK", new DialogInterface.OnClickListener() {

                    @Override
                    public void onClick(DialogInterface dialog, int which) {

                    }
                });

                builder.create().show();
                break;
        }
        return true;
```

Fig 20 Toolbar

Ensuite, c'est un graphe qui affiche les 20 dernières valeurs reçues. Elles sont trouvées par un curseur de DBHelper et sauvegardées dans une table de 20 éléments. Les fonctions pour dessiner le diagramme sont fournies par la librairie "hellocharts". Au-dessous c'est un tableau avec toutes les données reçues. S'il n'existe pas de données, cette application affiche "Nothing received yet" et la page est blanche.

```
if(cursor.moveToFirst()) {
    for (int i = 0; i < 20; i++) {
        cursor.moveToPosition(i);
        value[19 - i] = cursor.getInt(1);
        curDate.setTime(time - 60000 * i);
        date[19 - i] = formatter.format(curDate);
    }
}
if(cursor.moveToFirst()) {
    lineChart = findViewById(R.id.line_chart);
    getAxisXLables();
    getAxisPoints();
    initLineChart();
}
else {
    Toast.makeText(getApplicationContext(), "Nothing received yet", Toast.LENGTH_LONG).show();
}
```

Fig 21 Affichage de diagramme

Classe "LightControl.java"

Cette classe suit presque le même principe que la classe "TemhumDetection", sauf qu'ici l'adresse IP est fixe. Au milieu de la page apparait un grand bouton de commutation. Quand on le glisse à gauche, il envoie un "1" à la lumière pour l'allumer. Et un "0" est envoyé en glissant le bouton à droite pour éteindre la lumière.

```
switchButton.setChecked(false);
switchButton.isChecked();
switchButton.toggle();     //switch state
switchButton.toggle(true);//switch with animation
switchButton.setShadowEffect(true);//enable shadow effect
switchButton.setEnabled(true);//enable button
switchButton.setEnableEffect(true);//enable the switch animation
switchButton.setOnCheckedChangeListener((view, isChecked) -> {
        if (isChecked) {
            mTCPCL.setTextToBeSent("1");
            exec.execute(() -> { mTCPCL.send(); });
            Toast.makeText(getApplicationContext(), "Light On", Toast.LENGTH_SHORT).show();
        } else {
            mTCPCL.setTextToBeSent("0");
            exec.execute(() -> { mTCPCL.send(); });
            Toast.makeText(getApplicationContext(), "Light Off", Toast.LENGTH_SHORT).show();
        }
});
```

Fig 22 Switch Button

Layout d'application

Les fichiers .xml se trouvant à java/res/layout gèrent l'interface graphique de l'application Android. Dans ces fichiers on peut préciser la disposition globale des pages, les éléments utilisés et leur taille, position, couleur, contenu, etc. La façon de lier ces fichiers avec les fonctions de java est d'appeler "setContentView(R.layout.xxx)" ou "findViewById(R.id.xxx)" quand on initialise les éléments par code java selon les cas différents.

Conclusion

Ce projet ambitieux a mêlé deux aspects : du hardware et du software pour l'embarqué. Nous avons programmé en C un microcontrôleur ARM Cortex M4 de TI en utilisant une librairie LwIP pour gérer le protocole TCP/IP sur un système à faible ressources et à basse consommation. Ce microcontrôleur interagit avec différents capteurs en Bluetooth. Nous avons également développé une application Android mêlant programmation socket, base de données et affichage de graphe.

Ainsi, ce projet nous a permis de nous familiariser un peu plus avec les systèmes embarqués. La figure suivante montre notre système autour de la carte Tiva C mettant en œuvre le microcontrôleur ARM Cortex M4 de TI.

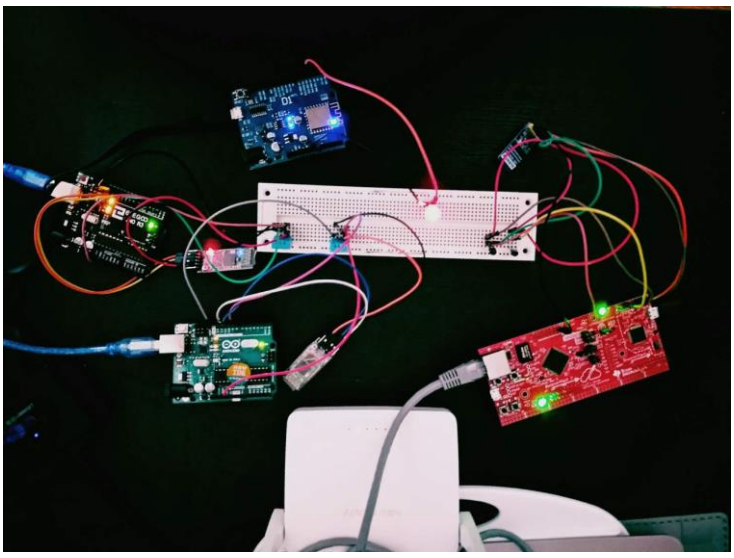

Fig 23 Objet connecté à base de microcontrôleur ARM