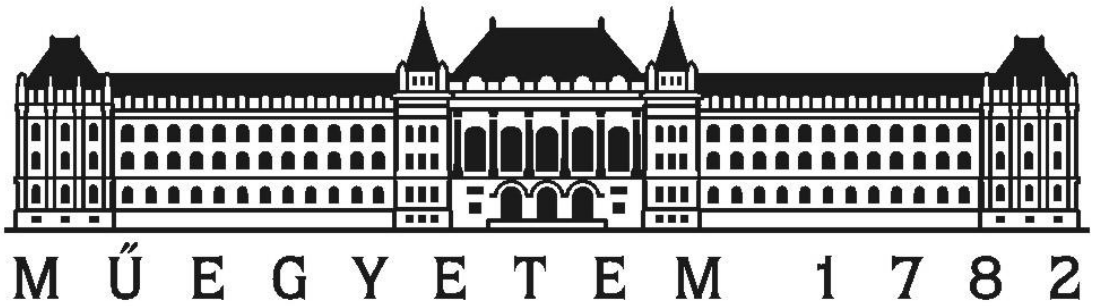



# ARM Cortex magú mikrovezérlők projekt

**Kárpáti Péter Milán**

**VW19BD**

# Tartalomjegyzék

# 1 Bevezetés

Projektem keretében egy De'Longhi kávéfőző gépbe hőmérséklet-szabályozást építettem, és a csepegtetőtálat súlymérési funkcióval egészítettem ki. A kávéfőzőhöz egy asztali alkalmazást is készítettem, amelyen grafikonon követhetjük a kávéfőző által mért adatokat. A számítógéphez a kávéfőző USB-n keresztül csatlakoztatható. Amennyiben nem szeretnénk a számítógéphez csatlakoztatni, az eszköz rendelkezik egy kijelzővel is, amin szintén láthatóak az adatok. A projekt elkészítéséhez egy STM32F411-es mikrovezérlőt használtam, ami egy ARM M4-es maggal rendelkezik.

A munkát a projekthez szükséges elemek felmérésével és kiválasztásával kezdtem. A modulok kiválasztása után a TouchGFX dokumentációjának tanulmányozásával és a kijelző illesztésével folytattam, erről a 2.1-es fejezetben írok. Ezt a hőmérséklet-szenzor beépítése és a hozzá tartozó szabályozó algoritmus megírása követte, melyet a 2.2-es fejezetben fejtek ki. Ezután a mérleg és a hozzá tartozó driver és hardver implementálása következett, ezt a 2.3-as fejezetben részletezem. Ezt követően mikrovezérlő és számítógép közötti USB-kommunikáció megvalósítását és az asztali program megírását végeztem el (2.4 fejezet).

Utolsó feladatom a szoftver komponensek integrálása és a megfelelő szoftverarchitektúra megvalósítása volt. A szoftver fejlesztésénél különös figyelmet kellett fordítanom a biztonságos működésre, hiszen hálózati feszültséggel és magas nyomással dolgoztam. (2.5 fejezet).

Végezetül a projekt jövőbeli fejlesztési terveit vázolom fel (3.0 fejezet).

# 2 Az egyes modulok bemutatása

## 2.1 Kijelző

A kijelző felé támasztott elvárásaim a következőek voltak. Legyen legalább **3,5”** hogy egyszerre grafikonok és adatok is megjeleníthetőek legyenek. Emellett rendelkezzen érintő panellel, hogy közvetlen a kávégépről is lehessen állítani a paramétereket. A másik fontos szempont az volt, hogy rendelkezzen SPI interfésszel, mert a mikrovezérlő limitált lábszáma miatt párhuzamos kommunikációra nincs lehetőség. Végül, de nem utolsó sorban egy 5000 forintos összeghatárt húztam meg.

A fenti követelményeknek megfelelően végül egy Ali Express-es 4” inches kijelzőt választottam. [1]

1. ábra Ali Express-es kijelző

Ahhoz, hogy a kijelzőn szép és igényes GUI-t tudjak megjeleníteni, szükség van egy grafikus engine-re. Az STM32 környezetbe integráltsága miatt erre a célra a TouchGFX-et választottam.

A TouchGFX alap működési elve, hogy van egy frame bufferünk, ezt módosítja maga az engine. Majd ezt a frame buffert kiküldjük a kijelzőnek megjelenítésre. [2] A limitált RAM memória miatt, a partial frame buffer startégiát választottam, amely azt jelenti, hogy egy frame buffer helyett, amely eltárolja kijelző minden pixelét, több részleges frame buffert használunk, amik a kijelző kisebb részét tárolják és mindig a kijelző éppen változó részét küldi ki.

Ahhoz, hogy a kijelzőnk a TouchGFX-szel működjön inicializálni kell a kijelzőt és három függvény kell implementálnunk [3]:

### 1. Egy függvényt, amellyel kiküldjük a megjeleníteni kívánt blokkot

A függvényt az alábbi módon valósítottam meg:

```
void touchgfxDisplayDriverTransmitBlock(uint8_t* pixels, uint16_t x, uint16_t
y, uint16_t w, uint16_t h)
{
    isTransmittingData = 1;
    ST7796_SetWindow(x, y, x+w-1, y+h-1);
    ST7796_DrawBitmap(w, h, pixels);
}
```

A függvény segítségével a kijelző határain belül a TouchGFX kijelöl egy tetszőleges méretű téglalap alakú blokkot, amelyet frissíteni fog, majd SPI kommunikáció segítségével egy DMA-csatornán elküldi a kijelzőnek.

### 2. Egy függvényt, amely megmondja, hogy éppen folyik-e adat küldés

```
uint32_t touchgfxDisplayDriverTransmitActive(void)
{
    return isTransmittingData;
}
```

### 3. Egy függvény, ami elindítja a következő blokk küldését

```
void HAL_SPI_TxCpltCallback(SPI_HandleTypeDef *hspi)
    if (hspi->Instance == SPI1) {
        isTransmittingData = 0;
        DisplayDriver_TransferCompleteCallback();
    }
}
```

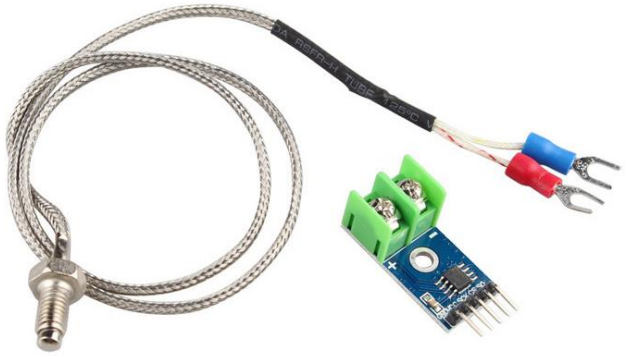
Ez a függvény akkor hívódik meg, amikor a DMA végzett az adatok elküldésével. Ebben a függvényben 0-ra állítom az adat küldés flagjét, majd meghívom a DisplayDriver_TransferCompleteCallback() függvényt ami meghívja a startNewTransfer() függvényt ezáltal elindítva az új adat csomag küldését.

Emellett a működéshez szükség van még fő szinkronizációs függvény meghívására, ami azt jelzi, hogy az adott képkocka adatait kiküldtük. Mivel nincsen operációs rendszer, amely valamilyen ütemezés szerint meghívná a ***touchgfxSignalVSync()*** ezért egy 18 Hz-es timer interrupt rutinban hívom meg a függvényt. (részletesebben a 2.5 fejezetben)

A kijelző driver részletesebben SmartCaso_F411\Drivers\ST7796 mappában érhető el.

## 2.2 Hőmérséklet szabályozás

A hőmérséklet szenzornak egy MAX6675 modult választottam K-típusú hőelemmel. A fő szempont az volt, hogy SPI kommunikációra képes, emellett 0.25 °C felbontással és a felhasználási célnak megfelelő mérési tartománnyal rendelkezik.



2. ábra MAX6675 K-típusú hőelemmel

A modullal való kommunikációhoz a Chip Select lábat alacsony állapotba állítom, majd az ST által biztosított HAL könyvtár ***HAL_SPI_Receive*** függvény segítségével megkapjuk a hőmérséklet adatokat.

1. táblázat MAX6675 adat struktúra

| BIT | DUMMY SIGN BIT | 12-BIT TEMPERATURE READING | | | | | | | | | | | | THERMOCOUPLE INPUT | DEVICE ID | STATE |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| | 0 | MSB | | | | | | | | | | | LSB | | 0 | Three-state |

A kiolvasás során 16 kimeneti bitet kapunk. Az első D15 bit, egy fiktív előjel bit és mindig nulla. A D14–D3 bitek tartalmazzák a konvertált hőmérsékletet MSB-től LSB-ig terjedő sorrendben. A D2 bit normál esetben alacsony és magasra vált, amikor a modul bemenete nyitott.

A hőmérséklet konvertálását végző kódrészlet:

Az adatlap szerint, ha az adat bitek csupa nullák azt jelenti, hogy az kiolvasott hőmérséklet 0 °C, és ha csupa egyek akkor +1023,75 °C. Ez alapján a hőmérsékletet megkaphatjuk az alábbi módon [4]:

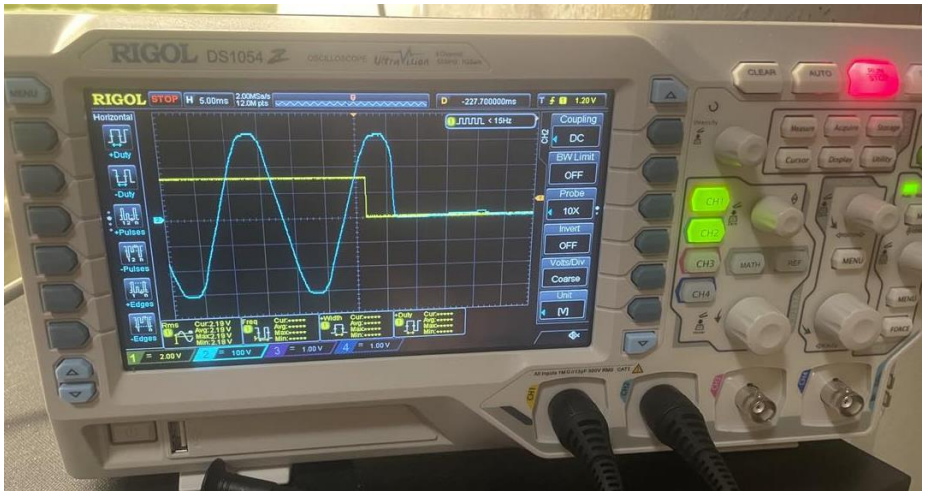
```
Temp=((((DATARX[0]|DATARX[1]<<8)))>>3);                    // Hőmérséklet adat
Temp*=0.25;                                                // Adat Celsiusba
```

Az adatlap továbbá azt írja, hogy az adatokat kiolvasása 0,22 másodpercenként történhet. Ezért a kiolvasást egy 0,25 másodpercenként meghívódó timer interrupt rutinban végzem. (részletesebben 2.5 fejezetben).

A MAX6675 driver részletesebben SmartCaso_F411\Drivers\MAX6675 mappában érhető el.

A kávéfőző bojlerének hőmérséklet szabályozásához a mikrovezérlő PID algoritmus alapján meghatároz egy vezérlő jelet. Majd ez alapján állít egy PWM kimenet impulzus szélességét. A jel egy TRU COMPONENTS félvezető relét vezérel, amely nyitott állapotban rákapcsolja a hálózati feszültséget a bojlerre.

Fontos megemlíteni, hogy a félvezető relé zero-crossing üzemmódban működik, amely azt jelenti, hogy a már vezető relé csak akkor nyeri vissza szigetelőképességét, ha a rajta átfolyó áram zérusra csökken. Mivel a hálózati feszültség 50 Hz-es ezért nem lehet magas frekvenciájú a PWM jel hiszen akkor a váltások két nullpont között történnek és ebben az esetben nincs hatásuk a kimeneti jelre. Ezért én projektemben egy 1 Hz-es PWM jelet használok.




3. ábra zero-cross relé

A szabályozáshoz statikus anti-windup PID szabályozót alkalmazok. A konstansok meghatározásánál komoly problémát okozott, hogy a rendszer nagy késleltetéssel rendelkezik. Ez két okból fakad, az egyik, hogy a fűtőszál nagyon gyorsan felmelegszik, de a körülötte lévő víz lassabban. A másik oka, hogy a hőmérsékletet a bojler felületén mérem, így a vízen felül a fém háznak is fel kell melegednie. A szabályozásban további problémát okoz, hogy negatív szabályozó jelet nem tudunk kiadni (hűteni nem tudjuk). Így, ha túllő a rendszer akkor nagyon lassan hűl vissza. Ezen okok miatt sem a Matlab PID kalibráló szoftvere sem az általánosan alkalmazott módszerek nem váltak be. Ezért az értékeket empirikus módszerrel állítottam be.

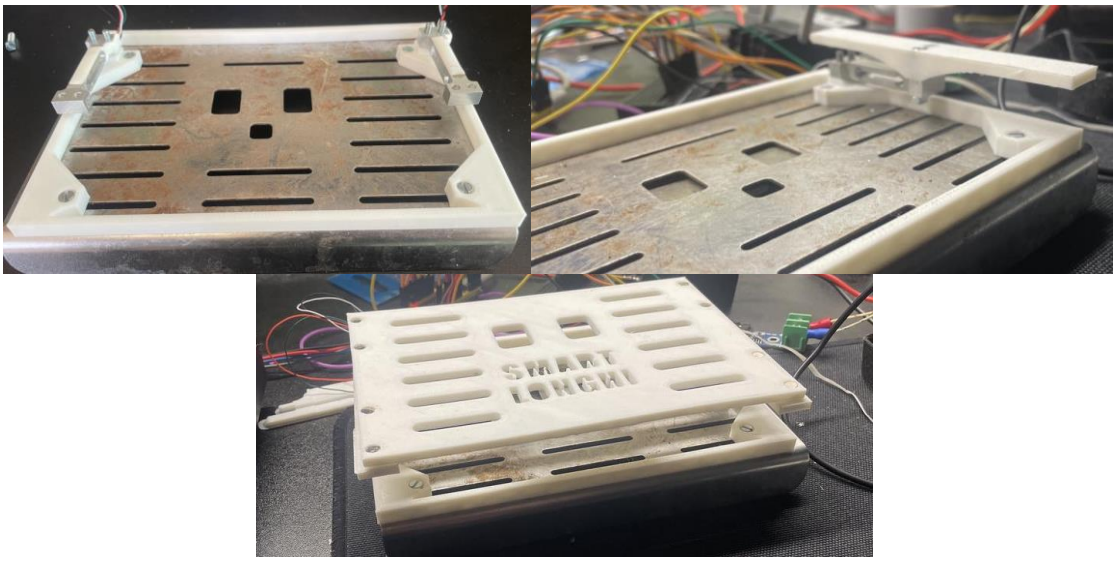
```
float proportional = pid->Kp * error;

pid->integrator = pid->integrator + 0.5f * pid->Ki * pid->T * (error + pid-
>prevError);

pid->differentiator = -(2.0f * pid->Kd * (measurement - pid>prevMeasurement)
                        + (2.0f * pid->tau - pid->T) * pid->differentiator)
                        / (2.0f * pid->tau + pid->T);
```

A pontos implementáció a SmartCaso_F411\Core\TemperatureControl.c fájlban.

A könyvtárat GitHub-ról töltöttem le [5].

## 2.3 Beépített mérleg

A mérleg megvalósításához 2 db 500g-os erőmérő cellát és HX711-es AD konvertert választottam. Ennek a modulnak az implementálására először 3D nyomtatóval egy platformot készítettem, hogy az erőmérő cellák alkalmasak legyenek a súly mérésére. A kezdeti próbálkozások nem voltak elég merevek, amely rontott a pontosságon. Ezért ez számos újra tervezést igényelt.



4. ábra Mérleg platform

A HX711 modul egyedi soros kommunikációval rendelkezik és 2 csatornával rendelkezik, melyek erősítését a kiküldött órajel impulzusok száma határozza meg.

| PD_SCK Pulses | Input channel | Gain |
|---|---|---|
| 25 | A | 128 |
| 26 | B | 32 |
| 27 | A | 64 |

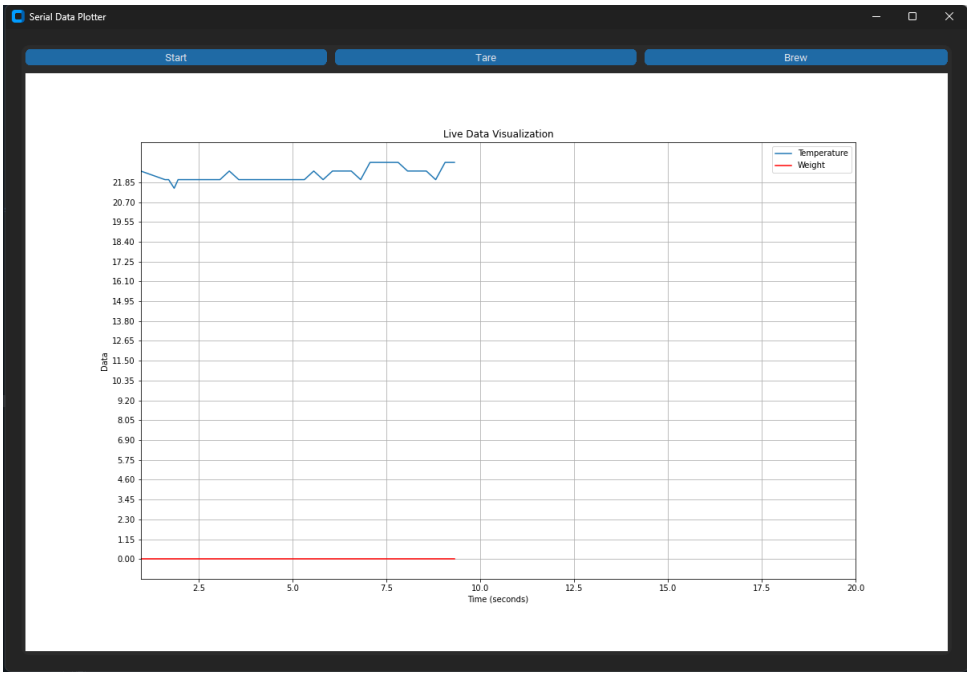
5. ábra HX711 csatornái és erősítései

A projektem során az ’A’-csatornát használtam 128-as erősítéssel. Amikor a modul nem áll készen az adatok küldésére abban az esetben a DOUT lábat magasan tartja, ekkor az órajelet alacsonyan kell tartanunk. Amikor DOUT alacsonyra esik, az eszköz készen áll az adatok küldésére, ekkor 25 pozitív óraimpulzussal kiolvassuk az adatokat. Minden egyes órajel impulzus egy bitet tol ki, az MSB-től kezdve, amíg mind a 24 bit kiküldésre kerül. A 25. impulzusra a modul DOUT-ot magasra húzza vissza. Fontos, hogy az adatok kiolvasása során ne érkezhessen interrupt, mert ha a CLK pin alacsonyól magasra változik és $60\mu s$-ig magas marad, akkor a modul alvó állapotba kerül.

A drivert az egyszerűség kedvéért GitHub-ról töltöttem le. A jövőben tervben van egy saját driver írása, mert a jelenlegi blokkoló késleltetéseket tartalmaz. Emellett az interuptok letiltása a kommunikáció közben elkerülhető lenne, ha UART-al lenne megoldva [6].

## 2.4 Asztali alkalmazás és USB kommunikáció

A kávéfőzőhöz egy asztali alkalmazást is fejlesztettem, melyben grafikonon láthatjuk a mért hőmérséklet és súlyadatokat. Az adatok küldéséhez a STM32Cube-ba beépített USB_Device middleware-t használtam a Communication Device Class beállítással.




6. ábra Asztali alkalmazás felhasználói felület

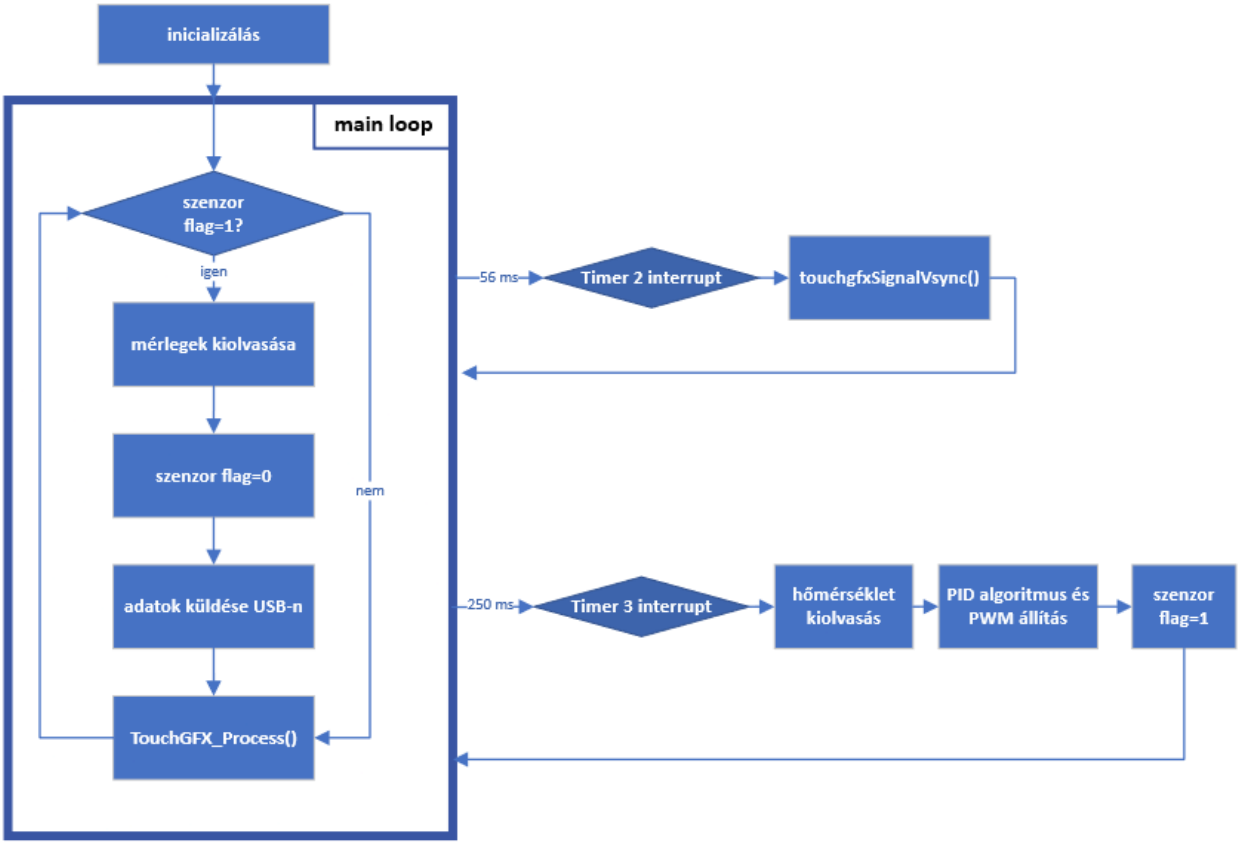
A felhasználói felületen a start gomb segítségével indíthatjuk el az adatok kirajzolását. Emellett van egy Brew gomb, amivel a jövőben a kávéfőzés indítható el. Emellett található egy Tare gomb, amellyel a mérlegeket nullázhatjuk. Ez a gyakorlatban úgy van implementálva, hogy a kezelőgombok stringeket küldenek a mikrovezérlőnek.

```
def tare_scales():
    command_tare = "tare\n"
    ser.write(command_tare.encode())
    print(f"Command '{command_tare.strip()}' sent.")

def brew():
    command_brew = "brew\n"
    ser.write(command_brew.encode())
    print(f"Command '{command_brew.strip()}' sent.")
```

A mikrovezérlőben a stringek fogadására egy cirkuláris FIFO lesz implementálva annak érdekében, hogy ne veszhessen el beérkező parancs. A USB_CDC-ben a CDC_Receive_FS az alap implementációban a beérkező üzenetek egy előre definiált tömbbe kerülnek, melyet a következő érkező üzenet fölülír [7].

## 2.5 Szoftver architektúra

A szoftver architektúrája egy megszakításokkal kiegészített round-robin. A program az alábbi UML diagrammon szemléltethető:




7. ábra Szoftver UML diagrammja

Megvalósítás:

```
while (1)
 {
   if(ReadSensorData==true){
     weightRight = get_weight(&loadcell_right, 1, CHANNEL_A);
     weightLeft = get_weight(&loadcell_left, 1, CHANNEL_A);
     weight=weightRight+weightLeft;

     ReadSensorData=false;
   }
 MX_TouchGFX_Process();
 }
}
```

Mivel erőmérő cellák kiolvasása nem időzítés kritikus, ezért ezt a fő ciklusban végzem. A modul kiolvasását egy flag engedélyezi, melyet a hőmérséklet mérést végző interrupt billent be.

Fontos, hogy az erőmérő cellát ne az interrupt-ban kezeljem le, mert blokkoló késleltetés tartalmaz.

A másik elem, amely megjelenik a fő ciklusban az a TouchGFX üzleti logikáját végző programrész.

A program során az alábbi timer interruptok érkeznek:

1. A TouchGFX képfrissítését végző interrupt:

```
if (htim->Instance == TIM2) {
    touchgfxSignalVSync();
  }
```

Mivel a kijelző nem küld visszajelzést az adatok megérkezéséről, ezért a nekem kell meghívnom a szinkronizációra szolgáló függvényt, de kérdés, hogy milyen időközönként. A teljes kijelző frissítéséhez szükséges pixel adatok kiküldése 16 bites színmélység esetén:

$$T = \frac{320 \cdot 480 \cdot 16}{48 \cdot 10^6 \cdot 8} = 0.0512\ s\ \approx 19.5\ Hz$$

Ezek alapján a timert úgy állítottam be, hogy 18 Hz-vel küldje a timer interruptokat.

2. A hőmérséklet kiolvasását és a PWM jelet állító interrupt:

```
if (htim->Instance == TIM3) {
   temperature=Max6675_Read_Temp();
   control=PIDController_Update(&temp_controller,goal_temp,temperature);
   __HAL_TIM_SET_COMPARE(&htim5, TIM_CHANNEL_1, control);
   ReadSensorData=true;
}
```

A hőmérséklet kiolvasása és a PWM jel impulzus szélességének állítása időzítés kritikus, hiszen az PID algoritmusban fix mintavételezési idővel számolunk, ha nem tudnánk pontosan a mintavételezési időt az hibát okozhatna a szabályozásban. Emellett, ha a program bármely részén megakad nagyon fontos, hogy a bojler hőmérséklete továbbra is szabályozva legyen. Amennyiben a bojler eléri a 184°C fokot a hőbiztosíték kiold és cserére szorul. Annak érdekében, hogy ezt biztosítsam ez a legmagasabb prioritású interrupt.

# 3 Fejlesztési tervek

A jövőben a kijelző le lesz cserélve egy 7” inches kapacitív érintőpaneles kijelzőre, mert a jelenlegi kijelző kicsi és a rezisztív érintő panel irreszponzív. Az új kijelző egy ESP32-es mikrovezérlővel van felszerelve, mellyel az USB-s asztali alkalmazás egy webszerveresre lesz lecserélve. Ezáltal elkerülhető a kábel használata. Tervben van a nyomás szabályozás implementálása is.

# 4 Irodalomjegyzék

[1] L. Wiki, „http://www.lcdwiki.com/4.0inch_SPI_Module_ST7796,”.

[2] T. Documentation, „https://support.touchgfx.com/docs/introduction/welcome,”.

[3] HELENTRONICA, „https://helentronica.com/2021/01/22/touchgfx-on-a-custom-made-low-cost-board-with-the-ili9341-controller-over-spi/,”.

[4] Digikey HX711 Datasheet, „https://www.digikey.com/htmldatasheets/production/1836471/0/0/1/hx711.html,”.

[5] P. Salmony, „https://github.com/pms67/PID/commits?author=pms67,”.

[6] N. Askari, „https://github.com/nimaltd/HX711/tree/master,” .

[7] N. Online, „https://nefastor.com/microcontrollers/stm32/usb/stm32cube-usb-device-library/communication-device-class/,”.

[8] P. Salmony, „https://github.com/pms67,”.