Komunikacijski sklad in nadzorna aplikacija za mikrokrmilnike

Mitja Nemec

Univerza v Ljubljani, Fakulteta za elektrotehniko, Tržaška 25, 1000 Ljubljana, Slovenija
E-pošta: mitja.nemec@fe.uni-lj.si

Communication stack and control application for microcontrollers

Abstract. *The article presents how an application for communication and control of an application running on microcontroller was developed. The main task of application running on micocontroller is control of power converter. This results in some specific requirements, the main ones being low software complexity and modularity which should enable reuse. Besides main application architecture, the article focuses on the description of communication stack.*

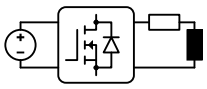
1 Uvod

Na Fakulteti za elektrotehniko v Laboratoriju za regulacijsko tehniko in močnostno elektroniko (LRTME) se primarno ukvarjamo z razvojem močnostne elektronike. Z napredkom na področju polprevodnikov, pa v vedno večji meri analogne in namenske (ASIC) rešitve, za krmiljenje in regulacijo močnostne elektronike, zamenjujejo digitalni pristopi [1]. Zaradi kadrovskih omejitev smo se med različnimi možnimi pristopi digitalizacije omejili na uporabo namenskih (power conversion) mikrokrmilnikov [2]. V primerjavi s programirljivimi logičnimi vezji (FPGA) so mikrokrmilniki v praksi bolj razširjeni in s stališča razvoja tudi lažje obvladljivi.

Pri razvoju močnostne elektronike, ki temeljil na mikrokrmilniku smo na začetku regulacijo nadzorovali s stikali in/ali potenciometri, interne veličine (e.g. veličine transformirane v *dq* prostor, …) pa smo opazovali preko DA pretvornika. S pojavom mikrokrmilnikov, ki so omogočili opazovanje in spreminjanje spremenljivk v realnem času brez zaustavitve mikrokrmilnika lahko opazovanje in nadzor izvajamo znotraj razvojnega okolja [3], [4]. Vendar pa je ta rešitev uporabna samo v fazi razvoja in ni primerna za uporabo v končnem izdelku. Tako se je vedno bolj izkazovala potreba po izdelavi aplikacije s katero bi preko osebnega računalnika upravljali mikrokrmilnik, ki izvaja regulacijo v močnostni elektroniki.

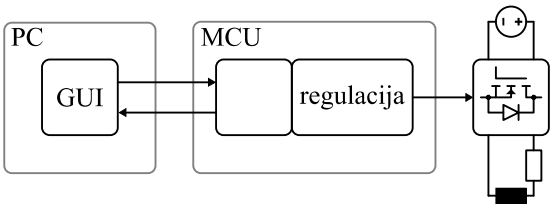
2 Opis tipičnega sistema

Tipična aplikacija s katero se srečujemo v LRTME obsega močnostni pretvornik s katerim reguliramo tok in preko njega posredno druge veličine (e.g. navor, hitrost, napetost, moč …). Pretvornik je lahko samo eden, lahko pa jih je več vezanih bodisi serijsko ali paralelno.



Slika 1: Tipičen močnostni pretvornik

Razvoj programske opreme za tak pretvornik tipično poteka v fazah. S prvimi verzijami programske opreme se preizkusi vse osnovne funkcionalnosti strojne opreme (e.g. ali zajem merjenih veličin dela, ali proženje močnostnih stikal dela, …). Nato se doda potrebne transformacije merjenih veličin (3-2, *αβ-dq*, …), čemur sledi implementacija notranje regulacijske zanke. Tipično je to regulacija toka. V tej fazi želimo nastavljati obliko želene vrednosti (pulz, sinus, …) in njene parametre (amplituda, frekvenca, …). Na podlagi opazovanja odziva v časovnem prostoru pa nastavimo parametre regulatorja. Nato tipično dodamo možnost normalnega vklopa in izklopa močnostne stopnje kot tudi možnost izrednega izklopa ob zaznanem napačnem delovanju. Sledi dodajanje morebitnih nadrejenih regulacijskih zank (regulacija napetosti, hitrosti, položaja, …). Pri razvoju programske opreme v vsakem od teh korakov zelo prav pride možnost nastavljanja in spremljanja veličin znotraj mikrokrmilnika kar preko osebnega računalnika.



Slika 2: Predlagana izvedba rešitve

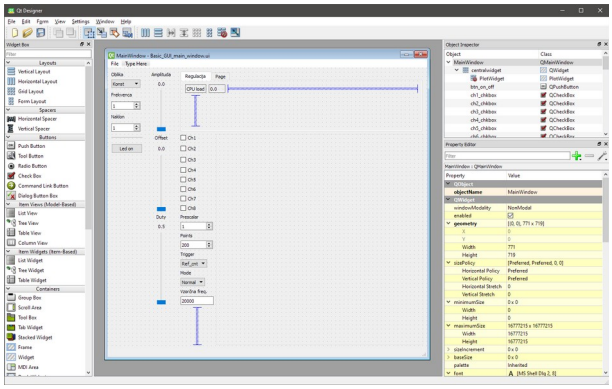
V preteklosti smo podobne aplikacije že razvijali na osnovi programskega okolja LabView, vendar pa se je pri delu z njim izkazalo nekaj pomanjkljivosti. Labview je okolje, kjer se program piše v grafični obliki. Tako zahteva risanje kode precej discipline, da je koda razumljiva in primerna za vzdrževanje. Ker pa imamo v LRTME veliko fluktuacijo kadra (študenti, raziskovalci) se je vzdrževanje discipline izkazalo za nemogoče. Prav tako je vnašanje popravkov v več aplikacij, ki temeljijo na isti osnovi, v grafičnem načinu duhomorno. Predvsem pa je bilo problematično nenehno nadgrajevanje razvojnega okolja in vzdrževanje različnih verzij za potrebe razvoja posameznih aplikacij, ki so temeljile na različnih verzijah.

Na osnovi vseh teh izkušenj, smo se odločili za izdelavo nove aplikacije, ki bi izpolnjevala večino naših potreb.

2.1 Zahteve

Na osnove preteklih izkušenj smo postavili nekaj osnovnih zahtev:
- aplikacija naj temelji na enem izmed splošno razširjenih programskih jezikov (C#, C++, java, python …)
- zaželeno je, da se uporabniški vmesnik načrtuje v grafičnem okolju
- arhitektura aplikacije mora omogočati enostavno prilagajanje specifičnim projektom in tako služi kot skupna osnova za posamezne aplikacije
- arhitektura aplikacije mora biti modularna, tako da se popravki v enem modulu, ki je skupen vsem aplikacijam za specifične projekte brez težav vstavijo/prenesejo v vse aplikacije
- uporabljene rešitve naj bodo čim bolj enostavne saj v LRTME nimamo dovolj virov za razvoj in vzdrževanje kompleksne programske opreme.

Na podlagi naštetih omejitev je bila sprejeta odločitev, da za razvoj aplikacije uporabim programski jezik Python. Pri izbiri programskega jezika smo upoštevali tudi dejstvo, da Python uporabljamo tudi pri reševanju občasnih numeričnih problemov oz. analizi podatkov. Za razvoj uporabniškega vmesnika smo uporabili knjižnico PyQt, ki omogoča grafično načrtovanje uporabniških vmesnikov (slika 3).



Slika 3: QtDesigner - orodje za načrtovanje grafičnega vmesnika

Po kratkotrajnem testiranju knjižnic za izris grafov smo namesto knjižnice sicer nekoliko bolj razširjene knjižnice Mathplotlib raje izbrali PyQtgraph, saj je slednja bistveno hitrejša, kar pri naših aplikacijah, ko želimo graf osvežiti večkrat na sekundo igra pomembno vlogo. Razlika v hitrosti je bila očitna že pri preprostih aplikacijah.

2.2 Komunikacija

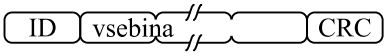
Velik del aplikacije predstavlja komunikacijski sklad, tako na strani osebnega računalnika, kot tudi na strani mikrokrmilnika. Na podlagi osnovni zahtev, da naj bodo uporabljene čimbolj preproste rešitve, smo pri implementaciji komunikacije izločili vse kompleksne komunikacijske protokole, kot so na primer USB, LAN …). Tako komunikacija temelji na dobro poznanem UART protokolu [5]. Dodaten razlog za to izbiro je tudi bistveno manjša poraba spomina in računske moči na strani mikrokrmilnika.

Ker UART protokol določa samo del povezovalne plasti po ISO/OSI referenčnem modelu [6], smo morali celoten protokol definirati. Načeloma bi se lahko pri dokončni definicij protokola naslonili na že razvite in poznane protokole (MODBUS, PPP) vendar smo zaradi nekaterih specifičnih zahtev razvili lasten protokol.

Prava od specifičnih zahtev je bila želja, da protokol podpira prenos podatkov, ki so daljši kot 255 bajtov, pri čemer je bila pričakovana najdaljša dolžina okvirno nekaj tisoč bajtov. Načeloma bi lahko pošiljanje daljših podatkov rešili na transportni plasti, vendar pa bi to dodatno zakompliciralo tako definicijo samega protokola kot tudi programske kode, ki bi ga implementirala.

Druga zahteva pa je bila čimvišja hitrost prenosa podatkov. Delno se ta zahteva lahko rešuje s čimvišjo bitno hitrostjo, vendar na to velik vpliv tudi način pošiljanja podatkov. Tako smo se odločili, da se podatki prenašajo v binarni obliki, saj je gostota tako prenesenih podatkov bistveno višja kot če bi podatke pošiljali v ASCII obliki.

Tako definiranemu protokolu smo za zagotavljanje pravilnosti prejetih podatkov dodali še 16 bitno CRC kodo. Končen podatkovni okvir je prikazan na sliki 4. Okvir temelji na 16 bitnih besedah. V prvi besedi je shranjen identifikator paketa, s čimer prejemniku sporočim katera informacija se nahaja v paketu. Nato sledijo podatki (do 65532 16 bitnih besed), na koncu pa je priključena še 16 bitna CRC koda.



Slika 4: Predlagan podatkovni okvir

Pri binarnem kodiranju se pojavi problem razmejitve podatkovnih okvirjev, ki smo ga pri predlaganem protokolu rešili tako da se podatki pred samim pošiljanjem kodirajo po COBS principu [7]. Le ta zagotavlja, da se v poslanih podatkih nikoli ne pojavi bajt z vsebino »0x00«. Tako smo lahko bajt z vsebino »0x00« uporabili za označevanje konca poslanega podatkovnega okvirja in s tem zagotovili uspešno sinhronizacijo podatkovnih poslanih in prejetih okvirjev.

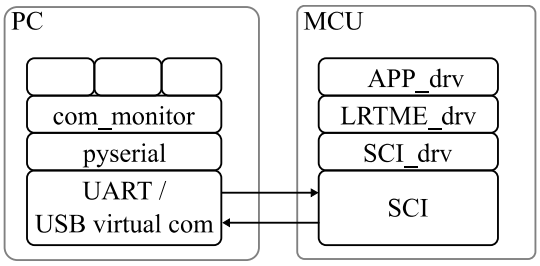
3 Razvoj aplikacije

Razvoj aplikacije je potekal v fazah. Velik del njih je bil načrtovan že od začetka, nekatere pa so so dodale kasneje, ko se je tekom uporabe izkazala potreba. V grobem se je najprej na strani osebnega računalnika implementiral osnovni okvir grafičnega vmesnika, nato pa se je pričel razvoj komunikacijskega vmesnika. Ko je razvoj le-tega bil zaključen pa so se postopoma dodajali gradniki končnega grafičnega vmesnika s pripadajočo kodo tako v programu, ki teče na osebnem računalniku, kot tudi v programu, ki teče na mikrokrmilniku

3.1 Razvoj in testiranje komunikacijskega sklada

Razvoj komunikacijskega sklada je v okviru razvoja celotne aplikacije zahteval veliko časa in je potekal postopno. V prvi fazi smo pripravili načrt okvirne arhitekture pri katerem smo predvideli tri med seboj bolj

ali manj ločene sloje. Najnižjo sloj protokola upravlja z UART/SCI periferno napravo ter skrbi za pošiljanje in prejemanje paketov. Za sestavljanje oz. tolmačenje paketov skrbi vmesni sloj, najvišji sloj pa nudi smiseln programski vmesnik (API) za končno aplikacijo (slika 5).



Slika 5: Arhitektura komunikacijskega sklada

Glavni namen predstavljene arhitekture je v modularnosti, saj je zamišljeno, da se bo za posamezen projekt spremenilo oz. prilagodilo samo zgornji sloj spodnji sloji pa bodo ostali nespremenjeni.

Ko je bila arhitektura zasnovana, smo se najprej lotili implementacije in testiranja najnižjega nivoja protokola. Nato smo implementirali in preizkusili delovanje srednji nivo. Na koncu pa je sledil še razvoj same aplikacije

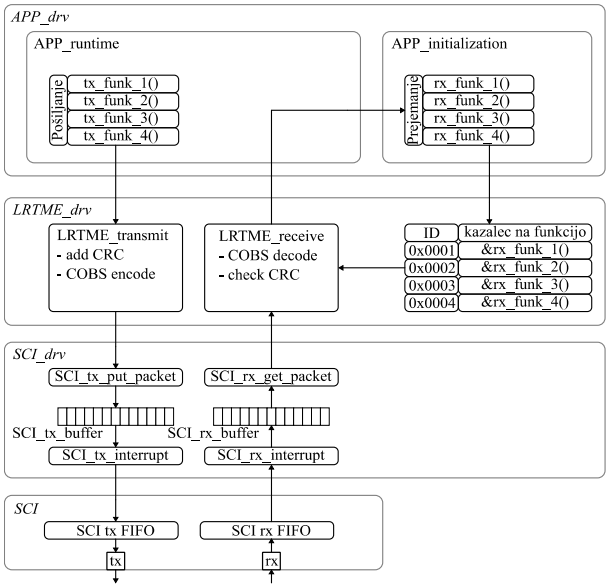
3.1.1 Komunikacijski sklad na strani mikrokrmilnika

Nekoliko bolj detajlna slika arhitekture komunikacijskega sklada na strani mikrokrmilnika je prikazana na sliki 6. Vsa koda, ki s strani aplikacije skrbi za pošiljanje in prejemanje podatkov je zbrana v modulu *APP_drv*.

Pošiljanje podatkov je relativno enostavno. Ko je klicana ustrezna funkcija, le ta pripravi podatek za pošiljanje in ga posreduje nižjemu sloju, ki se nahaja v modulu *LRTME_drv*. Le-ta podatkom doda CRC in zakodira podatke po COBS principu. Tako zakodirane podatke pa posreduje gonilniku za serijsko komunikacijo, ki po komunikacijskem vodilu pošilja posamezne bajte. Gonilnik podatke spravi v vrsto *SCI_tx_buffer*, ki primarno služi za ločitev med izvajanjem kode v glavni programski zanki in prekinitvijo, ki je prožena, ko je SCI enota pripravljena na pošiljanje novih bajtov.

Prejeti bajti se prav tako shranijo v vrsto *SCI_rx_buffer*, ki prav tako ločuje prekinitev od glavne programske zanke. V kolikor je prejeti bajt enak 0x00 to označuje konec prejetega paketa. V glavni programski zanki je periodično klicana funkcija *LRTME_receive*, ki preveri ali je kak paket na voljo, in če je, ga prevzame iz vrste *SCI_rx_buffer*, odkodira po COBS postopku, in v kolikor so podatki pravilni pokliče ustrezno funkcijo ter ji posreduje podatke. Katero funkcijo pa naj pokliče pa se nastavi inicializaciji kjer shranijo naslove funkcij, ki naj bodo klicani, ob sprejemu specifičnih podatkov.

Na koncu velja še omeniti nekaj tehničnih detajlov. Koda je bila razvita za mikrokrmilnike družine C2000 proizvajalca Texas Instruments. Omenjena družina omogoča naslavljanje najmanj 16 bitnih podatkov, izračun CRC-ja, COBS kodiranje in dekodiranje ter pošiljanje po UART protokolu pa temeljijo na 8 bitnih podatkih. Tako smo pri implementaciji bili primorani uporabiti s strani prevajalnika podprte posebne ukaze (compiler intrinsics) [8].



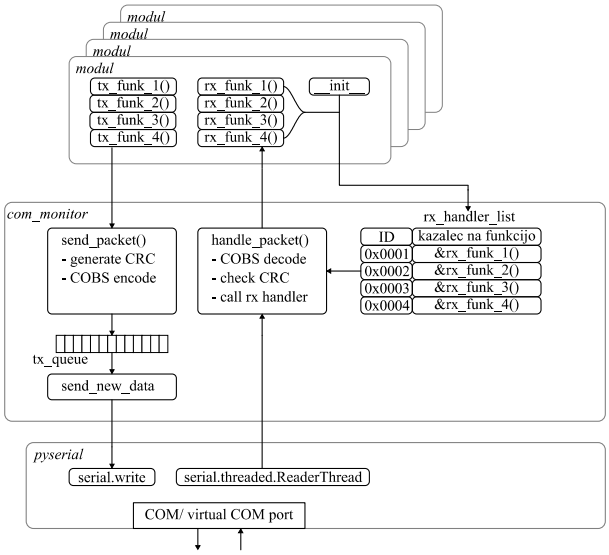
Slika 6: Detajlna arhitektura komunikacijskega sklada na strani mikrokrmilnika

3.1.2 Komunikacijski sklad na strani osebnega računalnika

Arhitektura komunikacijskega sklada na strani osebnega računalnika (slika 7) je precej podobna predhodno predstavljeni arhitekturi. Gornji sloj se razlikuje v toliko, da omogoča izvajanje komunikacije iz različnih modulov. Vmesni sloj skrbi za oblikovanje podatkovnih paketov za pošiljanje, kot tudi za dekodiranje in preverjanje prejetih paketov ter posredovanje in klicanje ustreznih funkcij. Spodnji sloj pa je v celoti realiziran v knjižnici *pyserial* ki pa jo je treba primerno nastaviti.

S *pyserial.threaded.ReaderThread* lahko nastavimo, da se ob prejemu paketa, ki je zaključen z bajtom 0x00, avtomatsko kliče *handle_packet* funkcija. Tako se večina kode, ki skrbi za prejemanje podatkov izvaja v sistemskih knjižnicah, kar omogoča da smo brez večjih težav dosegli bitne hitrosti 1 Mbps in več. Tako hitra komunikacija seveda zahteva tudi hiter odziv na prejete podatke. Da pa hitrost komunikacije ni vplivala na odzivnost grafičnega vmesnika se komunikacijski del kode izvaja v ločeni programski niti kot se izvaja uporabniški vmesnik. Tako se v *handle_packet* prejeti podatki dekodirajo ter se preveri CRC. Nato se podatki shranijo v vrsto, ter se generira signal s katerim se v niti, v kateri teče uporabniški vmesnik izvede ustrezna funkcija, ki obdela prejete podatke in osveži izbrane elemente v uporabniškem vmesniku. Seznam, katera funkcija naj se izvede za specifičen ID prejetih podatkov, pa se napolni ob inicializaciji, kjer vsak modul posreduje svoj seznam funkcij.

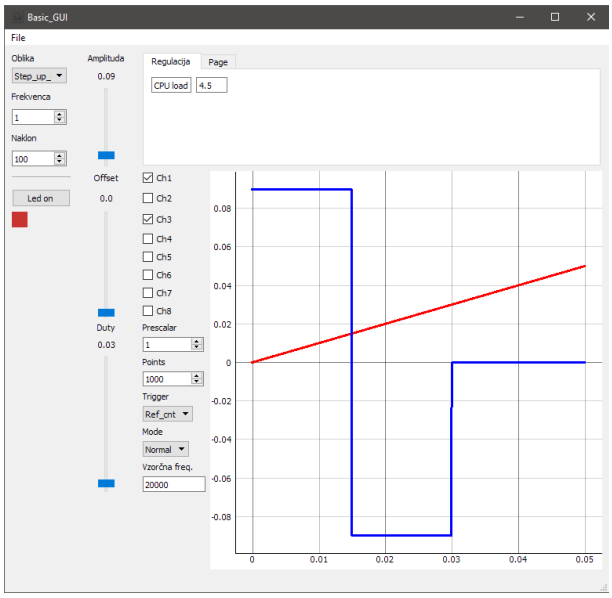
Pošiljanje je precej podobno pošiljanju na strani mikrokrmilnika. Aplikacija posreduje podatke vmesnemu sloju preko funkcije *send_packet*. Le ta podatkom doda CRC in jih zakodira po COBS principu. Pošiljanje se prav tako izvaja v ločeni niti, kjer je vrsta *tx_queue* uporabljena za posredovanje podatkov med nitjo v kateri teče uporabniški vmesnik in nitjo, ki skrbi za pošiljanje v kateri s klicem funckije *serial.write* podatke posredujemo knjižnici *pyserial*.



Slika 7: Detajlna arhitektura komunikacijskega sklada na strani osebnega računalnika

4 Zaključek

Primer grafičnega vmesnika končne aplikacije je prikazan na sliki 8. Aplikacija omogoča spremljanje signalov v mikrokrmilniku, kot tudi nastavljanje želenih vrednosti. Prve verzije aplikacije so bile razvite že leta 2015. Od takrat pa je arhitektura doživela nekaj manjših popravkov. V vsem tem času smo v LRTME zadovoljni z njenim delovanjem.



Slika 8: Primer aplikacije

Ker se v LRTME se redkokdaj srečamo z razvojem tako kompleksne programske opreme smo tekom razvoja smo prišli do ugotovitev, do katerih so prišli že mnogi pred nami [9]:

- temeljito načrtovanje arhitekture se obrestuje, saj je kasneje med samim pisanjem programa potrebnih manj prilagoditev
- modularnost kode je ključnega pomena. Le ta omogoča:
  - enostavno vnašanje popravkov v posamezne module, ki so v uporabi pri različnih aplikacijah,
  - enostavno prilagoditev tipske aplikacije specifičnim zahtevam.
  - lažji pregled nad kodo projekta tudi za neizkušene programerje, ki se lahko ukvarjajo samo z posameznim modulom in lahko ostali del odmislijo

Zahvala

Delo je bilo sofinancirano iz programa ARRS »Pretvorniki električne energije in regulirani pogoni« P2-0258 (B).

Literatura

[1] S. Buso and P. Mattavelli, “Digital Control in Power Electronics,” *Synth. Lect. Power Electron.*, vol. 1, no. 1, pp. 1–158, Jan. 2006, doi: 10.2200/S00047ED1V01Y200609PEL002.

[2] M. Nemec, “Tackling with problems of programming in LRTME,” in *2012 15th International Power Electronics and Motion Control Conference (EPE/PEMC)*, Sep. 2012, p. DS3e.2-1-DS3e.2-4. doi: 10.1109/EPEPEMC.2012.6397356.

[3] D. Pahl, “Debugging Your C24x™ DSP Design Using Code Composer Studio™ RealComposer Studio™ Real--Time Monitor”Time Monitor,” 2001. http://cfile219.uf.daum.net/attach/207F6B3D50DB206C0B872E (accessed Jun. 17, 2022).

[4] “C2000 Real-Time Features,” *TI Training*, Mar. 13, 2015. https://training.ti.com/c2000-real-time-features (accessed Jun. 17, 2022).

[5] “Universal asynchronous receiver-transmitter,” *Wikipedia*. Jun. 15, 2022. Accessed: Jun. 21, 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Universal_asynchronous_receiver-transmitter&oldid=1093313393

[6] “ISO/OSI referenčni model,” *Wikipedija, prosta enciklopedija*. Apr. 13, 2022. Accessed: Jun. 21, 2022. [Online]. Available: https://sl.wikipedia.org/w/index.php?title=ISO/OSI_referen%C4%8Dni_model&oldid=5678148

[7] S. Cheshire and M. Baker, “Consistent overhead byte stuffing,” *IEEEACM Trans. Netw.*, vol. 7, no. 2, pp. 159–172, Apr. 1999, doi: 10.1109/90.769765.

[8] “TMS320C28x Optimizing C/C++ Compiler.” Texas Instruments Inc., Jun. 2022. Accessed: Jun. 29, 2022. [Online]. Available: https://www.ti.com/lit/ug/spru514

[9] S. MacConnell, *Code complete: a practical handbook of software construction*. Microsoft Press, 1993.