Jelszókezelő hardver fejlesztése

Készítette: Nyitrai Bence (QEM0OR)
Konzulens: Kovács Viktor

Tartalomjegyzék

Feladat kiírása:....1
Bevezetés:....1
Működés:....2
USB Composite Device....2
USB HID....3
USB MSC....3
Fatfs....5
Belső flash....6
USB CDC....7
Főprogram....8
Összefoglalás:....11

Feladat kiírása:

A cél STM32F407-es mikrokontrolleren kifejleszteni egy beágyazott szoftvert, amely Universal Serial Bus-on (USB) keresztül kommunikál egy asztali számítógéppel. Az eszköz gombnyomásra képes a rajta eltárolt jelszavakat billentyűleütésekként beírni a számítógépen megnyitott weboldalra vagy alkalmazásra. A tárolás perzisztens módon a mikrokontroller flash memóriájában legyen megvalósítva tömör rekordokként. Egy rekord tartalmazza az alkalmazás nevét, ahova szeretnénk bejelentkezni, az ehhez tartozó felhasználónevet és jelszót és ezek beírásához szükséges tabokat, entereket. A rekordok chacha20 algoritmussal legyenek titkosítva az eszköz feloldásáig. A felhasználó tudja módosítani a jelszavakat, illetve vehessen fel újakat, és törölhesse is őket. Az eszközön lehessen beállítani, hogy milyen nyelvű billentyűzet van a számítógéphez csatlakoztatva, hogy ennek megfelelő billentyűleütések kerüljenek elküldésre.

Bevezetés:

Az eszköz működésének az alapját egy USB Composite Device adja. Ennek az eszközosztálynak a használata meglehetősen előnyös a probléma megoldására. Alkalmazásával lehetséges több különböző USB funkciót ellátni egyetlen fizikai eszközön belül, így lehetőség nyílik a teljesen különböző feladatokat végző komponensek működésének egyszerű összehangolására. A

Composite Device egyszerre valósít meg egy Human Interface Device (HID) osztályt, egy Mass Storage Class-t (MSC) és egy Communication Device Class-t (CDC) is.

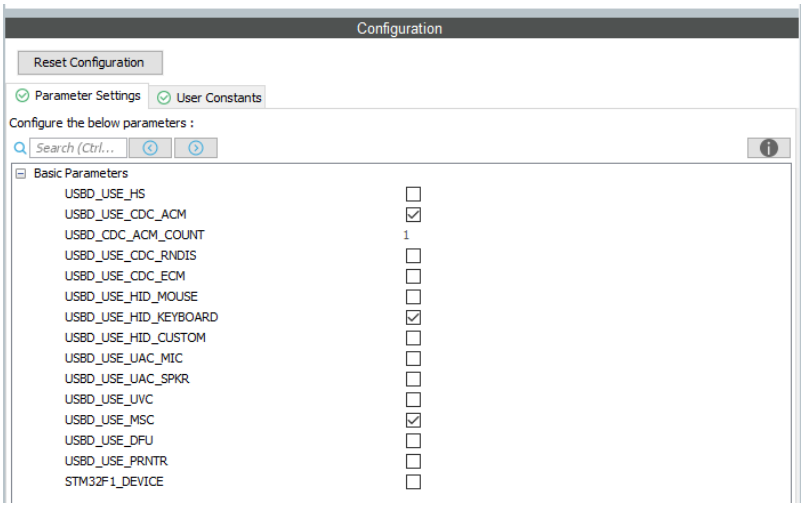
Az USB HID implementálja a billentyűzetet, ennek az eszközosztálynak a felelőssége a billentyűkódok továbbítása a személyi számítógép felé. Az USB CDC teremt kapcsolatot az eszköz és a felhasználó között. A felhasználó egy soros portot kezelő alkalmazáson (PuTTY, hercules) keresztül tud vezérlő üzeneteket küldeni a mikrokontrollernek, amely ezeknek az utasításoknak megfelelő műveleteket hajt végre. A vezérlő üzenetek pontos működése később lesz részletezve. A harmadik eszközosztály az USB MSC pedig az adatok Random Access Memory-ban (RAM) történő tárolását, és ezek módosítását, törlését valósítja meg. Ez az osztály teszi lehetővé, hogy az eszköz a host számítógépről egy külső meghajtónak tűnjön, így egyszerűen fájlokként lehet kezelni a különböző jelszavakat tároló rekordokat.

A fájlok tárolását és kezelését segíti a RAM-ba leképezett File Allocation Table file system (Fatfs). Használata lehetővé teszi az új jelszavak könnyű eltárolását és a már meglévő rekordok listázását. A Fatfs-ben tárolt fájlok szinkronizálva vannak az USB MSC-vel és a mikrokontroller belső flash memóriájával is, tehát ha az asztali számítógépen elmentünk egy fájlt a külső háttértárolóra, az a fájl a Fatfs-be bekerüléskor automatikusan mentődik a belső flash memóriába is, ahol a tápfeszültség megszűnése után is megörződik.

Működés:

USB Composite Device

Az USB Composite Device eszközosztály Full Speed (FS) üzemmódban használom. Ez az osztály alapból nincs támogatva az STMicroelectronics CubeIDE-ben, ezért egy nyílt forráskódú USB csomagolót használtam. A projekt elérhető ezen a linken: https://github.com/alambe94/I-CUBE-USBD-Composite . A README fájlban leírtak alapján egyszerűen hozzáadható a projektünkhöz, ott pedig az ioc fájlban grafikusan kiválaszthatjuk, hogy milyen eszközosztályok kerüljenek megvalósításra.



1. ábra: Eszközosztályok kiválasztása

A kódgenerálás után a Middlewares mappán belül találjuk a COMPOSITE mappában a különböző USB osztályok fájljait, amelyekkel dolgoztam. A Composite osztály forrásfájlját megvizsgálva láthatjuk, hogy először kioszt endpointokat a kiválasztott osztályoknak, majd meghívja a Setup

függvényüket. A végpont címek fontosak a kommunikációs csatornák megfelelő működésének szempontjából. A Composite osztály fő szerepköre ezek után a ki és bemenő adatforgalom eligazítása a megfelelő eszközosztályokhoz.

USB HID

Az USB HID eszközosztályom egy billentyűzetet valósít meg. Ennek a konfigurálása a my_usbd_hid_keyboard.c fájlban történik, az USB HID konfigurációs leírójában tudjuk ezt beállítani.

```
0x01, /* nInterfaceProtocol: 0=none, 1=keyboard, 2=mouse */
```

Ebben a leíróban találhatók egyéb fontos információk is, mint a maximális csomagméret, az IN (device → host) endpoint címe és a Report Descriptor mérete. A HID billentyűzet jelentés leírója határozza meg, hogy milyen formátumban és struktúrában küldi az eszköz az adatokat a számítógépnek. Ezt a leírót teljesen át kellett írni, mivel az alapértelmezett egy egérnek a működését valósította meg. A leíró, amit használok elérhető ezen az oldalon: https://www.instructables.com/STM32-As-HID-USB-Keyboard-STM32-Tutorials/ . Az itt található értékek közül a legfontosabb:

```
0x75,0x08, // Report Size (8)
```

The following table represents the keyboard input report (8 bytes).

Byte	Description
0	Modifier keys
1	Reserved
2	Keycode 1
3	Keycode 2
4	Keycode 3
5	Keycode 4
6	Keycode 5
7	Keycode 6

Note Byte 1 of this report is a constant. This byte is reserved for OEM use. The BIOS should ignore this field if it is not used. Returning zeros in unused fields is recommended.

2. ábra: HID Report

Ez határozza meg, hogy egy elküldött üzenet egyszerre hány billentyűkódot tartalmazhat. A billentyűkód lehet KEYCODE, ami betű, szám vagy valamilyen egyéb speciális karakter, és MODIFIER is lehet, ide tartoznak a jobb és bal shift és alt billentyűk. Ha egy üzenetben beállítunk egy MODIFIER-t, akkor az üzentben levő összes KEYCODE-ra hat, ezért a megoldásomban mindegyik üzenet csak egy darab KEYCODE-ot tartalmaz.

A feladat megvalósítását ennek az eszközosztálynak az implementálásával kezdtem. Ez viszonylag sok időt vett igénybe, mivel ahhoz, hogy elkezdhessek dolgozni először jobban meg kellett ismerkednem az USB dokumentációval. Amire érdemes különös hangsúlyt fektetni a HID eszköznél az a Report Descriptor helyessége. Az elsőnek használt leírómba néhány érték nem volt helyes, így a host számítógép nem tudta felismerni a HID eszközt.

USB MSC

Ennek az eszközosztálynak a segítségével lehet módosítani a már meglévő jelszavakat és újakat is felvenni. Az msc_if.c fájlban módosítani kell ezeket az értékeket:

```
#define STORAGE_LUN_NBR                  1
#define STORAGE_BLK_NBR                  192
#define STORAGE_BLK_SIZ                  512
```

Az első érték a Logical Unit Numbers (LUN) határozza meg, hogy az adatok egy partícióba lesznek tárolva. A második érték a maximális szektor szám, egy szektor tárol egy darab tömör rekordot. Az utolsó, pedig a szektor méretét definiálja, ezt 512 bájtnak választottam. Ebbe a méretbe kényelmesen belefér egy rekord. Az eszköz összes tárhelye tehát 192*512 = 96 kByte.

Az msc_if.c forrásfájlon belül kellett megvalósítani a működéshez szükséges függvényeket. Ezek között találhatóak rövid, egyszerű feladattal rendelkezők, mint a GetCapacity, ami visszaadja az MSC-nek szánt tárhely méretét, vagy az IsReady függvény, amely jelzi, hogy érkezhet-e a következő üzenet. A megvalósítás legfontosabb része az író és az olvasó függvény implementálása volt.

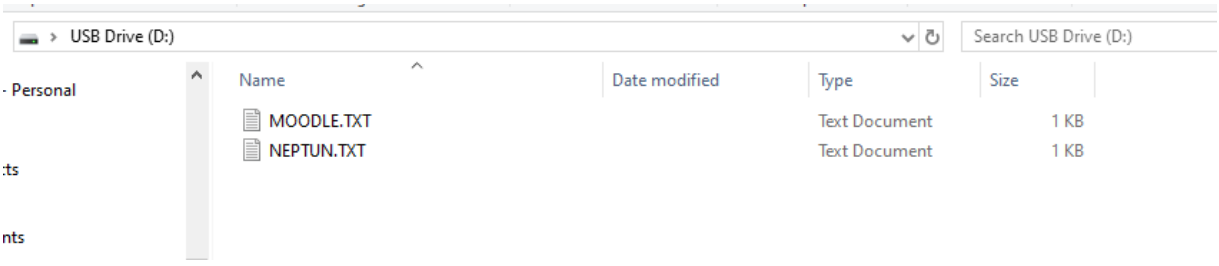
```
if (USER_read(lun, buf, blk_addr, blk_len) != RES_OK)
    return USBD_FAIL;
```

Olvasáskor meghívjuk a Fatfs olvasó függvényét, ezzel biztosítva, hogy az USB MSC és a Fatfs által tárolt adatok, azonos memóriaterületen legyenek, így a felhasználó mindig a legfrissebb, konzisztens adatokat látja. Az író függvény hasonló módon operál, viszont ennek a függvénynek még a kapott fájlokat mentenie kell a belső flash memóriába is.

```
if (!memcmp(txt, file, 4)) {
      for (uint8_t *c = buf; *c != '.'; c++) {
            filename[name_char] = buf[name_char];
            name_char++;
      }
      if (Fat_Write((strcat(filename, ".txt")), buf) != FR_OK)
            return USBD_FAIL;
      else
            return USBD_OK;
}
```

Az USB MSC-n keresztül a rekordok mellett vezérlő üzenetek is érkeznek, ezeket az üzeneteket nyilvánvalóan nem szeretnénk menteni. A rekordokat onnan tudjuk megkülönböztetni a vezérlő üzenetektől, hogy a rekord elején a fájlnév után rendelkeznek egy .txt szöveggel. A Fat_Write függvény gondoskodik a flashbe történő mentésről.

Az USB MSC elkészítésekor a flashbe való mentés időzítésével voltak problémák. Először minden üzenet érkezésekor frissítettem a flash-t, viszont annak az írása meglehetősen sok időt vesz igénybe, így túl volt terhelve a rendszer. A frissítést ezután megpróbáltam timer-es időzítéshez kötni, ezzel az egyik gond az mentési idő helyes megválasztása volt. Ha túl gyakori, akkor ez is képes a rendszer túlterhelésére, viszont ha nem elég gyakori, és az eszköz elveszti a tápfeszültséget, akkor nagy valószínűséggel adatvesztés fog bekövetkezni. Végül ezek a hátrányok ösztönöztek arra, hogy a fentebb említett, ha érkezik egy rekord, akkor azt mentjük egyből módszert válasszam.



3. ábra: USB MSC miatt külső háttértárnak látszik az eszköz

Fatfs

A Fatfs konfigurációját először az ioc fájlal kezdtem. Itt a User-defined módot választottam, aztán itt is beállításra került az 512 byte-os szektorméret és az 1 LUN, ezek mellett engedélyeztem a fájlrendszer írását. Ezek után következett a user_diskio.c fájl, ahol először megadtam a fájlrendszer kezdőcímét a RAM-ban. Itt úgyszintén találhatók rövidebb konfiguráló függvények, amelyek teljesen hasonlóak az MSC-hez, így azokat nem részletezem. A fájlrendszert író és olvasó függvény teljesen analóg módon működik.

```
memcpy((void*)(RAM_START+(sector*SECTOR_SIZE)), buff, count*SECTOR_SIZE);
    return RES_OK;
```

Itt az íráshoz használt kódrészlet látható, lényegében a buff tartalmát másoljuk be arra a helyre, ami a Fatfs kezdetétől a sector változóban tárolt értékkel van eltolva. Az író és olvasó feladatokat ellátó függvények után még a szektorok számát is megadtam az I/O control nevű függvénynek.

A fatfs.c forrásfájl tartalmazza a Fatfs funkcionalitásokat megvalósító függvényeket.

```
if (f_mkfs((TCHAR const*) USERPath, FM_ANY, 0, buff, sizeof buff) == FR_OK)
            if (f_mount(&USERFatFS, (TCHAR const*) USERPath, 1) == FR_OK) {
                  Flash_Read_Init();
                  return;
            }
```

Az Init függvényből látható ez a kódrészlet, ahol az f_mkfs függvény megpróbálja a Fatfs számára használhatóra formázni a RAM memóriát a korábban megadott területen, ha ez sikeres, akkor az f_mount pedig beregisztrálja a fájlrendszer objektumot a Fatfs-be. Ha ez is sikeres akkor beolvassuk a flash memóriában tárolt adatokat a fájlrendszerbe.

Az írás és olvasás itt is nagyon hasonlít egymásra.

```
if ((ret=f_open(&USERFile, filename, FA_OPEN_EXISTING | FA_READ)) == FR_OK)
            if ((ret=(f_read(&USERFile, (void*) buff, RECORD, (void*)
&rbytes)))== FR_OK) {
                  f_close(&USERFile);
                  return ret;
            }
f_close(&USERFile);
```

Olvasáshoz először meg kell, hogy nyissuk a fájlt. Az FA_OPEN_EXISTING gondoskodik róla, hogy ha nem létezik a filename-be megadott nevű fájl, akkor hibával térjen vissza a megnyitás. Az f_read függvény buff-ba másolja a rekord 512 bájtját, végül bezárjuk a fájlt. Teljesen hasonló mechanizmussal működik az olvasás is.

A Fat-hoz található még egy függvény, amelynek a feladata a fájlrendszerben található összes fájl nevének kiolvasása, és ezeknek a kiküldése a soros port-on keresztül a felhasználónak, aki majd ezek közül tud választani, hogy melyik alkalmazásba jelentkezzen be.

A Fatfs használat során több probléma is merült fel. Elsőnek az volt, hogy alapból túl kevés szektort foglaltam neki, ám ezt könnyen lehetett orvosolni a szektorszám növelésével. Ezután, ha a fájlrendszer üres, mert mondjuk töröltek belőle mindent, akkor belekerül egy „SYSTEM~1” nevű fájl, amit ha véletlenül megpróbálunk olvasni, akkor az egész fájlrendszer elszáll. Ezt a hibát a fájlok listázásakor történő vizsgálattal sikerült kijavítani. A többi probléma a mikrokontroller belső flash memóriájával való együttműködésből következett.

Belső flash

A projekt legnagyobb részét a belső flash memóriával kapcsolatos műveletek megvalósítása foglalta magába. A tárolás jól működő kialakítását nagyban nehezítette az STM32F407-es Disco board belső flash-e.

Table 5. Flash module organization (STM32F40x and STM32F41x)

Block	Name	Block base addresses	Size
Main memory	Sector 0	0x0800 0000 - 0x0800 3FFF	16 Kbytes
	Sector 1	0x0800 4000 - 0x0800 7FFF	16 Kbytes
	Sector 2	0x0800 8000 - 0x0800 BFFF	16 Kbytes
	Sector 3	0x0800 C000 - 0x0800 FFFF	16 Kbytes
	Sector 4	0x0801 0000 - 0x0801 FFFF	64 Kbytes
	Sector 5	0x0802 0000 - 0x0803 FFFF	128 Kbytes
	Sector 6	0x0804 0000 - 0x0805 FFFF	128 Kbytes
	. . .	. . .	. . .
	Sector 11	0x080E 0000 - 0x080F FFFF	128 Kbytes
System memory		0x1FFF 0000 - 0x1FFF 77FF	30 Kbytes
OTP area		0x1FFF 7800 - 0x1FFF 7A0F	528 bytes
Option bytes		0x1FFF C000 - 0x1FFF C00F	16 bytes

4. ábra: STM32F407VGT6 Flash memory map

A rekordok törlése, írása egyszerűbben megvalósítható lehetett volna, ha a mikrokontroller 512 byte-os flash sector-okkal rendelkezik.

Viszont itt nagy szektor méretek vannak, így az összes rekord belefér egy szektorba. A rekordok tárolása redundásan, két szektorosan lett megvalósítva a 10-es és a 11-es szektorban. Ha változtatni akarjuk az egyik rekordot, akkor az egész flash szektor tartalmát törölni kell, mivel a flash memória csak 1→0-ba tud csak adatot módosítani. Ha kitöröljük az egész szektort és utána a rekordokat egyesével elkezdjük visszaírni és ilyenkor marad abba a tápellátás, akkor nem csak a módosítandó, hanem lényegében az összes rekord inkonzisztens állapotba kerülhet. Ennek a megoldását nyújtja a két szektoros módszer. Egy sorszám összeggel nyomon követjük, hogy melyik szektor a régebbi. Ha írni kell, akkor először a régebbi szektort frissítjük, majd növeljük ennek a sorszám összegét és így ez lesz az újabb blokk. Ha a frissítés közben tápkiesés lenne, akkor semmi probléma, hiszen a másik szektor az újabb, így annak az értékei az aktívak. A következő írási ciklusban pedig a régebbi szektor érvénytelen adatai is megjavulnak.

Az első flash művelettel kapcsolatos függvény a Flash_Read_Init, ez felel indításkor a rekordok RAM-ba való betöltéséért.

```
if ((seqnum11>seqnum10)||((seqnum11<seqnum10) && (seqnum10-seqnum11 ==
255))) // or it's ...255->0
            ACTIVE_SECTOR = SECTOR_11; // sector 11 is currently active
      else
            ACTIVE_SECTOR = SECTOR_10; // sector 10 is currently active
```

Az a szektor az újabb amelyiknek nagyobb a sequence number-je. Az aktív szektor kiválasztásához még vizsgálni kell a szektor ellenőrző összeget. Az összeg úgy kerül meghatározásra, hogy a szektorban tárolt összes adatbájtot XOR-ozzuk, ebbe belevesszük sorszám összeget is. Az aktív szektor meghatározása után az ebben található rekordokat bemásoljuk a Fatfs-be.

A Flash_Write nevű függvény valósítja meg a flash szektorokba történő mentést. Először megvizsgálja, hogy melyik szektor az aktív, aztán ennek megfelelő sorrendben hívja meg a Write_Sector10 és Write_Sector11 függvényt. Ezeknek a működése teljesen hasonló, azért van két külön függvény, mert különben túl sok paramétert kéne átadni, ha csak egy lenne. Kezdésnek törli a szektor tartalmát, majd megnyitja a Fatfs-en belül azt a könyvtárat, ahol a fájlok vannak.

```
for (uint32_t j = 0; j < 192; j++) {
      filestatus = f_readdir(&dir, &fileinfo);
```

```
        if (filestatus != FR_OK || fileinfo.fname[0] == 0)
            break; /* Break on error or end of dir */

        if (Fat_Read(fileinfo.fname, (void*) sector) != FR_OK)
            break;

        for (uint32_t i = 0; i < RECORD; i++) {
            ret = HAL_FLASH_Program(FLASH_TYPEPROGRAM_BYTE,
                  SECTOR_10 + i + (j * RECORD), sector[i]);
            checksum ^= sector[i];
         }
 }
```

A fenti kódrészlet bemutatja, hogy a könyvtár megnyitása után a f_readdir fügvénnyel beolvassuk a fájl nevét, majd a Fat_Read-el az egész fájlt a Fatfs-ből, és a belső ciklusban bájtonként kimentjük az adott szektorba. Itt számoljuk még az ellenőrző összeget is. A függvény végén növeli a sequnce number-t és elmenti azt és az ellenőrző összeget is a flashbe.

A belső flash memóriában való tárolás kialakítását nehezítette, hogy nem találtam jól használható tutorialt hozzá. Ha valahol elakadtam, akkor különböző fórumokon keresgéltem addig, amíg nem találtam a saját problémámhoz hasonlót, kivitelezhető megoldással. Ezzel nagyon sok idő ment el, de a végére egyre gyorsabban sikerült a megfelelő forrásokat megtalálni.

Tapasztalataim alapján nem ajánlom, egy ilyen nagy szektorméretttel rendelkező mikrokontrollernek, a flash memóriájában a Fatfs kialakítását. Ha ekkorák a szektorok akkor valószínűleg nincs belőlük elegendő, a Fatfs minimális szektor számához. Ha mégis lenne akkor a Fatfs által megengedett maximális szektorméret az 4 kbyte, tehát ilyenkor az egyes flash szektoroknak (128 kbyte-os mérettel számolva) csak a 3 %-át töltenénk fel, ami elég pazarló.

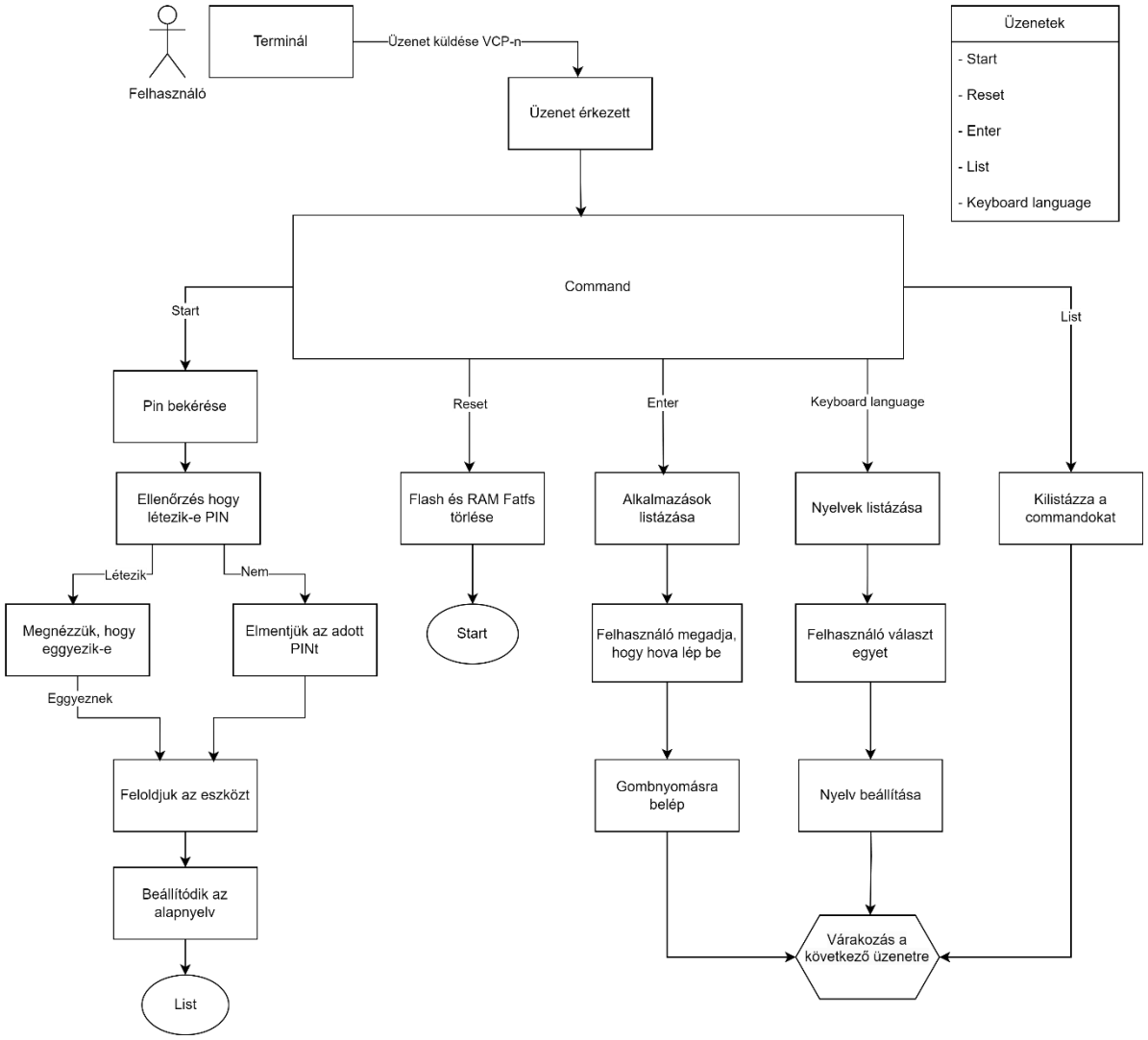
USB CDC

Az USB CDC forrásfájlján nem kellett sokat változtatnom. A Composite Device úgy van kialakítva, hogy egyszerre több soros porton is tud kommunikációt folytatni, viszont a felhasználói interakciók kezelésére elegendő egy is. A két fő függvény, amit használok a CDC_Transmit és a CDC_Receive.

```
static int8_t CDC_Receive(uint8_t cdc_ch, uint8_t *Buf, uint32_t *Len)
{
  message = 1; // signal the arrival of a user control message

  USBD_CDC_SetRxBuffer(cdc_ch, &hUsbDevice, &Buf[0]);
  USBD_CDC_ReceivePacket(cdc_ch, &hUsbDevice);

  memset(usb_RX_Buff, '\0', 64);  // Clear buffer before receiving
  memcpy(usb_RX_Buff, Buf, *Len);
  memset(Buf, '\0', 64);

  return (USBD_OK);
}
```

A Receive függvény, ha érkezett üzenet, akkor állítja a message flag-et, ezzel jelez a főprogramnak. Fogadáskor az usb_RX_Buff-ba érkezik a maximum 64 bájtos üzenet. Figyelni kell a buffer ürítésére, mivel ha fogadunk először egy hosszabb üzenetet, majd egy rövidebbet, és nem ürítettünk, akkor a rövidebb üzenet összemosódik a hosszabbiknak a végével.

Főprogram



5. ábra: Alkalmazás működésének folyamatábrája

A folyamatábráról látható, hogy a program a soros porton érkező vezérlő üzenetre reagál. Az eszköz csatlakoztatása és a soros port megnyitása után kell egy start üzenetet küldeni.

```
start
Beginning the start configuration.
Enter a PIN code, which must consists of 4 number
digits.
1234
PIN saved and device unlocked successfully!

Default language is Hungarian.
Listing available commands:
l : Lists the control commands.
e : Enters the desired website.
r : Resets the device. Deletes everything!
k : Set the keyboard language.
```

6. ábra: Start parancs a soros porton

Ezáltal hívódik meg a Start_Device függvény, ami először kiolvassa a flashből a PIN-t, ha a flash-ben értelmes PIN-t talált, akkor összehasonlítja a felhasználó által megadottal.

```
for(uint8_t i = 0; i < 4; i++){
	uint8_t c = usb_RX_Buff[i];
	if ('0' <= c && c <= '9')
		*pin += (uint16_t) (c - '0') * (uint16_t) pow(10, 3 - i);
```

```
        else
            return FAILURE;
}
```

Itt a felhasználó által megadott karakterek PIN kóddá való konvertálása igényel némi magyarázatot. A c változóban beolvasott ASCII karakterekből egy négyjegyű számot szeretnénk összeállítani, ehhez a karakterekből ki kell vonni '0'-át, így megkapjuk a számszerű értéket. Ezeket a számokat aztán helyiérték helyesen össze kell adni, és megkapjuk a PIN kódot. Amennyiben még nem volt PIN kód eltárolva, a felhasználó megad egyet, ez az előbb említett módszerrel lesz áttranszformálva és eltárolásra kerül a flash-be. Aztán ha a PIN kód ellenőrzését végrehajtó függvények sikerrel jártak, akkor beállításra kerül az alapértelmezett magyar nyelv a billentyűzetnek, végül pedig listázódnak az elérhető vezérlő üzenetek.

A listázást alapját a Transmit függvény adja, amely a soros portra küldi a kapott üzenetet, a CDC_Transmit felhasználásával.

```
Changing language:
e : ENGLISH
h : HUNGARIAN
e
Language changed to English!
```

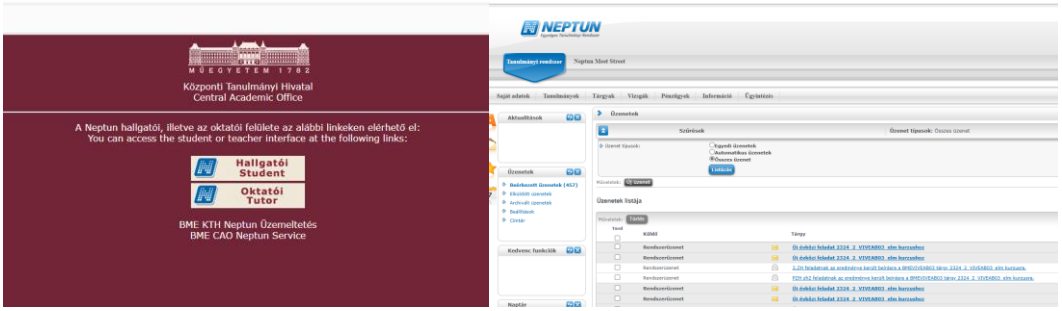
7. ábra: Billentyűzet nyelvének megváltoztatása

A nyelvcsere viszonylag egyszerűen működik. Soros porton keresztül kiküldjük a az elérhető nyelvek listáját, aztán a felhasználó bemenete alapján, beállítjuk a nyelvet, ha van ilyen opció.

```
r
Starting the erasure.

Flash memory has been erased.

Soft resetting the device...
```

8. ábra: Adatok törlése

A reset parancsot kiszolgáló függvény először meghívja a Flash_Sector_Erase függvényt, paramétereként pedig megadja az összes olyan flash memória szektort, ahol rekordokkal kapcsolatos adat van. A flash törlése után meghívásra kerül a NVIC_SystemReset függvény, ami törli a teljes RAM tartalmát, így kiürül a Fatfs és az MSC-n keresztül üresnek látjuk az eszközt.




```
e
Select the website you want to enter:
MOODLE.TXT
FACEBOOK.TXT
NEPTUN.TXT
NEPTUN.TXT
Press the button to transmit the password!

Password successfully typed.
```

9. ábra: Jelszó beírása

A jelszó beírásának első lépése a Fatfs-be található összes fájl nevének kilistázása a VCP-n keresztül a felhasználónak. Erre a felhasználó választ egyet ezek közül azt visszaírja. Ekkor ténylegesen az egész rekordot kiolvashatjuk a Fatfs-ből.

```
uint8_t *tilde_pos = strchr(password, '~');
if (tilde_pos != NULL) {
	memmove(password, tilde_pos + 1, strlen(tilde_pos) + 1);
	ret = OK;
}
```

A rekordot itt még formázni kell. Le kell vágni a nevét (a fenti példában a NEPTUN.TXT-t), mivel ez nem fog beírásra kerülni. A név végét egy '~' karakter jelzi így ha azt megtaláltuk, akkor az egész rekordot „előrébb csúsztatjuk” a név helyére. Ez után készen állunk a küldésre, de még meg kell várni, hogy a felhasználó megnyomja a gombot. Ha a gomb interrupt jelzett a button flagen keresztül, akkor meghívódik a Send_Keystroke függvény paraméterében a felhasználónévvel és a jelszóval.

A billentyűleütéseket elküldő függvényben figyelni kell arra, hogy a magyar karakterek ASCII kódja nem csak egy karakter, hanem kettő. Hasonló a helyzet az ENTER-el is amit egy újsor és kocsi vissza ad ki. Ezeknél a különleges eseteknél nem küldöm ki kétszer ugyanazt a karaktert. Az ASCII kódokat billentyű kódokká a Convert_char_to_Keystroke nevű függvény alakítja.

```
static USBD_StatusTypeDef Convert_char_to_Keystroke(uint8_t ascii) {
	for (uint8_t i = 0; i < Length; i++) {
		if (selectedKeyboard[i].ascii_code == ascii) {
			keycodes.MODIFIER = selectedKeyboard[i].mode;
			keycodes.KEYCODE1 = selectedKeyboard[i].keycode;
			return USBD_OK;
		}
	}

	// There are no matching characters
	keycodes.MODIFIER = 0;
	keycodes.KEYCODE1 = 0;
	return USBD_FAIL;
}
```

Ez a függvény karakterenként kapja a felhasználónevet és a jelszót, és az aktuálisan kiválasztott nyelvű billentyűzethez tartozó tömböt, indexeli a paraméterbe kapott ASCII karakterrel. Az így kapott billenytűkód az USB HID-en keresztül kerül elküldésre.

```
if(USBD_HID_Keyboard_SendReport(&hUsbDevice,&keycodes,
                                sizeof(keycodes))!= USBD_OK)
    return USBD_FAIL;
```

Fontos hogy gyorsan küldjünk egy csupa 0-ából álló üzenetet is, ezzel engedjük fel az adott billentyűt. Ha küldéskor ENTER billentyűt küldünk, akkor utána be van iktatva mesterségesen késleltetés, hogy az oldalnak/alkalmazásnak ahova szeretnénk bejelentkezni legyen ideje betölteni. A küldés végét egy `'~'` karakter jelzi, ezt már nem fogjuk elküldeni. Ha sikerült a teljes küldés, azt a kommunikációs csatornán jelezzük a felhasználónak.

Összefoglalás:

Az eszköz implementálásán még lehet javítani. Az egyik ilyen pont a felhasználói interakció kezelése, lehetne egy grafikus felülettel rendelkező vastag kliens vagy webes alkalmazás. Ez nagyban könnyítené a felhasználónév, jelszó és egyéb szükséges TAB-ok, ENTER-ek megadását, emellett a parancsok küldése is látványosabb lenne.

Ami pedig sajnos idő hiányában teljesen kimaradt az a titkosítás. A jelszavakat semmiképpen sem lenne szabad plain-text-ben tárolni. Mondjuk chacha20-al. Generálunk egy titkos kulcsot és egy nonce-ot, ezekkel inicializáljuk a chacha algoritmust. A generált pszeudó randomszám sorozattal kellene XOR-olni az egyes rekordokat és így megkapnánk őket titkosítva.

A feladat során megtanultam használni a CubeIDE fejlesztői környezetet, és az STM32F4-es családdal is sikerült jobban megismerkednem. Az USB miatti időzítés függő debuggolást is gyakoroltam, és most már magabiztosabban használom a különböző USB eszközosztályokat. A flash kezeléshez/törléséhez pedig kipróbáltam az STM32CubeProgrammer-t. https://www.st.com/en/development-tools/stm32cubeprog.html