Arduino ftp server i serijska komunikacija sa C# aplikacijom

Sadžaj

1 Potrebni alati i povezivanje

1.1 Potrebni delovi

1. Arudino mikrokontroler
2. Arudino modul za čitanje SD kartica
3. 6 žica sa golim krajevima
4. Opcionalno: lemilica

1.2 Potrebna programska okruženja

1. Arduino IDE
   Uputstvo za instalaciju: https://www.Arduino.cc/en/software
   Takođe koristili smo i SdFat biblioteku verzija: 2.0.1.
   Biblioteku možemo instalirati direktno u Arduino IDE: [Tools]->[Manage Libraries]...->pretražujemo i instaliramo biblioteku sa nazivom SdFat autora Bill Greiman verziju 2.0.1.
2. .NET Core
   Koristili smo .NET Core SDK 3.1.407
   Uputstvo za instalaciju: https://docs.microsoft.com/en-us/dotnet/core/install/
   Takođe koristili smo i System.IO.Ports paket koji možemo da instaliramo izvršavajući sledeću komandu:
   dotnet add package System.IO.Ports –version 5.0.1

1.3 Povezivanje Arudina i modula za čitanje SD kartica

Povezali smo pinove na sledeći način:

1. pin10 CS
2. pin11 MOSI

3. pin12 MISO
4. pin13 SCK
5. GND GND
6. 5V VCC

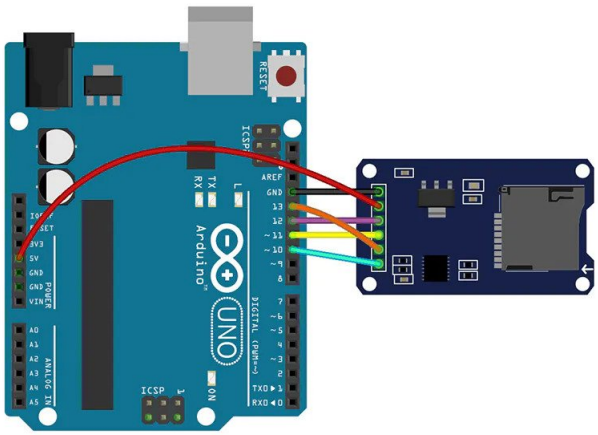

Figure 1: povezivanje

1.4 Sinhronizacija protoka tunela

Postavli smo brzinu protoka serijskog čitanja i pisanja na 9600 baud.

2 Opis klijentskog C# koda

2.1 Opis glavnog C# fajla: Program.cs

U fajlu *Program.cs* možemo videti primer upotrebe klijentskog koda pomoću kojeg možemo da šaljemo komande prema Arduinu i tumačimo rezultate izvršenih komandi.

Potrebno je da imamo u istom okruženju (*namespace*-u) sledeće *.cs* fajlove:

1. ArduinoCommands.cs
2. ArduinoInterpreter.cs
3. ArduinoIW.cs
4. ArduinoMessageBuilder.cs
5. CrcMath.cs
6. ErrorsInternal.cs
7. PortTalk.cs
8. TalkBuffer.cs

U $Main$ metodi pokrećemo nit u kojoj čekamo, čitamo i obrađujemo dostupne podatke koji su primljeni sa komunikacionog tunela. Definišemo port preko kojeg želimo da komuniciramo sa Arduinom. Koristimo objekat klase $SerialTalk$ koji sadrži osnovne metode za komunikaciju preko tunela.

Nakon inicijalizacije i otvaranja tunela na odgovarajućem portu pomoću metode $setup()$ prosleđujemo tunel dalje kao argument u konstruktoru:

1. objekta klase $ArduinoIW$ koji koristimo za interpretaciju već pročitanih podataka sa tunela odnosno za interpretaciju podataka koji su spremni za obrađivanje. Takođe objekat klase $ArduinoIW$ koristimo i za pisanje podataka na prosleđeni tunel. U ovoj klasi ne obavljamo čitanje sa tunela.
2. objekta klase $TalkBuffer$ koji koristimo za čuvanje pročitanih podataka sa tunela.

U glavnom programu pratimo da li objekat klase $CommandObject$ koji predstavlja trenutni rezultat izvršene komande zavisi od objekta koji predstavlja rezultat sledeće izvršene komande. Indikator smo nazvali $waitingForTheChain$. Primer: komanda GET bez poslatog opcionalnog argumenta **crc** zavisi od sledeće izvršene komande. U ovom slučaju automatski šaljemo sledeću komandu sa nazivom CRC i argumentima koji predstavljaju fajl-segment nad kojim je izvršena GET komanda.

Zavisnosti između dve uzastopne komande možemo da dodefinišemo (pogledati podsekciju *ArduinoCommands.cs*) .

Nakon inicijalizacije potrebnih i gore navedenih objekata izvršavamo $while$ petlju. U $while$ petlji očekujemo vrednost $true$ kao rezultat poziva metode $isavailable()$ nad

objektom - baferom $TalkBuffer$. Za dodatna uputstva oko čitanja podatka sa tunela pogledati sekciju *TalkBuffer.cs*.

Ukoliko imamo dostupne podatke sa bafera za dalje obrađivanje, podatke obrađujemo pomoću objekta koji predstavlja klasu *ArduinoIW.cs*. Za dodatna uputstva pogledati podsekciju *ArduinoIW.cs*. Obrađivanje vršimo pomoću metode $interpet()$. Ukoliko je obrađivanje bilo validano i rezultat ove metode predstavlja kompletiran rezultat izvšene komande proveravamo da li je rezultat zavistan od rezultata sledeće izvršene komande. U slučaju zavisnosti koristimo odgovarajuće provere da bi smo ispitali kompatibilnost između izvršenih komandi ili u slučaju nezavisnosti ispisujemo samo trenutni objekat.

Ukoliko rezultat nije validan preskačemo ispisivanje i premotavamo bafer (objekat klase $TalkBuffer$) na kraj ili do prvog $ simbola. $ simbol predstavlja kraj poruke. Takođe potrebno je da resetujemo graditelja koji sve vreme gradi rezultat izvršavanja komande i koji se nalazi u objektu koji predstavlja *ArduinoIW.cs* klasu.

Na kraju čuvamo rezultat trenutno izvršene komande u objektu koji predstavlja *ArduinoIW.cs* klasu i ukoliko smo obradili ceo bafer očistimo ga $sb.clear()$.

2.2 ArduinoCommands.cs

Struktura fajla

```
(a)CommandObject
--->CommentCommandObject
--->DirCommandObject
--->ErrorCommandObject
--->MdCommandObject
--->CdCommandObject
--->PutCommandObject
--->GetCommandObject
--->DelCommandObject
--->CrcCommandObject
```

Opis

U fajlu $ArduinoCommands.cs$ definišemo validiranje pojedinačnih delova rezultata izvršene komande.

Opšti šablon rezultata izvršene komande može da izgleda:

1. header-a deo:
   - $\#OK\ arg1\ arg2$
   - ili $\#ERROR\ arg1\ arg2$
2. payload deo, sadrži linije koje počinju simbolom $>$. Primer:

   ```
   > file1 2
   > file2 10
   ```

3. footer deo, sadrži jednu reč, reč $\#END$ kojom označavamo da smo dobili celu poruku. Nakon reči $\#END$ u sledećem redu sledi simbol $ kojim označavamo da je Arduino spreman da prima nove komande.

Konkretan objekat koji dopunjuje i predstavlja apstraktnu-klasu $CommandObject$ koristimo za predstavljanje rezultata izvršene komande.
U njemu deklarišemo i definišemo:

1. promenljive u kojima skladištimo prosleđene argumente za ponuđene parametre
2. da li je kompletiran
3. da li je validan
4. da li je zavistan od rezultata izvršavanja sledeće komande
5. metode u kojima obrađujemo linije payload-a
6. ostale pomoćne metode kao što su ispis rezultata i ostalo

Sada možemo detaljnije opisati i klase koje implementiraju ovu apstraktnu klasu. Za sve podržane komande na Arduinu imamo odgovarajuće klase koje koristimo za validaciju primljenog rezultata odgovarajuće izvršene komande. Implementirali klase:

1. $CommentCommandObject$
   obrađujemo komentar poslat sa Arduina.
2. $DirCommandObject$
   obrađujemo rezultat izvršavanja dir komande.
3. $ErrorCommandObject$
   obrađujemo rezultat izvršavanja er komande.
4. $MdCommandObject$
   obrađujemo rezultat izvršavanja md komande.
5. $CdCommandObject$
   obrađujemo rezultat izvršavanja cd komande.
6. $PutCommandObject$
   obrađujemo rezultat izvršavanja put komande.
7. $GetCommandObject$
   obrađujemo rezultat izvršavanja get komande.
8. $DelCommandObject$
   obrađujemo rezultat izvršavanja del komande.
9. $CrcCommandObject$
   obrađujemo rezultat izvršavanja crc komande.

U klasi $ArduinoCommands$ nabrojali smo nazive komandi.
Objekat klase $CommandObject$ sadrži i metode za definisanje zavisnosti između rezultata trenutno izvršene komande i drugog $CommandObject$ objekta koji će predstavljati rezultat sledeće izvršene komande. Koristimo metode:

1. (virtuelnu) $checkChainCmdResult$ u kojoj možemo dodefinisati način provere zavisnosti
2. $getChainCommandName$ za vraćanje naziva komande od koje zavisi $this$ objekat

3. $checkChainCommandName$ za proveru podudarnosti naziva komandi

4. $isWaitingForTheChain$ u kojoj definišemo da li objekat koji predstavlja rezultat trenutno izvršene komande poseduje zavisnost sa objektom koji predstavlja rezultat sledeće izvršene komande

2.3 ArduinoIntepreter.cs

Struktura fajla

```
(a)ArduinoLineProccesor
--->HeaderLineProccessor
--->ResultLineProccessor
--->FooterLineProccessor
ProccessorObjectResult
ArduinoInterpreter
```

Opis

U fajlu $ArduinoIntepreter.cs$ definišemo klase i metode pomoću kojih obrađujemo poruku koju smo izgradili od podataka koji su se nalazili na baferu. Kao bazu imamo apstraktnu klasu $ArduinoLineProccesor$ u kojoj deklarišemo apstraktni metod $process$ u kojem konkretne klase navode način obrađivanja svog dela poruke. Imamo obrađivače za sve delove poruke:

1. $HeaderLineProccessor$
   obrađujemo liniju poruke koja bi trebala da predstavlja header.

2. $ResultLineProccessor$
   obrađujemo liniju poruke koja bi trebala da predstavlja liniju payload-a.

3. $FooterLineProccessor$
   obrađujemo liniju poruke koja bi trebala da predstavlja footer.

Kao klasu vodilju imamo klasu $ArduinoInterpreter$ u kojoj obrađujemo celokupnu poruku koja je sastavljena od podataka sa bafera. Prvo obrađujemo header, nakon toga payload i na kraju footer i vraćamo odgovarajući rezultat.

Koristimo objekat klase $ProccessorObjectResult$ kao internu enkapsulaciju objekta klase $CommandObject$. Svrha klase $ProccessorObjectResult$ je kompatibilnije prenošenje rezultata između više obrađivača celokupne poruke.

2.4 ArduinoIW.cs

Struktura fajla

```
ArduinoIW
```

Opis

U ovom fajlu definišemo klase i metode za obrađivanje rezultata izvršene komande i slanje nove komande ka Arduinu. Na početku imamo objekat klase $ArduinoIW$ čijem konstruktoru prosleđujemo objekat klase $PortTalk$ u kojem definišemo tunel preko kojeg možemo da šaljemo podatke ka Arduinu. Korišćenjem metode $WriteLine$

možemo poslati liniju teksta ka Arduinu i metodom $interpret$ možemo da obradimo rezultat izvršene komande. U metodi $interpret$ koristimo objekat klase $ArduinoInterpret$ u kojoj se nalaze odgovarajuće metode za obrađivanje pojedinačnih delova poruke. Najpre uzmemo sve podatke koje smo pročitali sa ($PortTalk$) tunela i sačuvali u ($TalkBuffer$) bafer i proveravamo da li trenutni podaci zadovoljavaju opšti šablon validne poruke. Celokupnu poruku gradimo pomoću objekta klase $ArduinoMessageBuilder$.

Ukoliko je zadovoljen opšti šablon validne poruke odnosno ukoliko pozivom metode $isComplete()$ nad objektom klase $ArduinoMessageBuilder$ dobijemo povratnu vrednost $true$ prosleđujemo celokupnu poruku na dalje obrađivanje. Sledeći korak obrade predstavlja korak građenja objekta koji predstavlja rezultat izvršene komande. Podržane komande možemo videti u sekciji $ArduinoCommands.cs$. Za ovaj korak koristimo metodu $proccess()$ objekta klase $ArduinoInterpreter$. Ukoliko uspešno obradimo podatke metodom $proccess()$ vratićemo objekat koji predstavlja rezultat izvršene komande. U suprotnom vraćamo vrednost $null$. U ovom slučaju povratna vrednost signalizira da nismo uspešno uspeli da validiramo podatke i da je potrebno da resetujemo graditelja celokupne poruke pomoću $resetResultBuilder$.

Takođe ukoliko objekat koji predstavlja rezultat izvršene komande nije validan ili je kompletiran potrebno je isto da resetujemo graditelja celokupne poruke.

U objektu ove klase ($ArduinoInterpreter$) čuvamo objekat koji predstavlja rezultat prethodno izvršene komande. Koristimo $getPreviousCommandObject$ i $setPreviousCommandObject$ metode za postavljanje i vraćanje objekta koji predstavlja rezultat prethodno izvršene komande.

2.5 ArduinoMessageBuilder.cs

Struktura fajla

```
(a)MessagePartitionBuilder
--->HeaderPartitionBuilder
--->PayloadPartitionBuilder
--->FooterPartitionBuilder
--->ResultMessageBuilder
```

Opis

U ovom fajlu definišemo klase i metode za gradnju celokupne poruke koja bi trebala da predstavlja rezultat izvršene komande. Celokupna poruka sastoji se iz header-a, payload-a i footer-a. Imamo odgovarajuće graditelje i u svakom od njih različita stanja u kojima mogu biti. U graditeljima ne ispitujemo validnost poruke već samo popunjavamo prazna mesta definisana šablonom opšte poruke. Imamo sledeće graditelje i stanja:

1. HeaderPartitionBuilder i stanja:
   (a) STATUS_MODE, stanje u kojem očekujemo prvu reč (reči su razdvojeni razmakom) koja bi trebala da opisuje da li se komanda uspešno izvršila ili ne.
   (b) CMD_NAME_MODE, stanje u kojem očekujemo drugu reč koja bi trebala da opisuje naziv komande koju smo pokušali da izvršimo.
   (c) FILE_PATH_MODE, stanje u kojem očekujemo treću reč koja bi trebala da opisuje fajl nad kojim smo pokušali da izvršimo komandu.

2. PayloadPartitionBuilder i stanja:

   (a) LINE_MODE, stanje u kojem očekujemo liniju payload-a

3. FooterPartitionBuilder i stanja:

   (a) STATUS_MODE, stanje u kojem očekujemo prvu reč koja bi trebala da se poklapa sa stringom $\#END$.

U svakom od graditelja gradimo sve dok ne naiđemo na liniju koja počinje simbolom koji predstavlja sledeću particiju poruke.

Kao glavnog dirigenta imamo objekat klase $ResultMessageBuilder$. Objekat klase $ResultMessageBuilder$ takođe sadrži metodu $build$ kojoj prosleđujemo po jednu liniju na obrađivanje. U metodi $build$ prosleđujemo dalje liniju odgovarajućem graditelju. Objekat klase $ResultMessageBuilder$ može biti u stanju:

1. HEADER_MODE, stanje predstavlja da trenutno gradimo header deo poruke
2. PAYLOAD_MODE, stanje predstavlja da trenutno gradimo payload deo poruke
3. FOOTER_MODE, stanje predstavlja da trenutno gradimo footer deo poruke
4. MSG_COMPLETE_MODE, stanje predstavlja da smo uspešno izgradili celu poruku i naišli na simbol $. Simbol $ predstavlja indikator da je Arduino spreman da prima nove komande.

2.6 PortTalk.cs

Struktura fajla

```
(a)PortTalk
--->SerialTalk
```

Opis

U ovom fajlu definišemo apstraktnu klasu $PortTalk$ u kojoj deklarišemo osnovne apstraktne metode za komunikaciju preko tunela. Klase koje nasleđuju $PortTalk$ definišu način komunikacije preko tunela. Imamo konkretnu klasu $SerialTalk$. U klasi $SerialTalk$ nasleđujemo i definišemo sve apstraktne metode iz baze klase $PortTalk$ i koristimo objekat klase $SerialTalk$ za komunikaciju preko serijskog porta. Metode smo definisali na sledeći način:

1. U metodi $setup()$ inicijalizujemo tunel. Koristimo objekat klase $SerialPort$ koji se nalazi u paketu System.IO.Ports. Koristili smo verziju paketa 5.0.1.

   (a) Navodimo COM tunela: /dev/ttyACM0

   (b) Brzinu protoka: 9600

   (c) i otvaramo tunel.

   (d) Nakon poziva metode $setup()$ spremni smo za minipulisanje podacima preko tunela.

2.7 TalkBuffer.cs

Struktura fajla

`TalkBuffer`

Opis

U ovom fajlu definišemo klasu i metode pomoću kojih čuvamo pročitane podatke sa tunela. Objekat klase $TalkBuffer$ predstavlja "ne školski" omotač nad objektom klase $PortTalk$. U konstruktoru prosleđujemo objekat klase $PortTalk$ odnosno tunel sa kojeg čitamo podatke i smeštamo u naš objekat klase $TalkBuffer$. Koristimo metodu $loop()$ za čitanje i čuvanje podatka sa tunela. Definisali smo stop vremenski interval za čitanje na 50ms. Ukoliko je vremenska razlika između dva trenutka u kojima imamo dostupne podatke na tunelu veća od 50ms postavljamo indikator (promenljiva $finished$) da imamo podatke koji su spremni za dalje obrađivanje. Objekat klase $TalkBuffer$ poseduje i ostale metode i promenljive za manipulisanje baferom:

1. promenljivu $position$ koristimo da označimo na kojem mestu u baferu se trenutno nalazimo
2. metodu $read()$ koristimo za čitanje bafera na $position$ poziciji
3. metodu $peek(int\ offset)$ koristimo za vraćanje karaktera u baferu koji se nalazi na $position + offset$ poziciji
4. metodu $isnext()$ koristimo da proverimo da li smo naišli na kraj bafera
5. metodu $isnext(int\ offset)$ koristimo da proverimo da li $position + offset$ pozicija je validna pozicija u baferu
6. metodu $isavailable()$ koristimo da proverimo da li imamo podatke u baferu koji su spremni za dalje obrađivanje
7. metodu $clear()$ koristimo za vraćanje na početno stanje

2.8 CrcMath.cs

Struktura fajla

`CrcMath`

Opis

U ovom fajlu definišemo metode za izračunavanje CRC16 vrednosti za prosleđen string ili celobrojni broj. Algoritami koji koristimo za izračunavanje CRC16 vrednosti se poklapa sa algoritmom koji je trenutno u upotrebi na Arduino platformi.

2.9 ErrorsApp.cs

Struktura fajla

`ErrorsApp`

Opis

U ovom fajlu definišemo strukture za čuvanje grešaka. Greške mogu da se jave:

1. interno odnosno u procesu validiranja poruke koja predstavlja rezultat izvršene komande poslatog od strane Arduina
2. eksterno odnosno greške koje su nastale na Arduinu u procesu obrađivanja poslate komande

Koristimo globalno statičnu promenljivu $errno$ u kojoj smeštamo celobrojnu vrednost trenutne greške. U bilo kojem delu koda možemo da pozovemo statičnu metodu $ErrorsApp.get()$ kojom vraćamo vrednost tipa $string$ kojom opisujemo trenutnu grešku. Podržane greške čuvamo u niz sa nazivom $errors$. Rezervisali smo:

1. od [0-9] mesta za interne greške. Interne greške čuvamo u $enum$ sa nazivom $ErrnoInternalCodes$. Prva greška ima vrednost 0 i svaka naredna za po jednu veću. Lista sadrži:
   (a) $STATUS\_PROCCESSOR\_ERROR\_NOT\_VALID\_HEADER$ - greškom označavamo da header poruke nije validan odnosno ne ispunjava šablon validnog header-a.
   (b) $STATUS\_PROCCESSOR\_ERROR\_NOT\_VALID\_PAYLOAD$ - greškom označavamo da payload poruke nije validan odnosno ne ispunjava šablon validnog payload-a.
   (c) $STATUS\_PROCCESSOR\_ERROR\_NOT\_VALID\_FOOTER$ - greškom označavamo da footer poruke nije validan odnosno ne ispunjava šablon validnog footer-a.
   (d) $STATUS\_PROCCESSOR\_ERROR\_UNKNOWN\_CMD$ - greškom označavamo da header poruke za drugi parametar (naziv komande koja je izvršena) sadrži argument naziv komande koja nije podržana.
   (e) $RESULT\_PROCCESSOR\_ERROR\_COMMAND\_PARSE$ -greškom označavamo da payload poruke nije validan odnosno ne ispunjava šablon validnog payload-a.
   (f) $RESULT\_PROCCESSOR\_ERROR\_RESULT\_LINE\_MISSING$ - greškom opisujemo konkretnije da linija koja predstavlja deo payload poruke ne počinje simbolom $>$.
2. od [10-99] mesta za eksterne greške. Eksterne greške čuvamo u $enum$ sa nazivom $ErrnoExternalCodes$. Prva greška ima vrednost 10 i svaka naredna za po jednu veću. Lista sadrži:
   (a) od [10-49] predstavljaju mesta za greške koje su nastale prilikom obrade poslate komande na Arduinu:
      i. $CMD\_ERR\_GENERALERROR$ predstavlja opštu grešku
      ii. $CMD\_ERR\_MISSINGARGUMENT$ greška ukazuje na nedostatak argumenta za odgovarajući parametar
      iii. $CMD\_ERR\_EXTRAARGUMENT$ greška ukazuje na prekoračenje broja prosleđenih argumenta
      iv. $CMD\_ERR\_INVALIDVALUE$ greška ukazuje da prosleđeni argument nije validan

v. $CMD\_ERR\_VALUEOUTOFRANGE$ greška ukazuje da prosleđeni argument se ne nalazi u skupu dozvoljenih vrednosti

vi. $CMD\_ERR\_UNKNOWNCOMMAND$ greška ukazuje da prosleđeni argument koji odgovara parametru naziva komande koju želimo da izvršimo predstavlja naziv komande koja nije podržana na Arduinu

(b) ostalih [50-99] grešaka predstavljaju greške koje su nastale prilikom manipulisanja podacima na Arduinu prilikom obrade poslate komande.

3 Pokretanje programa i uključivanje projektnog koda u ostalim projektima

3.1 Uputstvo za pokretanje programa

1. Otvorimo .ino fajl u Arduino IDE koji se nalazi u folderu ArduinoFiles/Program.
2. Presnimimo program na Arduino mikrokontroleru.
3. Pokrenimo program kojim možemo da čitamo i interpretiramo rezultate izvršene komande poslate sa Arduina. Program se nalazi u folderu NETCore/Program/ReadProgram i možemo ga pokrenuti pomoću dotnet komande: dotnet run
4. Pokrenimo program kojim možemo da šaljemo komande ka Arduinu. Program se nalazi u folderu NETCore/Program/WriteProgram

Sada možemo da šaljemo komande ka Arduinu i čitamo rezultate izvršene komande iz dva nezavisna programa.

3.2 Uputstvo za korišćenje projektnog koda u ostalim projektima

3.2.1 Arduino projektni kod

Potrebno je da obezbedimo i uvezemo odgovarajuće fajlove koji su opisani u sekciji 5.1 sa nazivom *Opis glavnog Arduino koda - SerijskiKomandniProcesorSDCARD.ino*.

U glavnom programu uvozimo sledeće fajlove:

- cmdProc.h
- buf.h
- sdCardManipulator.h

U sekciji 5.1 možemo videti ideju upotrebe klasa, metoda i promenljivih koji su sadržani u gore navedim fajlovima.

3.2.2 .NET projektni kod

Potrebno je da obezbedimo i uvezemo u isto okruženje sve fajlove koji su opisani u sekciji 2.1 sa nazivom *Opis glavnog C# koda - Program.cs*.

Takođe potrebno je:

1. Da dodamo u .csproj fajl sledeće linije:

```
<ItemGroup>
<ProjectReference Include=
"..\\NETCore\Program\ReadProgram\coreapp.csproj" />
</ItemGroup>
```

2. i uvezemo paket $System.IO.Ports$:

```
<ItemGroup>
<PackageReference Include="System.IO.Ports" Version="5.0.1" />
</ItemGroup>
```

4 Uputstvo za korišćenje komandi

4.1 Značenje simbola

$	Nakon izvršavanja tekuće naredbe šaljemo da smo spremni za prijem nove naredbe i šaljemo simbol $.
#END	Signaliziramo korisniku da je prethodni ispis završen.
@	Poruke informativnog karaktera sadrže kao prvi znak "@".
>	Linije payload-a imaju prvi znak ">".
#	Linije koje počinju ovim simbolom označavaju status.

Table 1: Značenje simbola

Prilikom slanja komande ka Arduinu navodimo pored naziva komande i argumente za odgovarajuće parametre koji su podržani za navedenu komadnu.

4.2 Povratne vrednosti u opštem slučaju

- Slučaj greške

```
#ERROR <command-type> ERROR_CODE
>
#END
$
```

$ERROR\_CODE$ predstavlja celobrojni broj kojim opisujemo grešku nastalu pri izvršavanju $< command - type >$ komande.

- Uspešno izvršavanje

```
#OK <command-type> <item-path>
> <function-results-list>
#END
$
```

$< command - type >$ predstavlja naziv komande koju smo uspešno izvršili i $< item - path >$ predstavlja putanju fajla ili foldera nad kojim smo uspešno izvršili komandu.

4.3 Lista podržanih komandi i povratne vrednosti

4.3.1 HELP

Koristimo za vraćanje uputstva za korišćenje raspoloživih komandi.
Upotreba

```
help
```

Povratne vrednosti

- Uspešno izvršavanje

```
#OK help <cwd>
> <available-commands-list>
$
```

Primer:

```
[\folder1\folder2]: help
#OK help \folder1\folder2
> <available-commands-list>
$
```

4.3.2 DIR

Koristimo za vraćanje liste fajlova sa dužinama iz tekućeg foldera. Prilikom uspešnog izvršavanja vraćamo $\#OK$ i ime tekućeg foldera i zatim u narednim redovima listu fajlova sa dužinama. Sa Arduina šaljemo više redova rezultata i svaki red ima ">" kao prefiks.
Upotreba

```
dir
```

Povratne vrednosti

- Slučaj greške

```
#ERROR dir ERROR_CODE
>
$
```

50	SdFat open: Unable to open the current working directory

Table 2: Značenje kodova grešaka

- Uspešno izvršavanje

```
#OK dir <folder-path>
><folder-listing>
$
```

Primer:

```
[\folder1\folder2]: dir
#OK dir \folder1\folder2
>10 fajl1
>13 fajl2
>0 folder3\
$
```

4.3.3 CD

Vraćamo tekući folder sa poslatom CD komandom bez argumenta ili zadajemo tekući folder ukoliko je poslata komanda sa odgovarajućim argumentom. Ukoliko je argument "/" postavljamo tekući folder na root.
Upotreba:

```
cd [<relative-path>]
```

Povratne vrednosti

- Slučaj greške

```
#ERROR cd ERROR_CODE
>
$
```

52	SdFat chdir: Can't change to the folder (Check if the folder exists)

Table 3: Značenje kodova grešaka

- Uspešno izvršavanje

```
#OK cd <cwd-path>
>
$
```

Primer:

```
[\folder1\folder2]: cd folder3
#OK \cd folder1\folder2\folder3
>
$
```

4.3.4 MD

Kreiramo novi folder unutar tekućeg foldera. Vraćamo $\#OK$ i ime foldera ako prilikom uspešnog izvravanja ili $\#ERROR$ ukoliko se javila greška.
Upotreba

```
md <folder-name>
```

Povratne vrednosti

- Slučaj greške

```
#ERROR md ERROR_CODE
>
$
```

53	SdFat exists: File with the same name already exists.
55	SdFat mkdir: Can't make a folder

Table 4: Značenje kodova grešaka

- Uspešno izvršavanje

```
#OK md <folder-path>
>
$
```

Primer:

```
[\folder1\folder2]: md folder4
#OK md \folder1\folder2\folder4
>
$
```

4.3.5 DEL

Brišemo zadati folder ili fajl unutar tekućeg foldera. Ako ne možemo da obrišemo vraćamo grešku. Vraćamo $\#OK$ i ime fajla prilikom uspešnog izvršavanja ili $\#ERROR$ ako se javila greška.
Upotreba

```
del <item-name>
```

Povratne vrednosti

- Slučaj greške

```
#ERROR del ERROR_CODE
>
$
```

54	SdFat exists: File doesn't exists
50	SdFat open: Can't access to the file
56	SdFat rmRfStar: Can't delete the folder
57	SdFat remove: Can't delete the file

Table 5: Značenje kodova grešaka

- Uspešno izvršavanje

```
#OK del <item-path>
>
$
```

Primer:

```
[\folder1\folder2]: del folder4
#OK del \folder1\folder2\folder4
>
$
```

4.3.6 CRC

CRC komandom izračunavamo CRC16 checksum fajl segmenta i opcionalno ga upoređujemo sa prosleđenim odgovarajućim argumentom. Vraćamo dužinu fajla i CRC16 vrednost fajl segmenta kao dva celobrojna broja razdvojena razmakom.
Upotreba

```
crc <file-name> [<start-length>] [crc-value]
```

$< file - name >$ - predstavlja ime fajla za čiji segment želimo da izračunamo crc $[< start - length >]$ - predstavlja opcionalni parametar u kojem navodimo dužinu i početak fajl segmenta za koji želimo da izračunamo CRC16 checksum. Ukoliko ne navedemo argument za ovaj parametar uzimamo za $start = 0$ i za $length$ dužinu celog $< file - name >$ fajla. $[< crc - value >]$ - predstavlja opcionalni parametar u kojem navodimo celobrojnu vrednost koju želimo da uporedimo sa CRC16 vrednošću od fajl segmenta. Ukoliko prosledimo $< crc - value >$ argument prilikom uspešnog izvršavanja funkcija pored ostalih povratnih vrednosti vraća 0 ili 1. U suprotnom na to mesto vraćamo izračunatu CRC16 vrednost. Povratne vrednosti

- Slučaj greške

```
#ERROR crc ERROR_CODE
>
$
```

54	SdFat exists: File doesn't exists
50	SdFat open: Can't access to the file
56	SdFat rmRfStar: Can't delete the folder
57	SdFat remove: Can't delete the file

Table 6: Značenje kodova grešaka

- Uspešno izvršavanje

```
#OK crc <file-path>
><start> <length>
><file-length> <crc-value || ( 0 || 1) >
$
```

Argument $< start >$ predstavlja poziciju od koje smo počeli računanje CRC16 vrednosti fajla. Argument $< length >$ predstavlja broj bajtova za koje smo računai CRC16 vrednosti.

Primer 1:

```
[\folder1\folder2]: crc fajl1
#OK crc \folder1\folder2\fajl1
>687 19238
$
```

Primer 2:

```
[\folder1\folder2]: crc fajl1 19238
#OK crc \folder1\folder2\fajl1
>687 1
$
```

4.3.7 PUT

Snimamo n bajtova u zadati fajl. Bajtovi su zadati kao celobrojne vrednosti u ASCII kodu. Primer PUT "fajl.txt" 65 66 67 69 –> u fajlu će se naći slova "ABCE". Ako fajl ne postoji, kreiramo ga. Ako postoji bajtove dodajemo na njega.
Upotreba

```
put <file-name> <array-of-dec-values>
```

Povratne vrednosti

- Slučaj greške

```
#ERROR put ERROR_CODE
>
$
```

50	SdFat open: File exists but we couldn't open the file (check if the folder with same name exists.)
51	SdFat open: Can't create file.
61	Parsing error: Not valid argument. (Unsuccessful decimal value parsing)

Table 7: Značenje kodova grešaka

- Uspešno izvršavanje

```
#OK put <file-path>
><written-data-length>
$
```

Primer:

```
[\folder1\folder2]: put new_file 65 66
#OK put \folder1\folder2\new_file
>2
$
```

4.3.8 GET

Vraćamo sadržaj fajl segmenta u vidu niza celobrojnih brojeva. Počinjemo čitanje od pozicije $start$ i čitamo $length$ bajtova. Na kraju niza bajtova opcionalno dodajemo CRC16 vrednost za pročitani blok.
Upotreba

```
get <file-name> [<start> <length>] ["crc"]
```

$< file - name >$ - predstavlja ime fajla $[< start >< length >]$ - opcionalni parametar kojem prosleđujemo poziciju od koje želimo da čitamo fajl i koliko bajtova želimo da pročitamo.
$[$"crc"$]$ - opcionalnom parametaru prosleđujemo doslovno *"crc"* ukoliko želimo da na kraju bloka bajtova koji smo pročitali da nalepimo i CRC16 vrednost tog bloka.
Čitanje počinjemo od pozicije $< start >$ koji je opcionalni argument. Ukoliko ne prosledimo argument $< start >$ podrazumeva se da počinjemo čitanje od početka fajla. Čitamo $< length >$ karaktera. Ukoliko ne prosledimo argument za parametar $< start >$ podrazumevano uzimamo za parametar $< length >$ vrednost ukupne veličine fajla sa nazivom argumenta koji smo prosledili parametru $< file - name >$. $< start >$ i $< length >$ parametri su opcionalni. Takođe možemo opcionalno i doslovno da prosledimo kao zadnji argument, argument na parametar $[$"crc"$]$. Ukoliko prosledimo ovaj argument želimo da izračunamo CRC16 vrednost fajl segmenta koji čitamo.
Povratne vrednosti

- Slučaj greške

```
#ERROR get ERROR_CODE
>
$
```

66	Not valid argument. (Unsuccessful decimal value parsing for the <start>argument)
67	Not valid argument. (Unsuccessful decimal value parsing for the <length>argument)
68	Not valid arguments. (<start>+ <length>is greater then the size of the file)
69	Not valid arguments. (The <start>argument is not null but the <length>argument is null)
52	SdFat open: File exists but we couldn't open the file (check if the folder with same name exists.)
60	Fgets: Can't read from a file
61	Fgets: Can't read from a file [line too long]
54	SdFat exists: File doesn't exist

Table 8: Značenje kodova grešaka

- Uspešno izvršavanje

```
#OK get <file-path>
><start> <length>
><file-sector-data>
[><crc-value>]
$
```

Argument $< start >$ predstavlja poziciju od koje smo počeli čitanje fajla $< file-path >$. Argument $< length >$ predstavlja broj bajtova koji smo pročitali i $< file - sector - data >$ predstavlja podatke koje smo pročitali.

Primer:

```
[\folder1\folder2]: get new_file 0 2 crc
#OK get \folder1\folder2\new_file
>65 66
>13123
$
```

4.4 Kodovi ostalih grešaka

COMMAND_ERR_GENERALERROR	10
COMMAND_ERR_MISSINGARGUMENT	11
COMMAND_ERR_EXTRAARGUMENT	12
COMMAND_ERR_INVALIDVALUE	13
COMMAND_ERR_VALUEOUTOFRANGE	14
COMMAND_ERR_UNKNOWNCOMMAND	15

Table 9: Ostale greške

5 Opis serverskog Arduino koda

5.1 Opis glavnog Arduino fajla: SerijskiKomandniProcesorSDCARD.ino

Potrebno je da uvezeno instaliranu biblioteku *SdFat* (pogledati sekciju sa nazivom *Potrebna programska okruženja* za upuststvo za instalaciju.) Takođe uvozimo i sledeće fajlove:

- cmdProc.h
- buf.h
- sdCardManipulator.h

Pogledati odgovarajuće podsekcije za opis i način upotrebe ovih fajlova.

Koristimo tri interne promenljive:

- Objekat klase $CmdProc$. Klasu deklarišemo u fajlu sa nazivom "cmdProc.h". U objektu klase $CmdProc$ čuvamo podržane komande.
- Objekat klase Buf koristimo za čuvanje pročitanih podataka sa serijskog porta.
- $chipSelect$ u kojoj čuvamo celobrojnu vrednost pina preko kojeg vršimo vremensku sinhronizaciju sa modulom za čitanje SD kartica.

Takođe koristimo i globalne promenljive koje smo uvezli kroz uvezene fajlove:

- iz fajla "sdCardManipulator.h" koristimo sledeće promenljive:
  - Objekat sa nazivom sd klase $SdFat32$. Objekat klase $SdFat32$ predstavlja "kernel" za upravljanje modulom čitača SD kartica.
  - Objekat sa nazivom $cwdFile$ klase $File32$. Objekat klase $File32$ predstavlja fajl koji je trenutno otvoren odnosno sa kojim trenutno radimo i koji se nalazi na SD kartici.

Dopunjujemo metode:

1. $setup()$ i postavljamo brzinu protoka serijskog čitanja i pisanja na 9600 baud. Nakon toga inicijalizujemo tunel za manipulisanje podacima između Arduino mikrokontrolera koji izvršava Arduino serverski kod i modula čitač SD kartica. U slučaju neuspešne inicijalizacije prijavljujemo grešku. Takođe ukoliko nismo uspeli da otvorimo root korišćenjem metode nad $File32$ objektom: $cwdFile.open("/")$ prijavljujemo grešku. Nakon završetka rada sa fajlom zatvaramo ga korišćenjem metode $cwdFile.close()$.

   Ostalo je još da inicijalizujemo, obezbedimo i ponudimo programeru podržane komande. Koristimo objekat klase $CmdProc$ u kome čuvamo podržane komande. Na početku izvršavamo metodu $Init(int)$ kojoj prosleđujemo za argument broj podržanih komandi. Nakon toga dodajemo komande korišćenjem metode $Add(string, (*cmdCallback), int, int)$ gde

   - za prvi argument navodimo ime komande
   - za drugi argument pokazivač na funkciju koja treba da se izvrši
   - broj obaveznih argumenta
   - broj ukupnih argumenata

2. $loop()$ u kojoj navodimo logiku za čitanje i interpretiranje pročitanog sa serijskog porta. Kada imamo dostupne podatke na serijskom portu čitamo ih i smeštamo u prethodno definisanu promenljivu $sbuf$. Ukoliko smo naišli na podatak koji ima celobrojnu vrednost 10 (novi red) označavamo da je naš bafer (promenljiva $sbuf$) spreman za obrađivanje i prosledimo ga na obrađivanje. Obrađivanje vršimo korišćenjem statične metode $Parse(Buf)$ klase $Commands$. U slučaju

neuspešnog obrađivanja bafera vraćamo celobrojnu vrednost različitu od 0. U tom slučanju vraćamo grešku pošiljaocu preko serijskog tunela kao rezultat izvršavanja primljene komande. Sadržaj greške modeliramo korišćenjem statične metode $printErrorV2(char*, int)$ klase $ReturnValues$.

U oba slučaja bilo da se javila greška prilokom obrađivanja ili ne, čistimo bafer korišćenjem metode $Clear()$ nad promenljivom $sbuf$.

Ukoliko se nije javio podatak koji ima celobrojnu vrednost 10 ali je različit od 13 (povratak na početak reda) pokušavamo da dodamo taj podatak u bafer. U slučaju neuspešnog dodavanja podatka u bafer javlja se prekoračenje bafera i vraćamo vrednost $false$.

5.2 buf.h

U ovom fajlu navodimo strukturu sa nazivom Buf u kojoj definišemo metode i promenljive za čuvanje pročitanih podataka sa tunela.

5.3 cmdProc.h

U ovom fajlu deklarišemo klase koje koristimo za pravljenje objekata u kojima čuvamo i koristimo podržane *komande*.

U klasi Cmd definišemo okruženje za izvršavanje komandi. U objektu klase Cmd čuvamo naziv komande, memorijsku lokaciju funkcije koja se izvršava, broj obaveznih parametara i ukupan broj parametara.

U objektu klase $CmdProc$ čuvamo sve podržane komande. Komande dodajemo korišćenjem metode $Add(char, (*cmdCallback), int, int)$. Objekat klase $CmdProc$ sadrži metode $GetNextToken()$, $Parse(char*)$ i $Exec()$ koje koristimo da odredimo kontekst u kojem izvršavamo funkciju komande odnosno određujemo argumente koje je potrebno da prosledimo funkciji komande. Ukoliko se broj argumenta poklapa sa traženim brojem argumenata izvršavamo funkciju komande.

Takođe deklarišemo pomoćne funkcije za parsiranje i ispitivanje raznih podataka koje definišemo u fajlu $cmdProc.cpp$:

1. $tryParseInt(char*, int\&)$ koju koristimo za pokušaj konvertovanja prosleđenog prvog argumenta koji predstavlja reč u celobrojnu ili heksadecimalnu vrednost. Rezultat smeštamo na memorijsku lokaciju na kojoj se nalazi drugi argument.

2. $tryParseDec(char*, int\&, bool)$ koju koristimo za pokušaj konvertovanja prosleđenog prvog argumenta koji predstavlja reč u decimalnu vrednost. Rezultat smeštamo na memorijsku lokaciju na kojoj se nalazi drugi argument. Kao treći opcionalni argument imamo indikator da li prvi argument na prvom mestu sadrži karakter $'-'$.

3. $tryParseHex(char*, int\&, bool)$ koju koristimo za pokušaj konvertovanja prosleđenog prvog argumenta koji predstavlja reč u heksadecimalnu vrednost. Rezultat smeštamo na memorijsku lokaciju na kojoj se nalazi drugi argument.

4. $isHexDigit(char)$ koristimo za proveru da li prosleđeni karakter predstavlja heksadecimalni simbol.

5. $tryParseTime(char*, int\&, int\&, int\&)$ koristimo za pokušaj konvertovanja prvog prosleđenog argumenta koji predstavlja reč u dve vrednosti. Prva vrednost

predstavlja broj sati i druga vrednost predstavlja broj minuta. Vrednosti smeštamo redom na memorijske lokacije prosleđene u drugom i trećem argumentu.

6. $tryParseDate(char*, int\&, int\&, int\&)$ koristimo za pokušaj konvertovanja prvog prosleđenog argumenta koji predstavlja reč u tri vrednosti. Prva vrednost predstavlja broj dana, druga vrednost predstavlja broj meseca i treća vrednost predstavlja broj godine. Vrednosti smeštamo redom na memorijske lokacije prosleđene u drugom, trećem i četvrtom argumentu.

7. $isValidDate(int, int, int)$ koristimo za proveru da li prvi argument predstavlja dan, da li drugi predstavlja mesec i da li treći predstavlja godinu.

5.4 cwdPath.h

U objektu klase $SdFat32$ biblioteke $SdFat$ ne postoji praćenje apsolutne putanje trenutnog radnog direktorijuma. Definisali smo novu klasu $CwdPath$. Objekat klase $CwdPath$ koristimo za praćenje apsolutne putanje trenutnog radnog direktorujima.

5.5 returnValues.h

U ovom fajlu definišemo metode u kojima modeliramo povratne vrednosti. Model rezultata izvršavanja primljene komande prati šablone povratnih vrednosti u opštem slučaju koje smo definisali u podsekciji 4.2 - *Povratne vrednosti u opštem slučaju*.

5.6 sdCardManipulator.h

U ovom fajlu definišemo funkcije komandi. Najpre uvozimo potrebne fajlove:

1. $< arduino.h >$ i $< avr/pgmspace.h >$ uvozimo za čitanje podataka iz programske memorije
2. <string.h> za manipulaciju stringovima
3. <SdFat.h> za manipulaciju podataka preko modul čitača za SD kartice
4. <util/crc16.h> za računanje CRC16 checksum-e
5. "cwdPath.h" za praćenje apsolutne putanje trenutno radnog direktorijuma
6. "returnValues.h" u ovom fajlu se nalaze modeli povratnih vrednosti

U programsku memoriju smeštamo

- sadržaj koji koristi $HELP$ komanda
- nazive podržanih komandi

U nastavku su nabrojane funkcije za podržane komande. Sve funkcije za parametar imaju pokazivač na objekat klase $CmdProc$. U ovom objektu nalazi se kontekst u kojem se funkcija izvršava.

Na početku svake funkcije kopiramo naziv komande iz programske memorije u izvršni deo koda korišćenjem metode $strcpy\_P$. Nakon iteraranja zatvaramo i $cwdFile$ fajl i štampamo zaglavlje i simbol $ korišćenjem statične metode $printEndAndReady$ klase $ReturnValues$.

5.7 cmdDir

Definisali smo dve promenljive sa nazivima i_file i $cwdFile$ klase $File32$. Korišćenjem metode $open(char*)$ objekta $cwdFile$ otvaramo trenutni radni direktorijum. U slučaju greške prijavljujemo i vraćamo grešku.

Metodom $rewind()$ premotovamo $cwdFile$ na početak i $while$ petljom iteriramo kroz trenutni radni direktorijum. Iterativno postavljamo i_file objekat da predstavlja fajl koji sledeći čitamo. Postavljamo ga korišćenjem metode $openNext$ kojoj prosleđujemo direktorijum koji čitamo i tip privilegije čitanja.

U svakoj iteraciji:

1. ispisujemo simbol $>$ korišćenjem statične metode $printResult$ klase ReturnValues
2. ispisujemo veličinu fajla
3. ispisujemo ime fajla i u slučaju direktorijuma dodajemo na kraju simbol "/".
4. na kraju svake iteracije zatvaramo i_file.

5.8 cmdHelp

Koristimo klasu $\_\_FlashStringHelper$ pomoću koje čitamo reč iz programske memorije. Ovo je alternativni način i ovo možemo da uradimo i korišćenjem metode $strcpy\_P$.

Iteriramo kroz niz sa nazivom $helpTable$ koji je smešten u programsku memoriju i u svakoj iteraciji čitamo element niza iz programske memorije i štampamo ga.

5.9 cmdPut

Obrađujemo prosleđene argumente. Proveravmo da li fajl sa nazivom prvog prosleđenog argumenta postoji. Ako postoji pokušavamo da ga otvorimo i u suprotnom pokušavamo da ga napravimo. Prilikom neuspešnog izvršavanja vraćamo odgovarajuću grešku.

Drugi i ostali argumenti predstavljaju podatke koje želimo da upišemo u fajl. Svaki argument obrađujemo i proveravamo da li predstavlja celobrojnu vrednost. U slučaju celobrojne vrednosti upisujemo u fajl.

5.10 cmdDel

Ispitujemo da li fajl postoji i ukoliko postoji pokušavamo da ga obrišemo. Ukoliko nam je prosleđen naziv foldera koji trebamo da obrišemo koristimo metodu $rmRfStar$ objekta klase $File32$ za rekurzivno brisanje ne nužno praznog foldera.

5.11 cmdMd

Ispitujemo da li postoji folder sa istim nazivom korišćenjem metode $exists$ objekta klase $SdFat32$ i ukoliko postoji vraćamo odogovarajuću grešku. U suprotnom pokušavamo da napravimo folder korišćenjem metode $mkdir$ nad objektom klase $SdFat32$.

5.12 cmdCrc

Obrađujemo argumente. Prvi argument predstavlja naziv fajla i ostali argumenti su opcionalni. Prilikom uspešnog izvršavanja računamo CRC16 vrednost za segment fajla i vraćamo odgovarajuću poruku.

5.13 cmdGet

Obrađujemo argumente i prilikom uspešnog validiranja argumenata čitamo iz fajla odgovarajući broj karaktera. Za pročitani karakter uzimamo njegovu celobrojnu vrednost i šaljemo je preko serijskog porta. Na kraju ukoliko je prosleđen i opcionalni argument ["crc"] računamo i CRC16 vrednost za pročitani blok podataka i njega takođe šaljemo.

5.14 cmdCd

Obrađujemo argumente. Ukoliko prvi argument predstavlja:

1. "/" ; vraćamo se na root korišćenjem metode $chdir$ objekta klase $SdFat32$. Takođe i objekat wp klase $CwdPath$ resetujemo.

2. ".." ; *izbacimo iz lanca* ("pop"-ujemo) zadnji fajl koji se nalazi u apsolutnoj putanji trenutnog radnog direktorijuma. Apsolutnu putanju trenutnog radnog direktorijuma čuvamo u objektu sa nazivom wp klase $CwdPath$. Nakon toga korišćenjem metode $chdir$ nad objektom klase $SdFat32$ sa argumentom $wp.get()$ menjamo trenutni radni direktorijum.

3. naziv direktorijuma ; pokušavamo da pređemo na zadati folder. U slučaju greške vraćamo odgovarajuću poruku.

5.15 errorCodes.h

U ovom fajlu navodimo kodove grešaka.