Modifikace řídicího softwaru pro pohon robotického ramene

Pracoviště: KEV/RICE

Číslo dokumentu: 22190-016-2018

Typ zprávy: Výzkumná zpráva

Řešitelé: Ing. Zdeněk Kehl,
Ing. Tomáš Glasberger, Ph.D.

Hlavní řešitel: Prof. Ing. Zdeněk Peroutka, Ph.D.

Počet stran: 33

Datum vydání: 24. 7. 2018

Oborové zařazení: JA – Elektronika a optoelektronika, elektrotechnika

Zadavatel / zákazník:

Zpracovatel / dodavatel:

Západočeská univerzita v Plzni
RICE
Univerzitní 8
306 14 Plzeň

Kontaktní osoba:

Ing. Zdeněk Kehl
tel. 377 634 140
kehlz@rice.zcu.cz

Zpráva vznikla s finanční podporou TAČR v rámci projektu TE02000103 (CIDAM) a s finanční podporou projektu SGS-2018-009.

Anotace

Tato výzkumná zpráva je zaměřena na úpravy řídicího softwaru pro pohon robotického ramene. Je zde podrobně rozepsáno hledání a řešení problémů, které se objevily při ladění řídicího softwaru. Zmiňované problémy narušovaly plynulý chod pohonu a nahodile vyvolávaly chybové stavy měniče. Dále jsou v této zprávě popsány úpravy řídicího softwaru, které si přál zákazník. Jednalo se především o úpravy stávajícího kódu, úpravy komunikace CAN a přidání několika dalších funkcí.

Klíčová slova

Pohon, software, CAN, Cycle counter, Process data watchdog

Název zprávy v anglickém jazyce / Report title

Modification of control software for robotic arm drive

Anotace v anglickém jazyce / Abstract

This research report focuses on editing control software for robotic arm drive. There is a detailed description of the problem-solving and troubleshooting that occurred while debugging the control software. The problems mentioned interfered with the smooth operation of the drive and accidentally caused the inverter fault states. Additionally, this report describes modifications to the control software that the customer has designed. This included editing existing code, modifying CAN communications, and adding several additional features.

Klíčová slova v anglickém jazyce / Keywords

Drive, software, CAN, Cycle counter, Process data watchdog

Seznam symbolů a zkratek

REMCS	Rice Embedded Modular Control System
PMSM	Synchronní motor s permanentními magnety
RPM	Rotation per minute
CAN	Sériový komunikační protokol

Obsah

1 Úvod

Robotické rameno je poháněno synchronním motorem s permanentními magnety. Motor je napájen z frekvenčního měniče. Celý pohon je řízen jednou řídicí kartou ze systému REMCS. Řídicí systém využívá pro regulaci PMSM vektorové řízení v kartézských souřadnicích. Vektorové řízení umožňuje řídit motor na úrovni proudové smyčky, rychlostní smyčky nebo zadávání momentu. Výběr pracovního režimu, zadávání požadavků a parametrů regulátorů se provádí přes komunikaci CAN pomocí CANoe Vector VN1640A CAN/LIN Interface viz [1].

Tato zpráva je rozdělena na dvě části. V první polovině budou popsány problémy, které se vyskytly během ladění řídicího softwaru. Nejpodstatnějšími problémy bylo kmitání otáček a náhodné proudové špičky při běhu pohonu. V dalších kapitolách této zprávy budou rozvedeny zmiňované problémy a jejich řešení.

V druhé polovině této zprávy budou popsány úpravy řídicího softwaru, které byly požadovány zákazníkem. Jednalo se o úpravy stávajícího kódu, doplnění funkcí CycleCounter a Process data watchdog. Také zde bylo řešeno ukládání parametrů regulátorů do paměti dataflash a úpravy v komunikaci CAN.

2 Kmitání otáček motoru

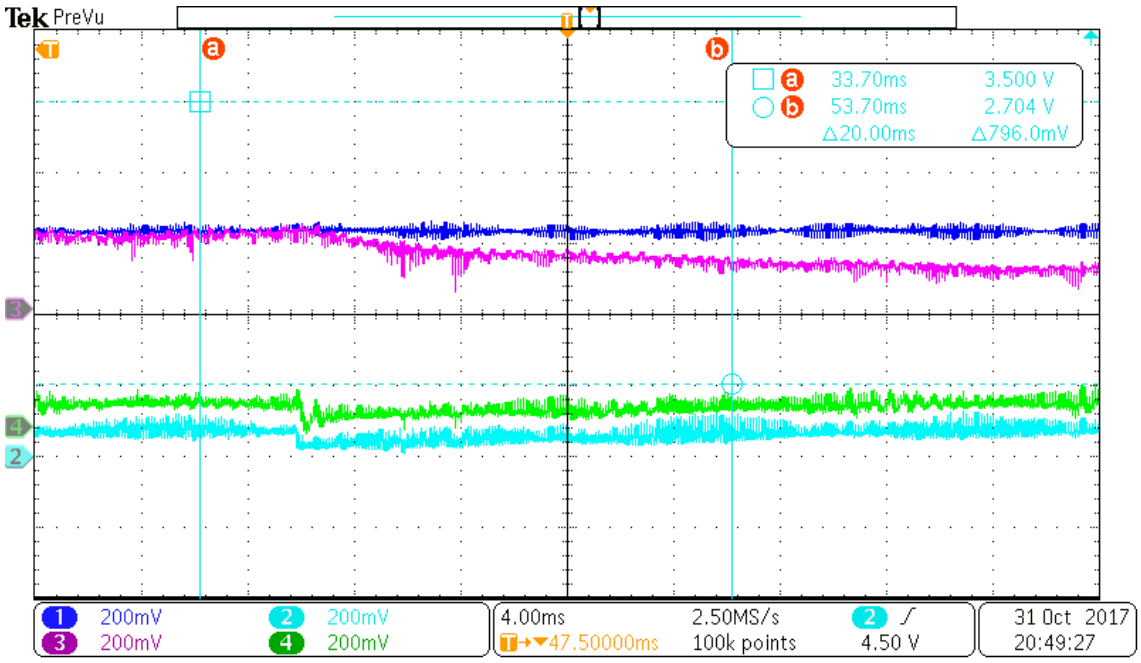
Kmitání otáček motoru se objevovalo ve dvou variantách. První varianta byla periodické kmitání otáček motoru a druhá varianta byla kmitání otáček při zatížení motoru. Obě zmiňované varianty kmitání jsou popsány v následujících kapitolách.

2.1 Periodické kmitání otáček motoru

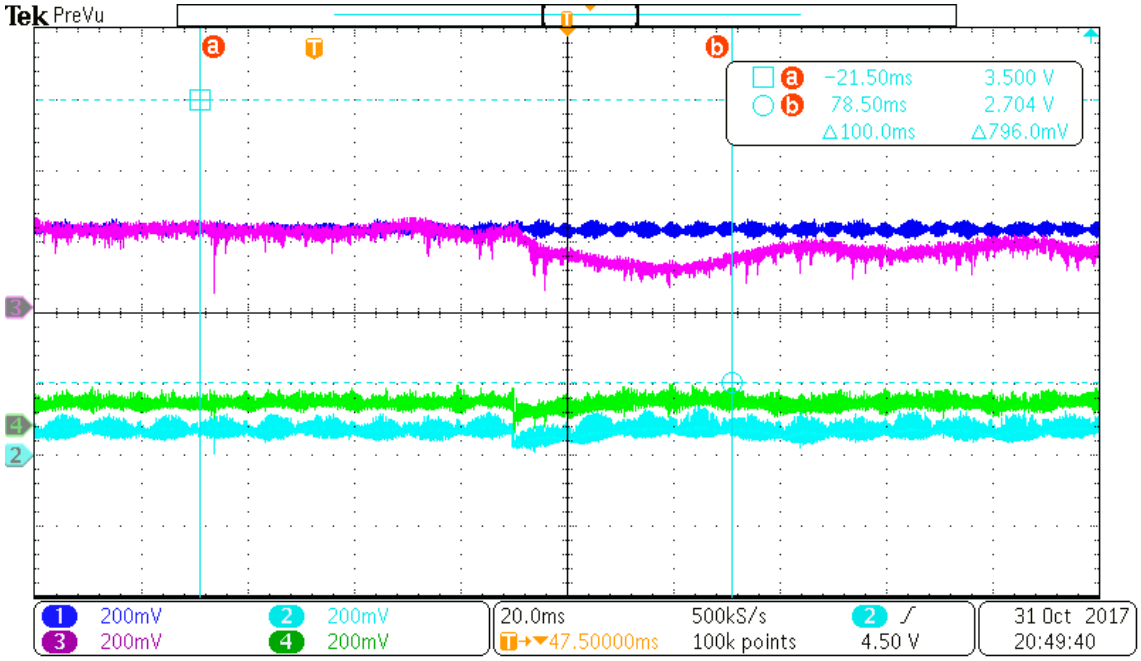
Periodické kmitání otáček se projevovalo jak u zatíženého, tak i nezatíženého motoru. Perioda kmitání byla přibližně 1s. Toto kmitání se projevovalo pouze během regulace rychlosti. V okamžiku, kdy byla nadřazená regulace rychlosti vypnuta a motor byl řízen pouze proudovou smyčkou, toto kmitání nenastávalo.

Průběh kmitání lze popsat takto. Pomocí uživatelského prostředí byl zadáván požadavek na konstantní rychlost motoru. Vždy po 1s došlo nejprve ke skokové změně požadavku na proud i_{sq}. Na tento požadavek zareagovala proudová smyčka a přibrzdila motor. Tím poklesly otáčky. Jelikož byl po celou dobu požadavek na otáčky konstantní, rychlostní smyčka začala regulovat pokles otáček. Tím vzniklo zakolísání rychlosti motoru, které se jevilo jako pravidelné zasekávání pohonu. Skoková změna požadavku na proud a způsobený pokles otáček jsou zobrazeny na Obr. 2.1 a Obr. 2.2.

Význam jednotlivých průběhů: tmavě modrá křivka = požadované otáčky, fialová křivka = změřené otáčky, zelená křivka = změřený proud i_{sq}, světle modrá křivka = požadovaný proud i_{sq}. Jednotlivé průběhy jsou vnitřní veličiny v procesoru, které jsou vykresleny z DA převodníku řídicí karty REMCS.



Obr. 2.1 Detail skokové změny požadavku na proud i_{sq}



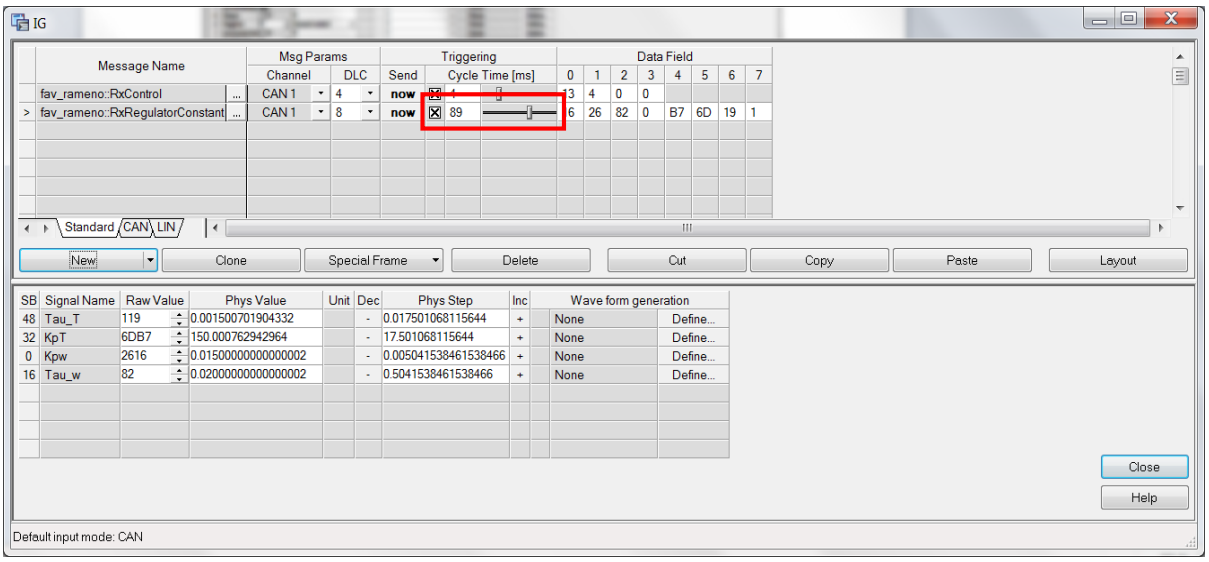
Obr. 2.2 Detail zakmitání otáček po skokové změně proudu i_{sq}

Řešením tohoto problému byla úprava řídicího softwaru. Konkrétně se jednalo o část kódu, ve které probíhal výběr pracovního režimu pohonu mezi regulací rychlosti, regulací momentu a regulací proudu i_{sq}. V případě režimu regulace rychlosti docházelo k dvojitému spuštění výpočtu vektorového řízení. To způsobovalo, že po skončení výpočtu vektorového řízení se spouštěl výpočet znovu. Následek tohoto dvojitého spouštění vektorového řízení způsobil, že v některých případech nebyl dokončen výpočet požadovaného proudu. Tím vznikla skoková změna požadavku na proud i_{sq}. Tato část kódu byla nahrazena níže uvedeným kódem. Tím byl vyřešen problém s periodickým kmitáním otáček motoru.

```
switch(regime)
{
case CAN_TORQUE_REQUEST:
  {
    VC_Input.Required_value = CAN_req*2.37;
    VC_Par.Isqw_Calcul.ModeDef = TORQUE_REQUEST;
    VC_control1b.Calcul(VC_Input, VC_Output, VC_Par);
    break;
  }
case CAN_ISQ_REQUEST:
  {
    VC_Input.Required_value = CAN_req;
    VC_Par.Isqw_Calcul.ModeDef = ISQ_REQUEST;
    VC_control1b.Calcul(VC_Input, VC_Output, VC_Par);
    break;
  }
case CAN_SPEED_CONTROL:
  {
    VC_Input.Required_value = CAN_req;
    VC_Par.Isqw_Calcul.ModeDef = SPEED_CONTROL;
    VC_control1b.Calcul(VC_Input, VC_Output, VC_Par);
    break;
  }
default:
  {

  }
}
```

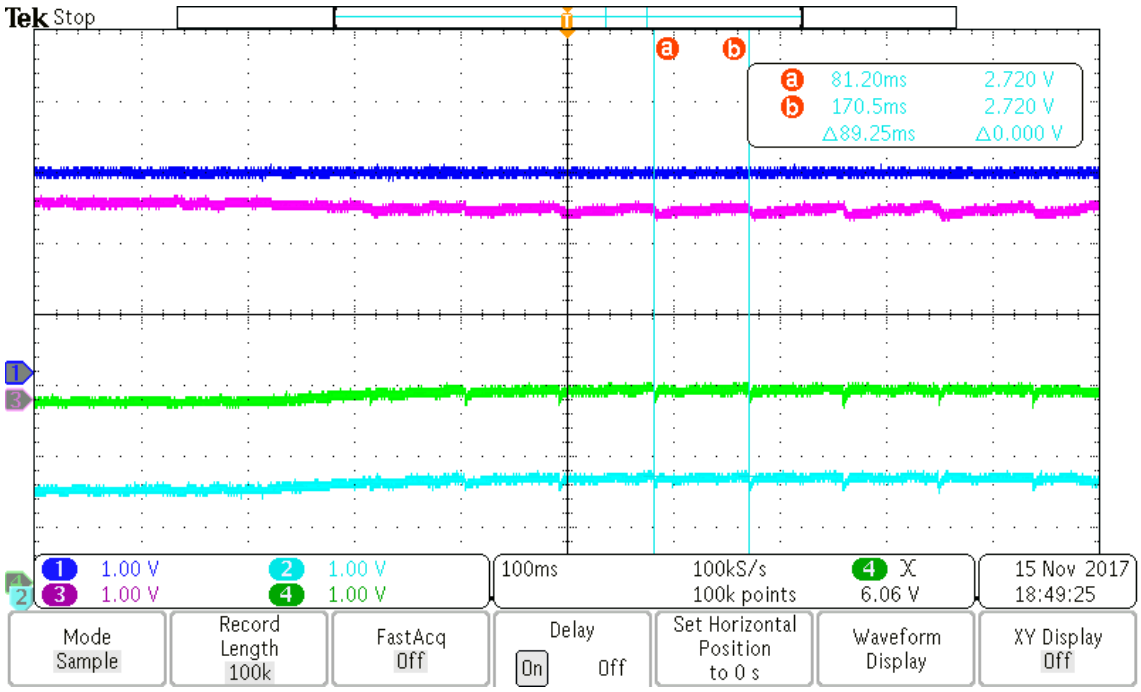
2.2 *Kmitání otáček zatíženého motoru*

Tato porucha byla zjištěna až po vyřešení problému periodického kmitání otáček motoru, který byl popsán v předchozí kapitole. Toto kmitání otáček se projevovalo pouze při zatížení motoru. Při běhu naprázdno nebylo pozorováno. Testováním bylo zjištěno, že kmitání otáček zatíženého motoru nastává pouze v případě, kdy je zapnuté periodické odesílání parametrů regulátorů z uživatelského rozhraní přes CAN do procesoru. Perioda kmitání byla totožná s časovou smyčkou komunikace CAN, ve které se posílaly parametry regulátorů. Periodické posílání parametrů regulátorů sice uživatelské prostředí umožňuje, ale v mikroprocesoru nebylo přijímání parametrů ošetřené.

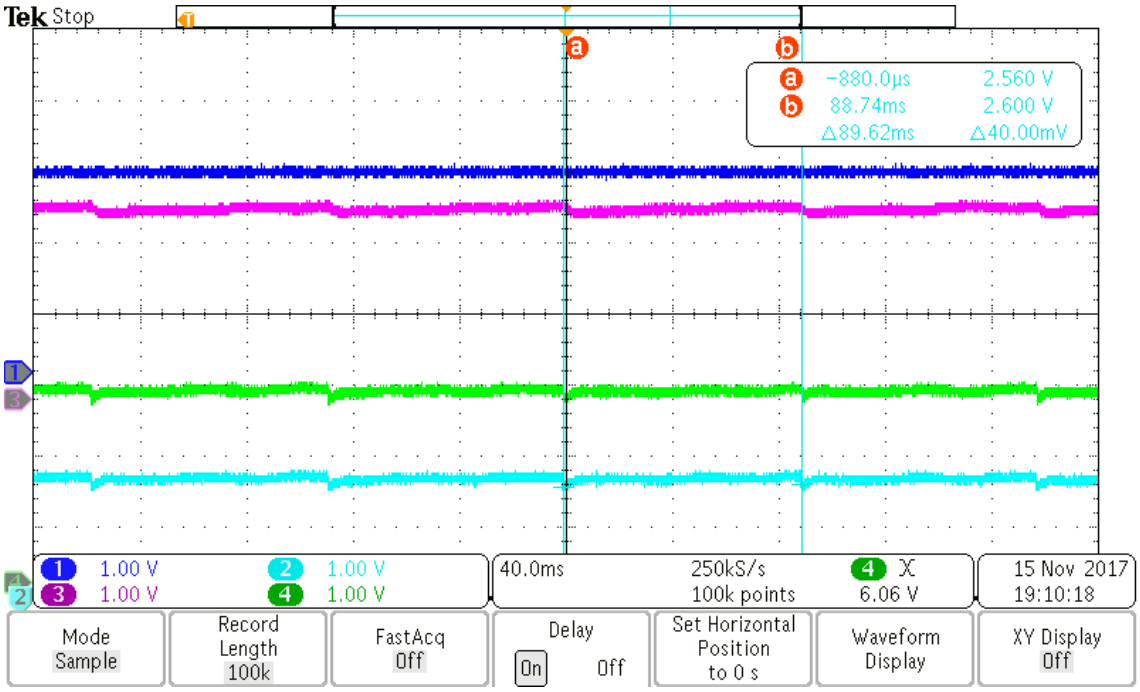


Obr. 2.3 Volba periody posílání parametrů regulátorů v uživatelském prostředí

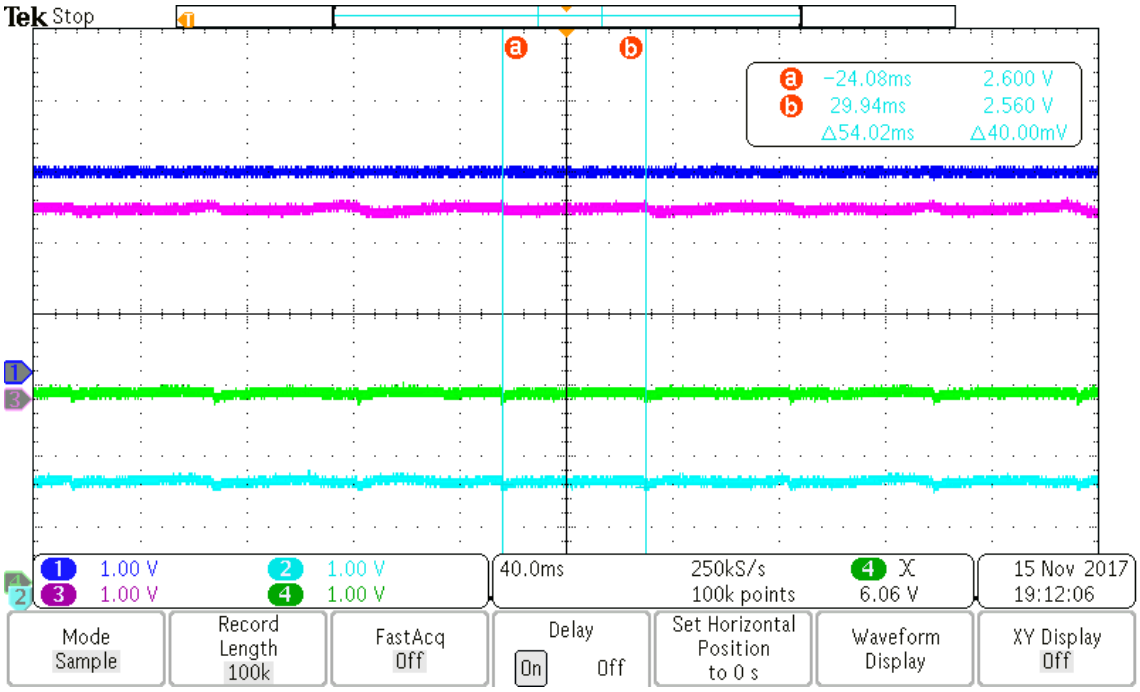
Kmitání otáček zatíženého motoru se projevovalo ve skutečných otáčkách, požadovaném i měřeném proudu. Požadovaná rychlost byla při testování konstantní. Popisované kmitání je zobrazeno na následujících obrázcích.



Obr. 2.4 Přechod z nezatíženého motoru na zatížení, perioda kmitání 89,62ms



Obr. 2.5 Zatížený motor, perioda kmitání 89,62ms



Obr. 2.6 Zatížený motor, perioda kmitání 54,02ms

Na obrázku Obr. 2.4 jsou vidět průběhy požadovaných a skutečných otáček a proudů při náhlém zatížení motoru. Z obrázku je vidět, že kmitání nastalo až po zatížení. Na obrázcích Obr. 2.5 a Obr. 2.6 je znázorněna odlišná perioda kmitání otáček. Na Obr. 2.5 je perioda kmitání 89,62ms zatímco na Obr. 2.6 je perioda kmitání 54,02ms.

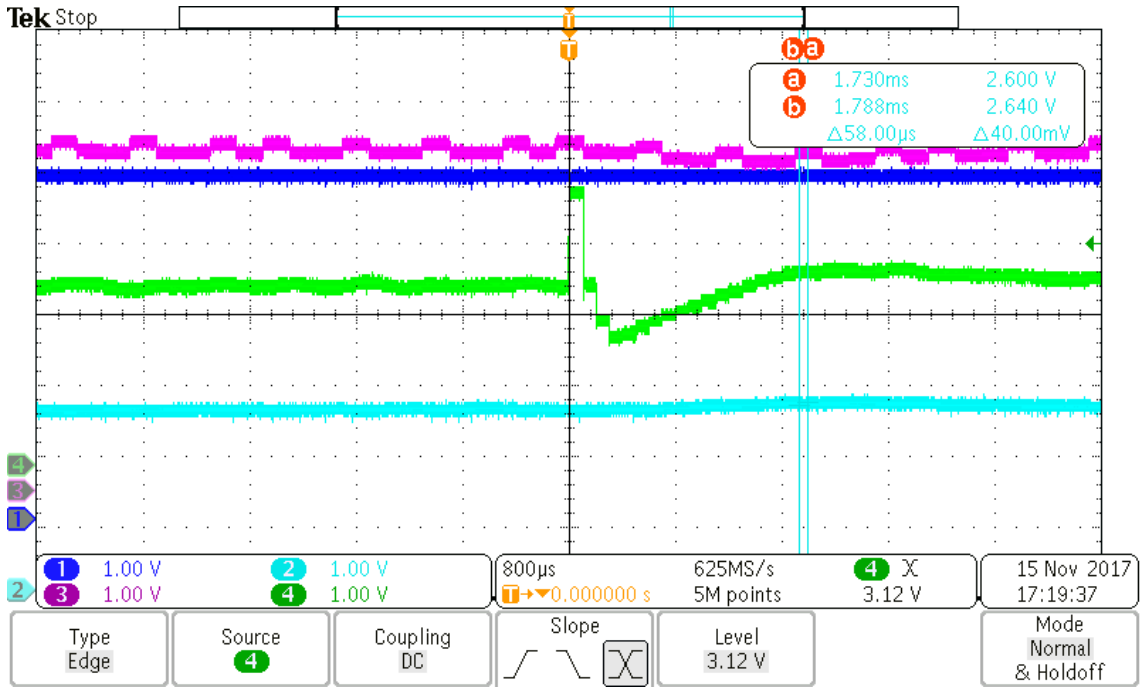
Význam jednotlivých průběhů je stejný jako v předchozí kapitole: tmavě modrá křivka = požadované otáčky, fialová křivka = změřené otáčky, zelená křivka = změřený proud i_{sq}, světle modrá křivka = požadovaný proud i_{sq}. Jednotlivé průběhy jsou vnitřní veličiny v procesoru, které jsou vykresleny z DA převodníku řídicí karty REMCS.

Řešením tohoto problému bylo doplnění řídicího kódu o pomocné proměnné, do kterých se ukládají přijaté parametry regulátorů z CANu. Nově přijaté hodnoty parametrů regulátorů jsou v těchto pomocných proměnných drženy až do začátku výpočtu nové smyčky regulace. Na začátku nové smyčky regulace se nahradí původní parametry regulátorů za nové a spustí se výpočet. Tímto ošetřením je zajištěno, že nedochází k přepsání parametrů regulátorů během výpočtu regulační smyčky, ale vždy na jejím počátku.

3 Náhodné proudové špičky

Během činnosti pohonu se proudové špičky objevovaly nezávisle na poruchách, které byly popsány v předchozích kapitolách. Proudové špičky se objevovali zcela náhodně a nezávisle na vybraném pracovním režimu regulace pohonu. Proudové špičky způsobovaly momentové rázy.

Zmiňované proudové špičky jsou zachyceny na následujících obrázcích. Na obrázku Obr. 3.1 je znázorněna zachycená proudová špička, která byla vykreslena na výstupech DA převodníku řídicí karty REMCS. Na tomto obrázku jsou tedy zachyceny vnitřní veličiny procesoru. Význam jednotlivých průběhů: fialová křivka = měřené otáčky, tmavě modrá křivka = požadované otáčky, zelená křivka = měřený proud a světle modrá křivka = požadovaný proud. Z obrázku je patrné, že proudová špička vznikla v okamžiku, kdy byly požadavky na otáčky a proud konstantní.

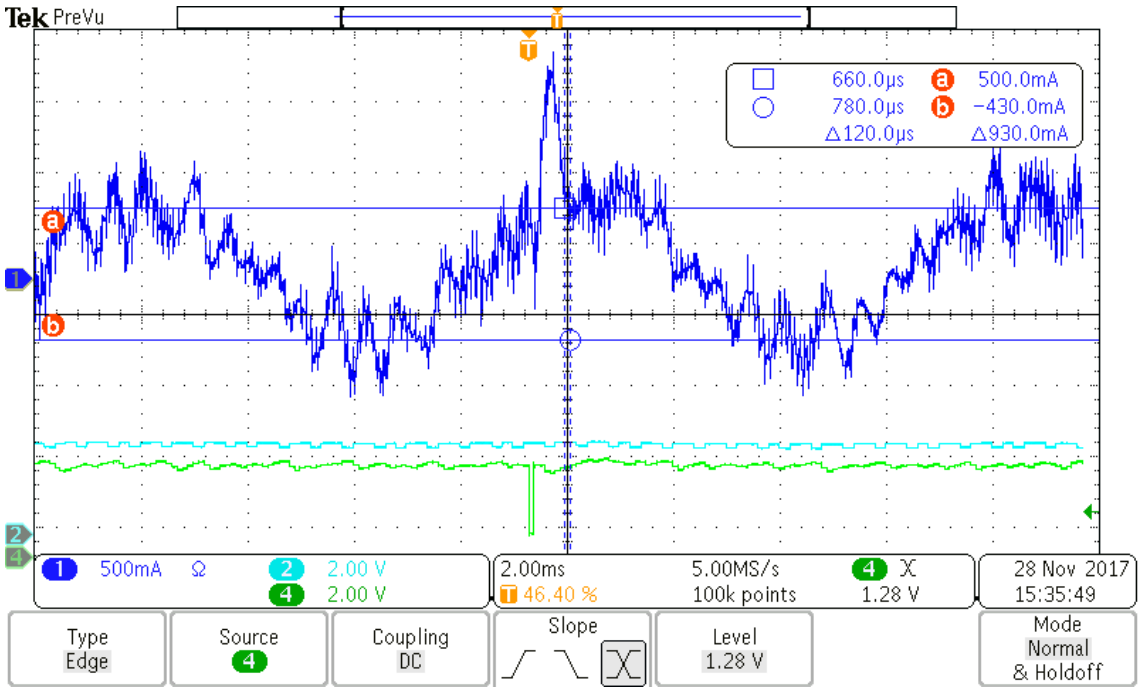


Obr. 3.1 Proudová špička zachycená na DA výstupech řídicí karty REMCS

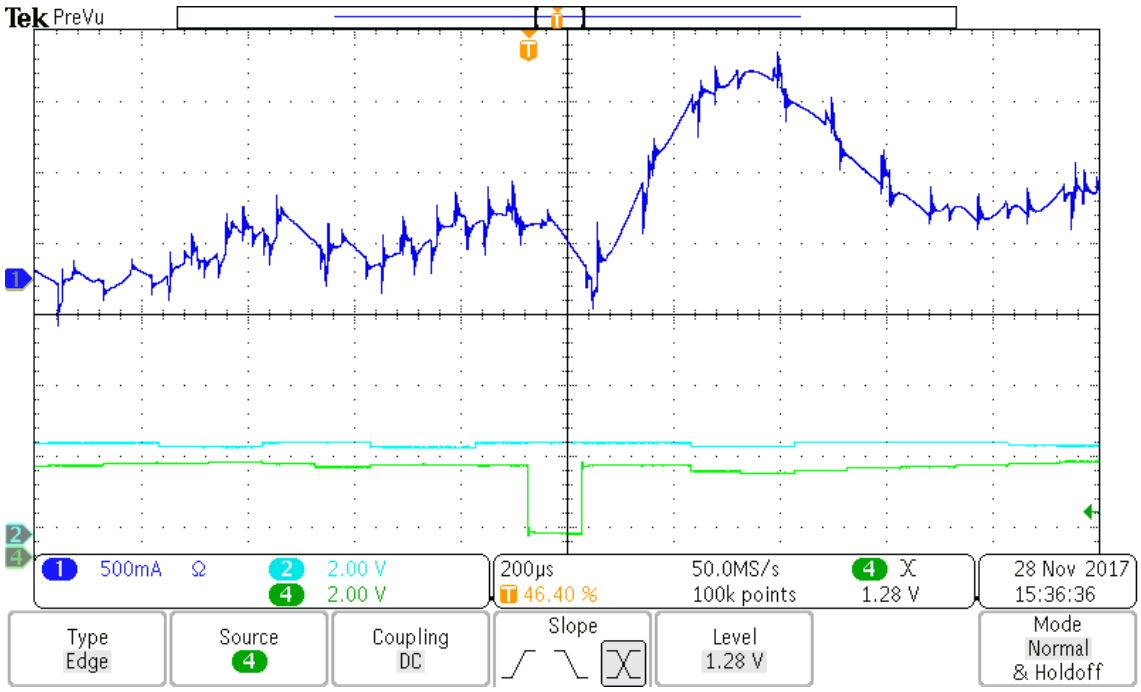
Na obrázku Obr. 3.2 je znázorněna proudová špička, která byla změřena osciloskopem a proudovými kleštěmi ve fázovém proudu i_{su}. Na obrázku Obr. 3.3 je zachycen detail špičky z obrázku Obr. 3.3.

Význam jednotlivých průběhů: tmavě modrá křivka = změřený skutečný proud ve fázi u, zelená křivka = měřený proud a světle modrá křivka = požadovaný proud. Požadovaný a

měřený proud jsou vykresleny z DA převodníku řídicí karty REMCS, tudíž se jedná o vnitřní veličiny procesoru.



Obr. 3.2 Proudová špička změřená ve fázovém proudu i_{su}

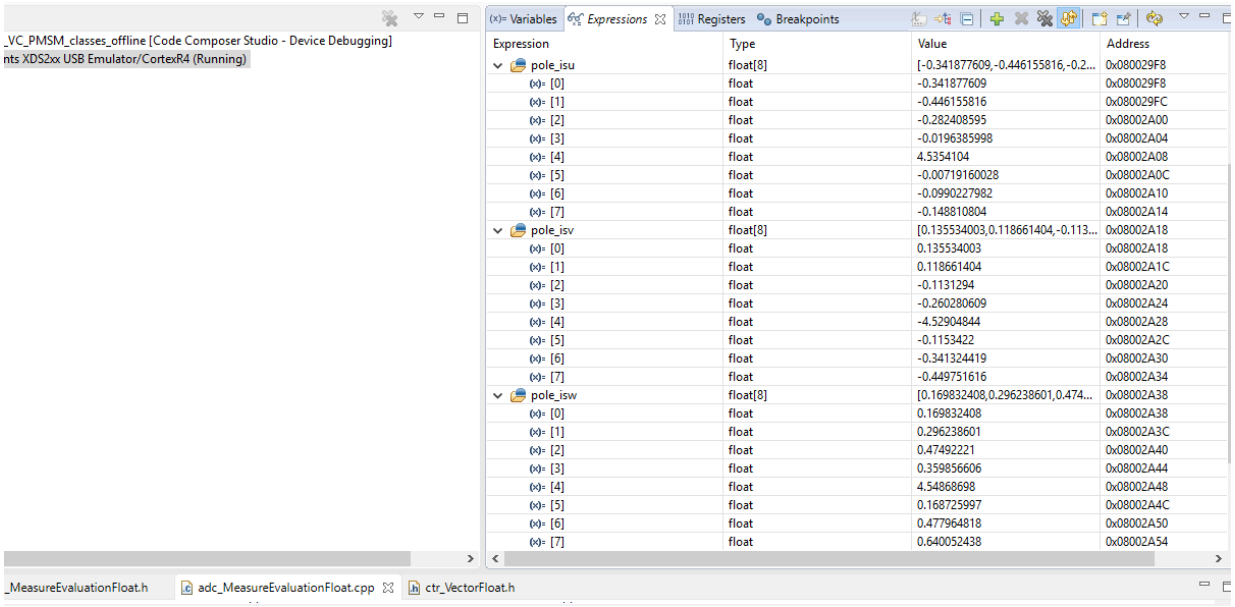


Obr. 3.3 Detail proudové špičky změřené ve fázovém proudu i_{su}

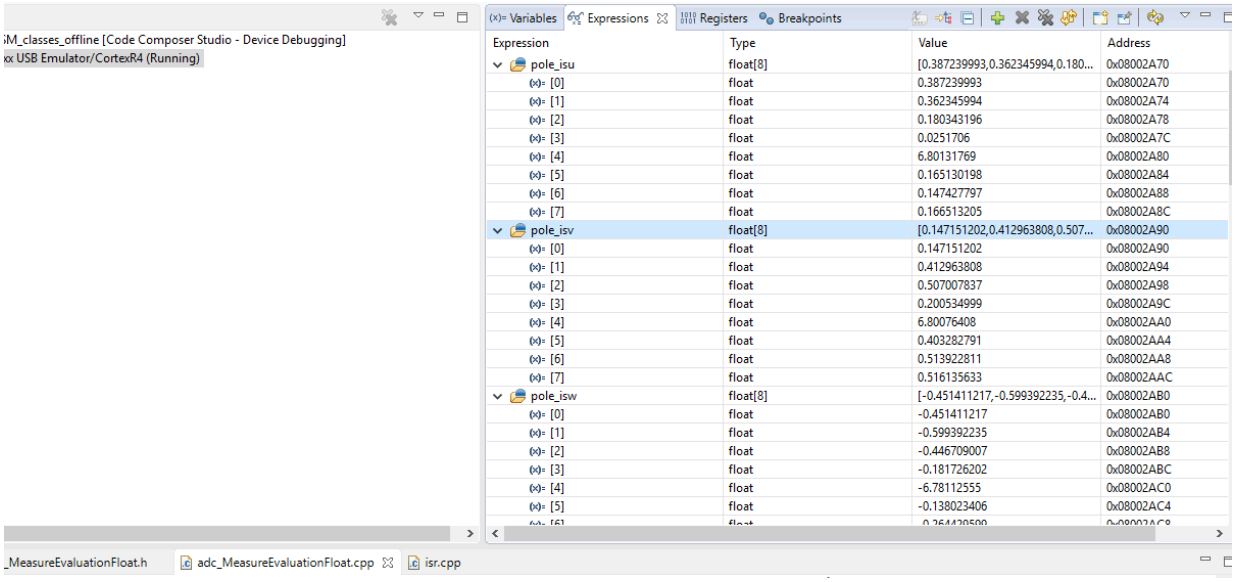
3.1 Hledání příčiny proudových špiček

Příčina proudových špiček byla hledána jak v softwaru procesoru, tak i v designu FPGA. V procesoru byly vytvořeny měřící buffery, které zachytávaly vnitřní veličiny procesoru. Konkrétně se jednalo o měřené proudy všech fází. Měřící buffery byly nastaveny tak, aby zaznamenaly tři hodnoty měřeného proudu před výskytem proudové špičky a tři hodnoty měřeného proudu po výskytu proudové špičky.

Na obrázcích Obr. 3.4 a Obr. 3.5 jsou znázorněny zaznamenané hodnoty změřených proudů, které jsou přepočteny na reálné veličiny. Z obrázků je vidět, že při chybném měření byly absolutní hodnoty proudů ve všech fázích buď 4,53A (Obr. 3.4, index bufferu [4]) nebo 6,8A (Obr. 3.5, index bufferu [4]).

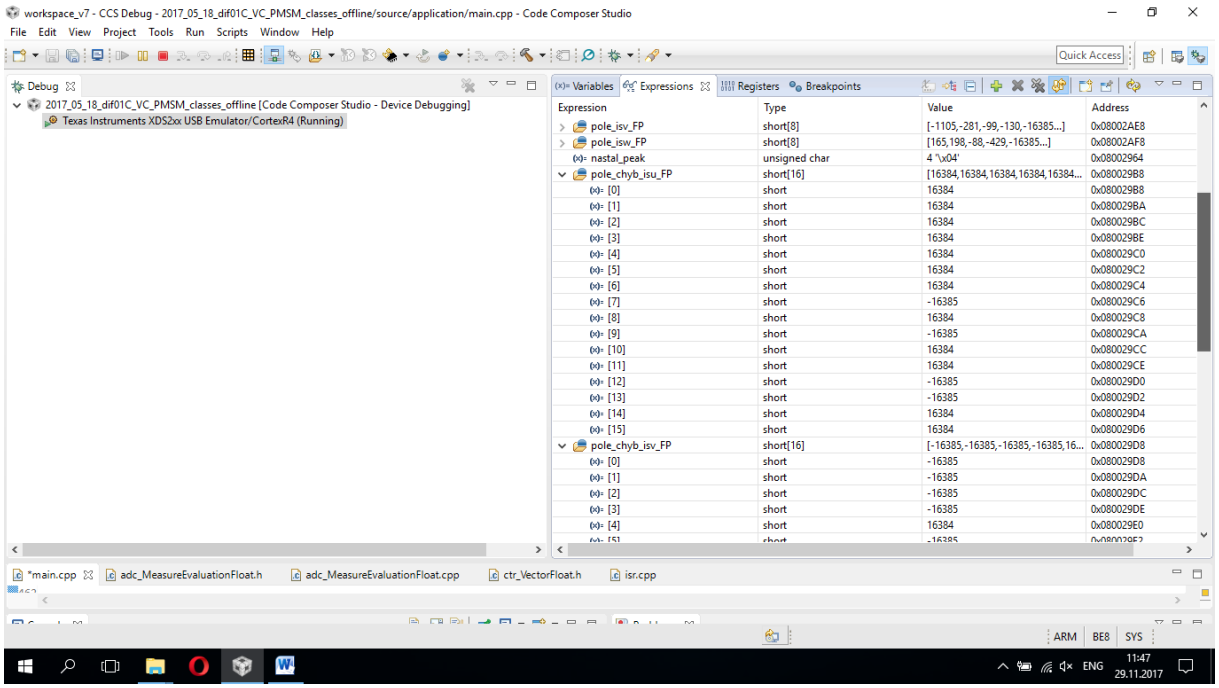


Obr. 3.4 Obsah měřících bufferů při chybném měření (hodnota 4,53A)

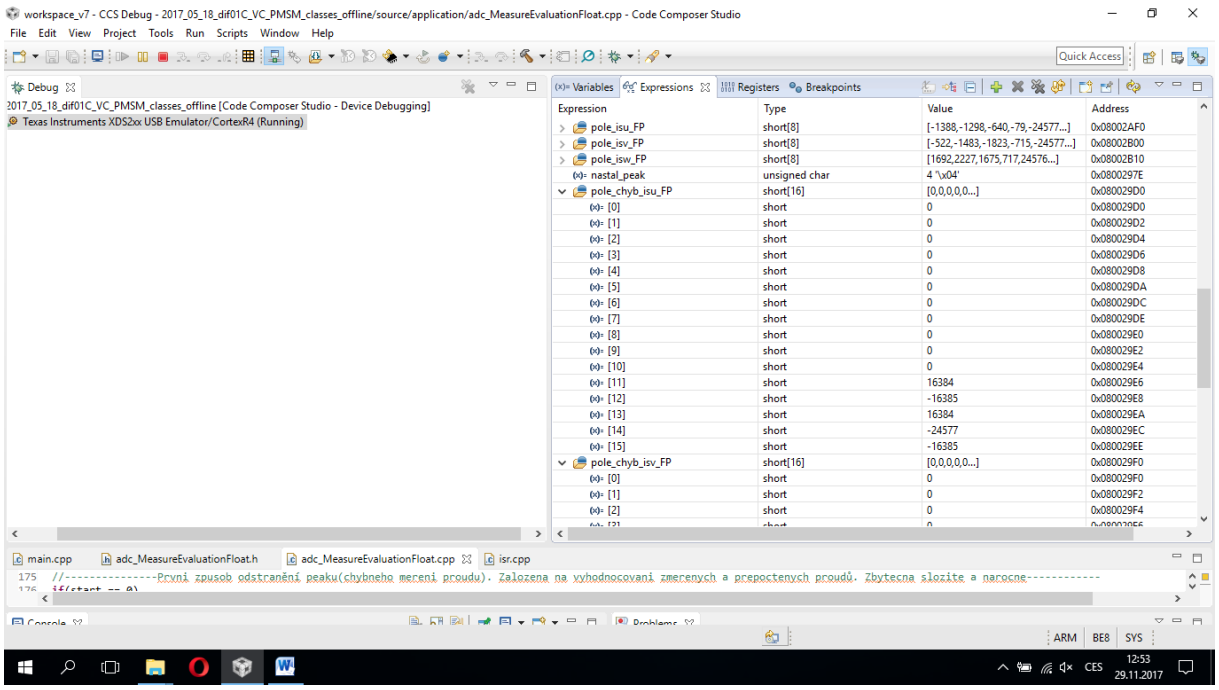


Obr. 3.5 Obsah měřících bufferů při chybném měření (hodnota 6,8A)

Na obrázcích Obr. 3.6 a Obr. 3.7 jsou zaznamenána data, která byla vyčtena přímo z registrů AD převodníků. Hodnoty byly zaznamenávány do bufferů pouze v případě, kdy došlo k chybnému měření. Z naměřených hodnot je vidět, že chybnému měření 4,53A odpovídá hodnota 16384 (případně -16385 -> rozdíl „1“ vychází ze dvojkového doplňku) a chybnému měření 6,8A odpovídá hodnota 24576 (případně -24577).



Obr. 3.6 Pole dat vyčtených z registrů AD převodníků (chybné měření 4,53A)



Obr. 3.7 Pole dat vyčtených z registrů AD převodníků (chybné měření 5,8A)

3.2 *Eliminace náhodných proudových špiček*

Pro eliminaci náhodných proudových špiček byl vytvořen níže uvedený filtr. Tento filtr testuje, jestli změřené hodnoty fázových proudů překročí hranice definované hodnotami v makrech THRESHOLD_POSITIVE_PEAK_CURRENT_ADC_VALUE a THRESHOLD_NEGATIVE_PEAK_CURRENT_ADC_VALUE. Pokud při AD převodu dojde k překročení této hranice, znamená to, že buď nastal nadproud, nebo došlo k chybnému měření. Detekce chybnému měření je vyhodnocena faktem, že všechny změřené hodnoty proudů se v absolutní hodnotě rovnají. Pokud tato rovnost nastane, jsou pro regulační smyčku použity změřené hodnoty proudů z předchozího taktu. Pokud nastane případ, že bude nadproud vyhodnocen ve dvou následujících taktech, znamená to, že na zátěži je skutečný nadproud a dojde k zablokování PWM výstupů. Kód filtru náhodných špiček změřených proudů je znázorněn níže.

```
/Thresholds in peaks filter
#define THRESHOLD_POSITIVE_PEAK_CURRENT_ADC_VALUE 12644    //[-] Positive level of ADC number of measured current,
                                                           whenever is suspicion of failure reading of data
#define THRESHOLD_NEGATIVE_PEAK_CURRENT_ADC_VALUE -12644   //[-] Negative level of ADC number of measured current,
                                                           whenever is suspicion of failure reading of data
                                                           //12644 odpovida hodnote proudu cca 3.5A, coz je nad
                                                           jmenovitym proudem

//---proměnné pro filtr peaků--- ---------------
  int16_t isu_FP = 0;
  int16_t isv_FP = 0;
  int16_t isw_FP = 0;
  int16_t isu_FP_abs = 0;
  int16_t isv_FP_abs = 0;
  int16_t isw_FP_abs = 0;
  static int16_t isu_FP_predchozi = 0;
  static int16_t isv_FP_predchozi = 0;
  static int16_t isw_FP_predchozi = 0;
  static uint16_t pocitadlo = 0;
//-----------------------------------------------
  static int16_t adc_offset1=-12,adc_offset2=-10,adc_offset3=-60,adc_offset4=-25;   //correction sensor offsets

//Vyctení z registrů AD prevodniku
  isu_FP = ((int16_t)Data->Adc0);
  isv_FP = ((int16_t)Data->Adc4);
  isw_FP = ((int16_t)Data->Adc2);

//Vypocet abs. hodnot zemerenych proudu
  if(isu_FP < 0)
  {
      isu_FP_abs = ((isu_FP * (-1)) - 1);
  }
  else
  {
      isu_FP_abs = isu_FP;
  }

  if(isv_FP < 0)
  {
      isv_FP_abs = ((isv_FP * (-1)) - 1);
  }
  else
  {
      isv_FP_abs = isv_FP;
  }

  if(isw_FP < 0)
  {
      isw_FP_abs = ((isw_FP * (-1)) - 1);
  }
  else
  {
      isw_FP_abs = isw_FP;
  }

//Peak filter
  if((isu_FP > THRESHOLD_POSITIVE_PEAK_CURRENT_ADC_VALUE) || (isv_FP > THRESHOLD_POSITIVE_PEAK_CURRENT_ADC_VALUE) ||
(isw_FP > THRESHOLD_POSITIVE_PEAK_CURRENT_ADC_VALUE) || (isu_FP < THRESHOLD_NEGATIVE_PEAK_CURRENT_ADC_VALUE) ||
(isv_FP < THRESHOLD_NEGATIVE_PEAK_CURRENT_ADC_VALUE) || (isw_FP < THRESHOLD_NEGATIVE_PEAK_CURRENT_ADC_VALUE))
  {
      pocitadloX++;

      if((isu_FP_abs == isv_FP_abs)&&(isu_FP_abs == isw_FP_abs)&&(isv_FP_abs == isw_FP_abs)) //Nastalo chybne mereni
```

```
    {
        pocitadlo++;

        if(pocitadlo == 2)    //nastal nadproud -> zablokuj PWM (kód spadne do chyboveho stavu)
        {
            fpg_BpcWriteSwBlck(fpg_BpcGetSwBlck_u32() | 0x00fc);//pwm outs are blocked
            Drive_error=OVERCURRENT;
            fpg_LedOff(FPG_LED_RUN);
        }

        isu_FP = isu_FP_predchozi;        //pro tento takt pouzij predchozi zmerene hodnoty proudu
        isv_FP = isv_FP_predchozi;
        isw_FP = isw_FP_predchozi;
    }
    else
    {
        isu_FP_predchozi = isu_FP;        //Uloz zmerene hodnoty pro dalsi takt
        isv_FP_predchozi = isv_FP;
        isw_FP_predchozi = isw_FP;
    }
}
else
{
    pocitadlo = 0;        //nuluj pocitadlo, v predchozim taktu nastalo chybne mereni (nenastal realny nadproud)
    isu_FP_predchozi = isu_FP;    //Uloz zmerene hodnoty pro dalsi takt
    isv_FP_predchozi = isv_FP;
    isw_FP_predchozi = isw_FP;
}


adc_InvMeasuredValuesFloat.isu=-13.83e-4 * NUMBER_OF_PASSES *((float)(isu_FP + adc_offset1));
adc_InvMeasuredValuesFloat.isv=-13.83e-4 * NUMBER_OF_PASSES *((float)(isv_FP + adc_offset2));
adc_InvMeasuredValuesFloat.isw=-13.83e-4 * NUMBER_OF_PASSES *((float)(isw_FP + adc_offset3));
```

4 Modifikace softwaru

Na přání zákazníka bylo nutné upravit a doplnit řídicí software měniče. Jednotlivé požadavky jsou uvedeny níže:

1) *Jak je řešen offset čidla vůči motoru pro komutaci, je nastavený napevno? Máte nějaký postup jak ho nastavit pro konkrétní motor?*

2) Měnič neposílá aktuální moment, jak jsme se domlouvali, a jak je zmíněné i v dokumentaci, ale rychlost.

3) CAN zpráva TxConverterState: Poloha je zdá se jen na 24 bitech, nejvyšších 8 bitů polohy je pořád 0. Víceotáčkový enkodér má 20 bitů/ot + 12 bitů multiturn, takže má význam celých 32 bitů.

4) CAN zpráva TxConverterState: Podle obrázku je obsahem RPM, což odpovídá skutečnosti, ale hodilo by se spíš posílat moment (i v režimu řízení rychlosti). Ideálně umožnit přepínat obsah této zprávy (moment/rychlost) pomocí nějakého bitu ve zprávě RxControl.

5) CAN zpráva TxConverterState: Po vypnutí měniče zůstává hodnota RPM posílaná přes CAN na konstantní hodnotě v okamžiku vypnutí. Měla by se počítat dále, poloha se normálně posílá, a motor se může otáčet vnější silou.

6) Do CAN zprávy TxConverterState by se hodilo přidat "CycleCounter" - na 4 bitech, např. bity 60-63, který by se s každou odeslanou CAN zprávou zvyšoval o 1. Na druhé straně pak lze spolehlivě detekovat výpadek komunikace a časovou konzistenci zpráv.

7) V CAN zprávách TxRegulatorConstant vs. RxRegulatorConstant je rozdílné pořadí parametrů - může být, ale je to matoucí a samozřejmě jsme to poprvé spletli...

8) Zdá se, ze RxRegulatorConstant se ukládají do nějaké trvalé paměti - změna zůstává i po vypnutí. Když tam nastavím něco nefunkčního nemám se jak vrátit na výchozí stav, pokud jsem si ho nezapsal, ve zprávě jsem výchozí hodnoty taky nenašel. Preferovali bychom uložení hodnot až na explicitní požadavek, např. nějakým signalizačním bitem v RxControl.

9) Chybí process data watchdog - pokud po nějakou dobu (10-100 ms) nepřijde RxControl zpráva, měl by se měnič vypnout a signalizovat chybu "ztráta komunikace".

4.1 Offset polohového čidla

Postup nastavení offsetu polohového čidla je následovný. Ve vektorovém řízení nastavíme jmenovitou hodnotu tokotvorného proudu Isd a momentotvorný proud Isq nastavíme nulové. Tímto nabudíme motor. Magnetické pole vzniklé nabuzením vtáhne rotor do nulové elektrické polohy. Pootočení rotoru (z mechanické nulové polohy do elektrické nulové polohy) definuje offset čidla polohy.

4.2 Posílání aktuálního momentu po CANu

Požadavkem v bodech zadání č. 2 a č. 4 bylo posílat aktuální moment po CANu, nebo alespoň umožnit přepínání mezi odesíláním otáček RPM nebo odesíláním momentu.

Řešením těchto bodů byla úprava komunikace CAN tak, aby byl moment i otáčky RPM odesílány neustále a nezávisle na pracovním režimu měniče. Byla vytvořena nová zpráva TxMotorState s ID 0x15. Do této zprávy byly přesunuty signály Position_RAW a RPM, které se původně odesílaly ve zprávě TxConverterState. Ve zprávě TxMotorState byl vytvořen nový signál Torque, ve kterém je uložen aktuální moment. Tímto byly vyřešeny body zadání č. 2 a č. 4.

Úpravy v kódu:

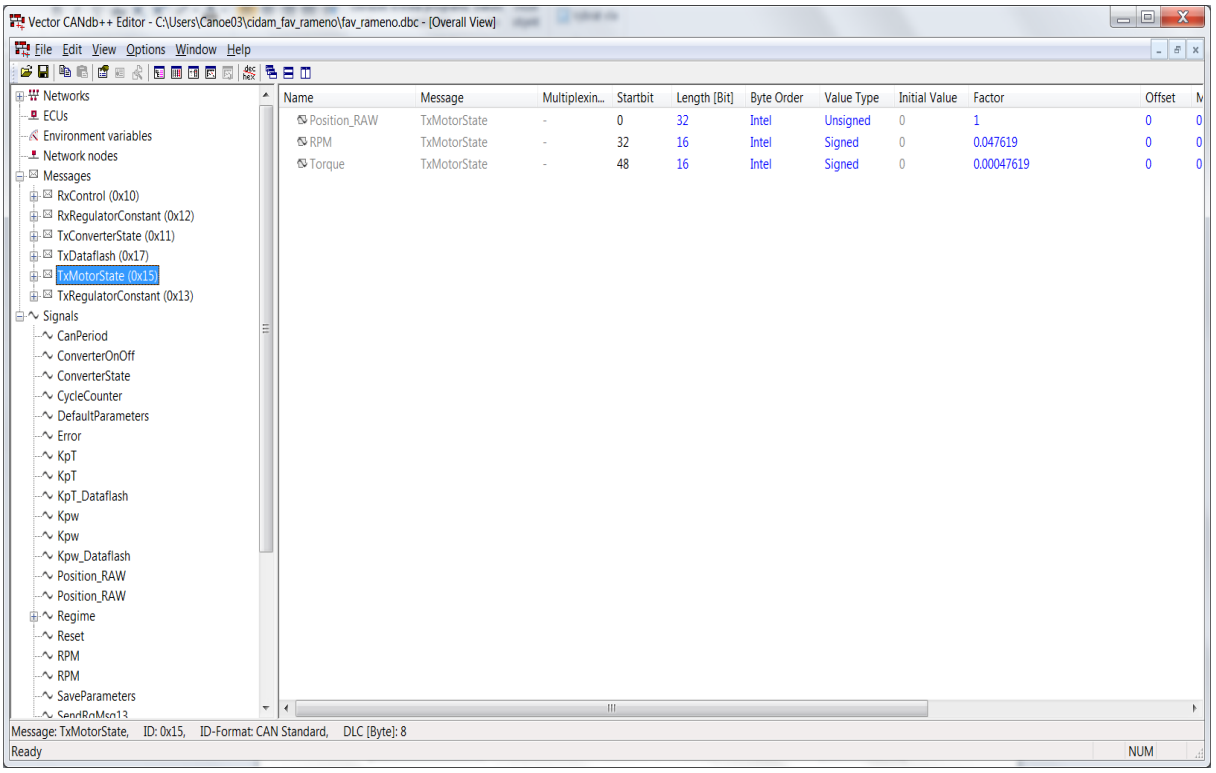
Ctr_VectorFloat.h:

```
#define SCALE_RPM 21        //Scale pro otacky v TxMotorState = 21 * 1480ot/min = 31080 (je zde
                              dostatecna rozerva do 32768)
#define SCALE_Torque 2100   //Scale pro moment v TxMotorState = 2100 * 14.4 = 30240 (je zde dostatecna
                              rozerva do 32768)
```

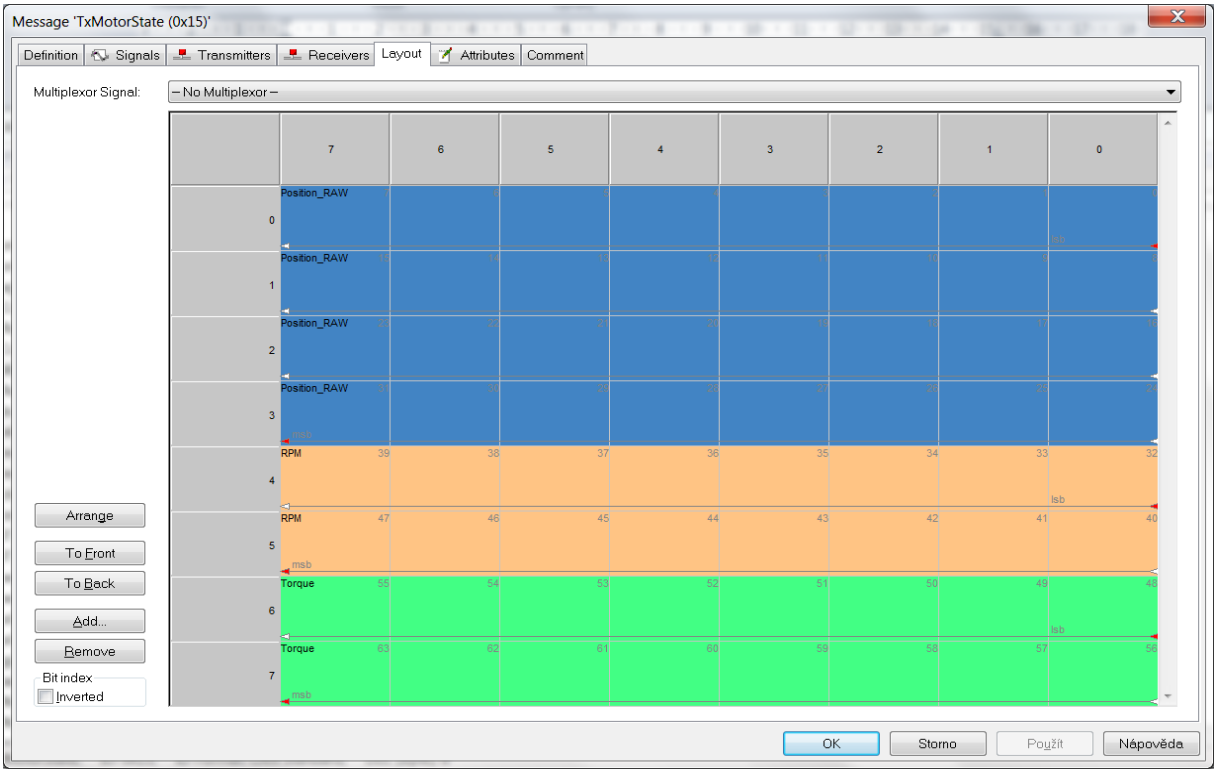
main.cpp:

```
//CAN init

 /* MB 5, TX */                          //Inicializace zprávy TxMotorState
 MbConf.Id_u32 = 0x15u;
 MbConf.Dir_e = DEV_CAN_MSG_TX;
 MbConf.Dlc_u8 = 8u;
 MbConf.Mask_u32 = DEV_CAN_MASK_STD;
 MbConf.TypeId_e = DEV_CAN_ID_STD;
 MbConf.MsgBoxNr_u8 = 5u;
 dev_CanConfigMsgBox_e(DEV_CAN_IFACE_1, &MbConf);


//Upload zprávy TxMotorState
 ((int16_t *)&CanTx_u64)[0] = (int16_t)(VC_Output.Math_Model.Isq*KP*PP*FPM*0.20833f * SCALE_Torque);
 ((int16_t *)&CanTx_u64)[1] = nME_drive_i16 * SCALE_RPM;
 ((uint32_t *)&CanTx_u64)[1] = adc_InvMeasuredValuesFloat.theta_from_ARC;

 dev_CanTransmitData_e(DEV_CAN_IFACE_1, 5, CanTx_u64);  //Odeslání zprávy TxMotorState
```

Úpravy v CANoe:



Obr. 4.1 Signály v CAN zprávě TxMotorState



Obr. 4.2 Uspořádání zprávy TxMotorState

4.3 Posílání otáček RPM nezávisle na stavu měniče

V této kapitole bude popsáno řešení zadání č. 5. Dosavadní řešení bylo založeno na výpočtu hodnoty RPM z matematického modelu vektorového řízení. Vektorové řízení je počítáno v přerušení, tudíž je závislé na zapnutí a vypnutí měniče. Pokud je měnič vypnutý, nedochází k výpočtu vektorového řízení.

Tento bod zadání byl vyřešen tím, že výpočet hodnoty mechanických otáček rotoru RPM pro komunikace CAN není čten z vektorového řízení, ale hodnota RPM je vypočítávána v hlavní smyčce programu přímo ze změřené hodnoty úhlové mechanické rychlosti rotoru ω_{rm}. Úhlová mechanické rychlosti rotoru ω_{rm} je vypočítávána v rámci funkce „měření“, která běží nezávisle na vektorovém řízení.

Úpravy v kódu:

Ctr_VectorFloat.h, řádek: 112:

```
#define RECAL_Wrm_to_nrm 9.54929658551372           //Konstanta 9.54929658551372 = 60/(2*PI) -
> RECAL_Wrm_to_nrm vyjadřuje přepočet z mechanické rotorové omegy Wrm na mechanické otáčky za
min. nrm
```

main.cpp:

```
nME_drive_i16 = (adc_InvMeasuredValuesFloat.wrm * RECAL_Wrm_to_nrm);
```

4.4 Realizace funkce CycleCounter

Požadavkem v bodě č. 6 bylo doplnit řídicí software měniče o CycleCounter. CycleCounter je čítač, který čítá počet odeslaných zpráv z měniče po komunikaci CAN. Hodnota CycleCounteru je následně odesílána v jedné zprávě. Tím nadřazené řízení může detekovat jednak výpadek komunikace a také časovou konzistenci přijatých zpráv.

CycleCounter byl realizován statickou proměnnou typu uint8_t. Počáteční hodnota CycleCounteru je 255 a s každou odeslanou zprávou dochází ke snížení této hodnoty o jedna. V okamžiku, kdy se hodnota CycleCounteru rovná nule, dojde k přetečení a CycleCounter opět začíná na hodnotě 255. Dekrementace proměnné CycleCounter byla použita proto, aby na straně přijímače (CANoe v PC) se CycleCounter choval jako čítač, jehož hodnota se zvyšuje. Toto je způsobeno odlišnou endianitou procesoru v měniči a PC s CANoe(little a big endian). Hodnota CycleCounteru se posílá ve zprávě TxConverterState.

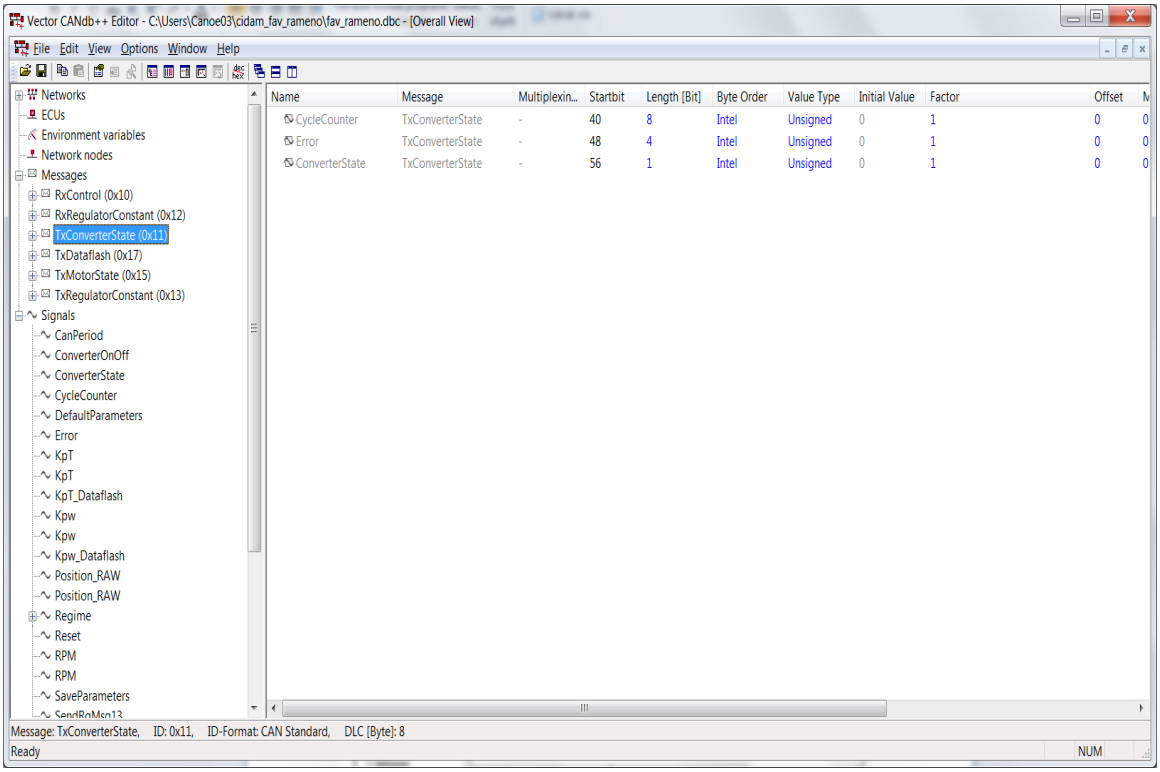
Úpravy v kódu:

main.cpp:

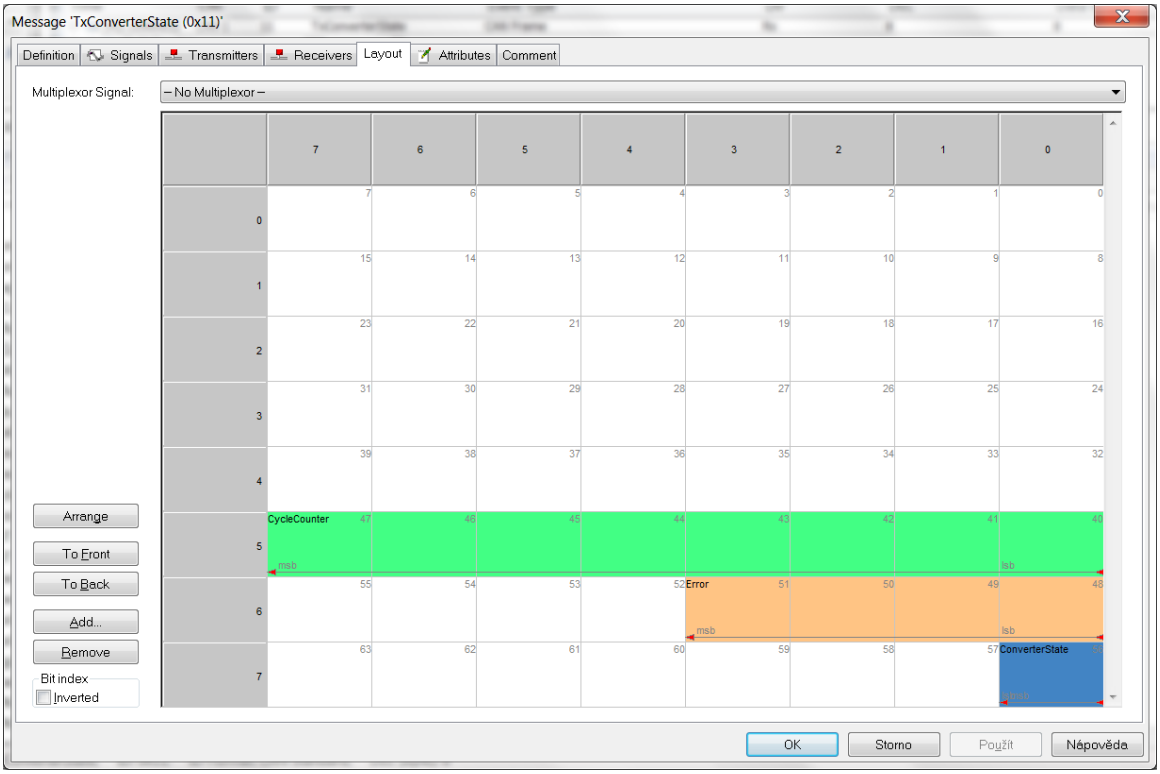
```
static uint8_t CycleCounter = 255;

/* transmitt CAN message */
         Isq_drive_i16= VC_Output.Math_Model.Isq*KP*PP*FPM*0.20833f*SCALE_Isq;
         nME_drive_i16 = (adc_InvMeasuredValuesFloat.wrm * RECAL_Wrm_to_nrm);

         Kpw_ui16=VC_Par.Isqw_Calcul.Contr_Wr.Kp*SCALE_Kpw;
         Tau_w_ui16=(VC_Par.Isqw_Calcul.Contr_Wr.Tau)*SCALE_Tau_w;
         KpT_ui16=VC_Par.Vect_Contr.Contr_Isd.Kp*SCALE_KpT;
         Tau_T_ui16=(VC_Par.Vect_Contr.Contr_Isd.Tau)*SCALE_Tau_T;

//Upload zprávy TxconverterState
       (((int8_t *)&CanTx_u64)[0]) = (foc_Started) ? 1u : 0u;
       (((int8_t *)&CanTx_u64)[1]) = Drive_error;
       (((int8_t *)&CanTx_u64)[2]) = CycleCounter;

         dev_CanTransmitData_e(DEV_CAN_IFACE_1, 1, CanTx_u64);
         CycleCounter--;    //Cycle counter, slouží k detekci výpadku komunikace na straně nadřazeného řízení
```

```
//Upload zprávy TxRegulatorConstant
        if(Send_out2)
        {
                ((uint16_t *)&CanTx_u64)[3] = Kpw_ui16;
                ((uint16_t *)&CanTx_u64)[2] = Tau_w_ui16;

                ((uint16_t *)&CanTx_u64)[1] = KpT_ui16;
                ((uint16_t *)&CanTx_u64)[0] = Tau_T_ui16;

              dev_CanTransmitData_e(DEV_CAN_IFACE_1, 3, CanTx_u64);
              CycleCounter--;

              Send_out2=0;
        }

//Upload zprávy TxMotorState
      ((int16_t *)&CanTx_u64)[0] = (int16_t)(VC_Output.Math_Model.Isq*KP*PP*FPM*0.20833f*SCALE_Torque);
      ((int16_t *)&CanTx_u64)[1] = nME_drive_i16 * SCALE_RPM;
      ((uint32_t *)&CanTx_u64)[1] = adc_InvMeasuredValuesFloat.theta_from_ARC;

      dev_CanTransmitData_e(DEV_CAN_IFACE_1, 5, CanTx_u64);
      CycleCounter--;
```

Úpravy v CANoe:



Obr. 4.3 Signály v CAN zprávě TxConverterState



Obr. 4.4 Uspořádání zprávy TxConverterState

4.5 Synchronizace pořadí parametrů ve zprávách Rx/TxRegulatorConstant

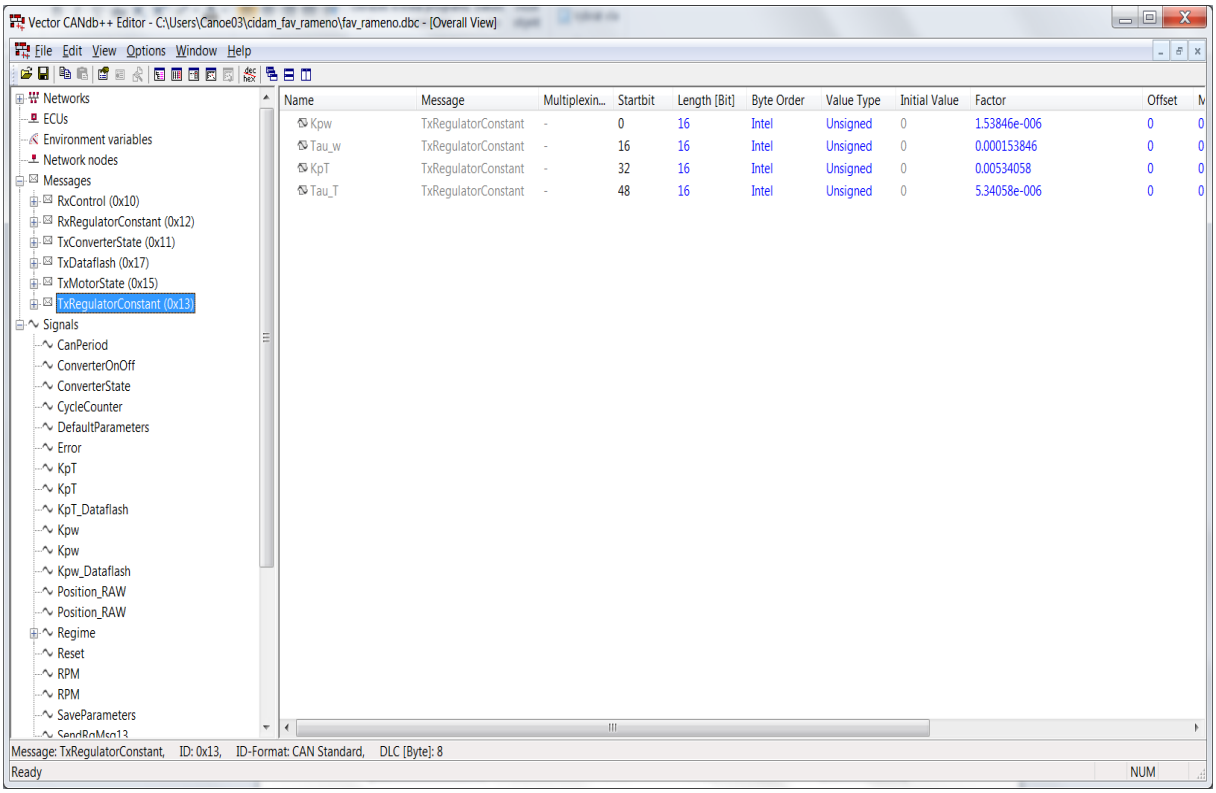
V této kapitole bude popsáno řešení zadání č. 7. V tomto bodě byl požadavek synchronizovat pořadí parametrů regulátorů ve zprávách RxRegulatorConstant a TxRegulatorConstant. Změny byly provedeny jak v kódu c, tak i v prostředí CANoe. V současném stavu je pořadí následuíjící: Kpw, Tau_w, KpT a Tou_T. První dva parametry jsou proporční zesílení a časová konstanta regulátoru rychlosti a zbylé dva jsou proporční zesílení a časová konstanta regulátorů proudů. Regulátory složek proudu i_{sd} a i_{sq} mají stejné parametry.

Úpravy v kódu:

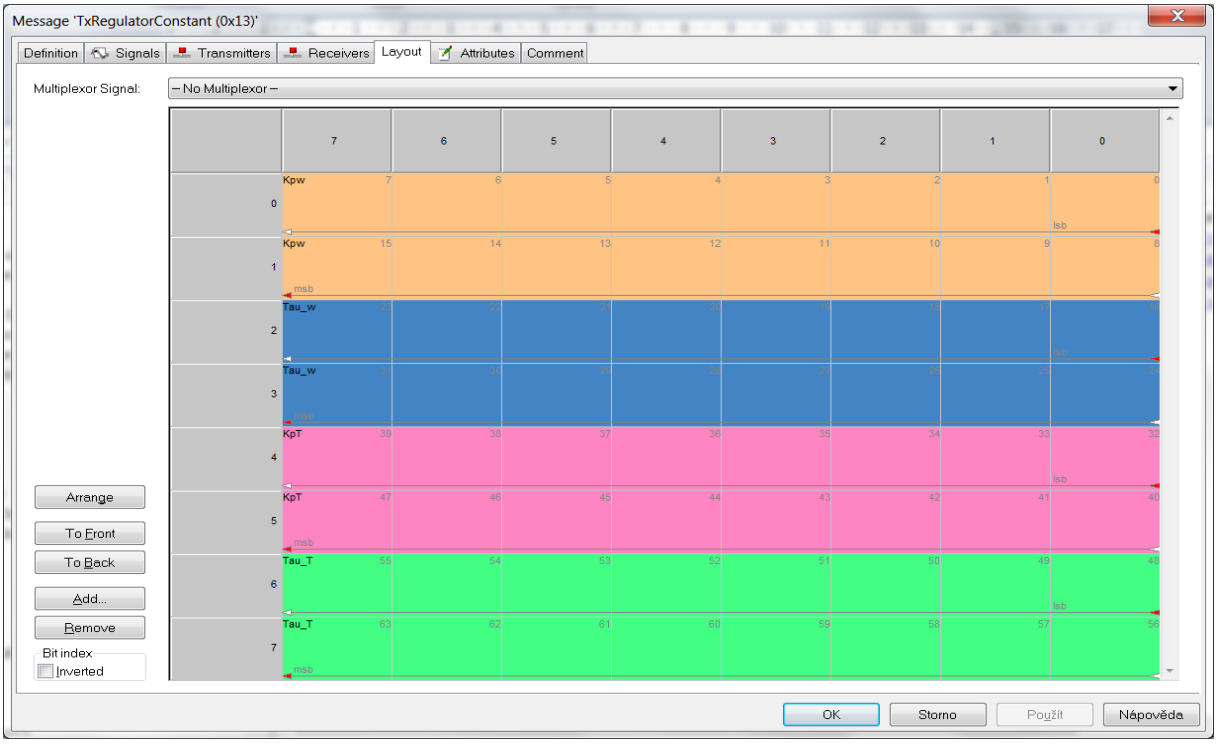
main.cpp, řádky: 803 - 816:

```
//Upload zprávy TxRegulatorConstant
if(Send_out2)
{
	((uint16_t *)&CanTx_u64)[3] = Kpw_ui16;
	((uint16_t *)&CanTx_u64)[2] = Tau_w_ui16;

	((uint16_t *)&CanTx_u64)[1] = KpT_ui16;
	((uint16_t *)&CanTx_u64)[0] = Tau_T_ui16;

	dev_CanTransmitData_e(DEV_CAN_IFACE_1, 3, CanTx_u64);
	CycleCounter--;

	Send_out2=0;
}
```

Úpravy v CANoe:



Obr. 4.5 Signály v CAN zprávě TxRegulatorConstant

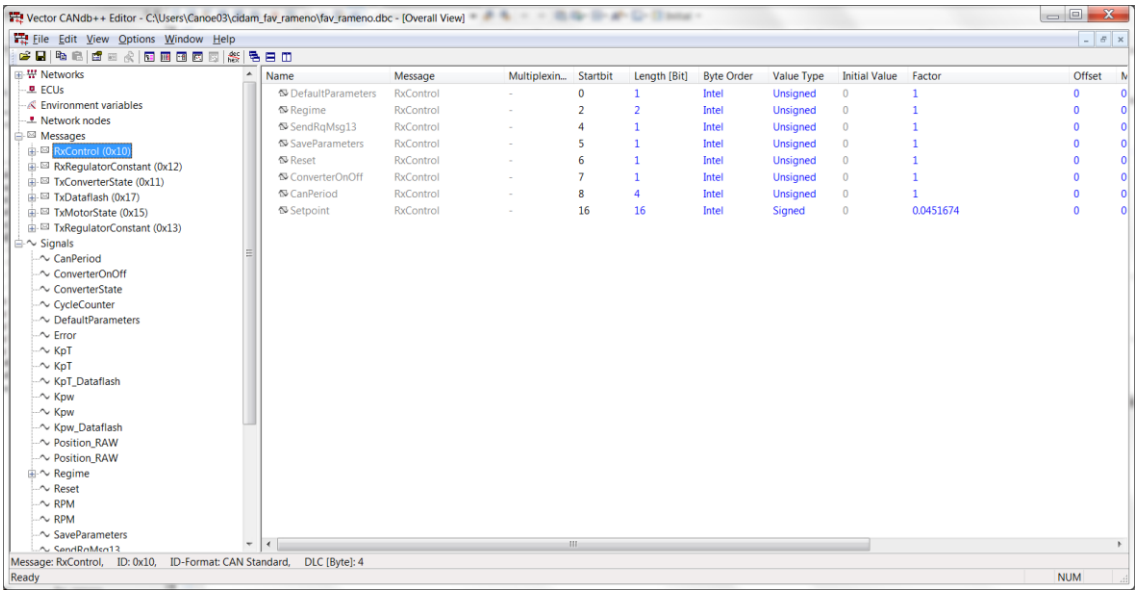


Obr. 4.6 Uspořádání zprávy TxRegulatorConstant

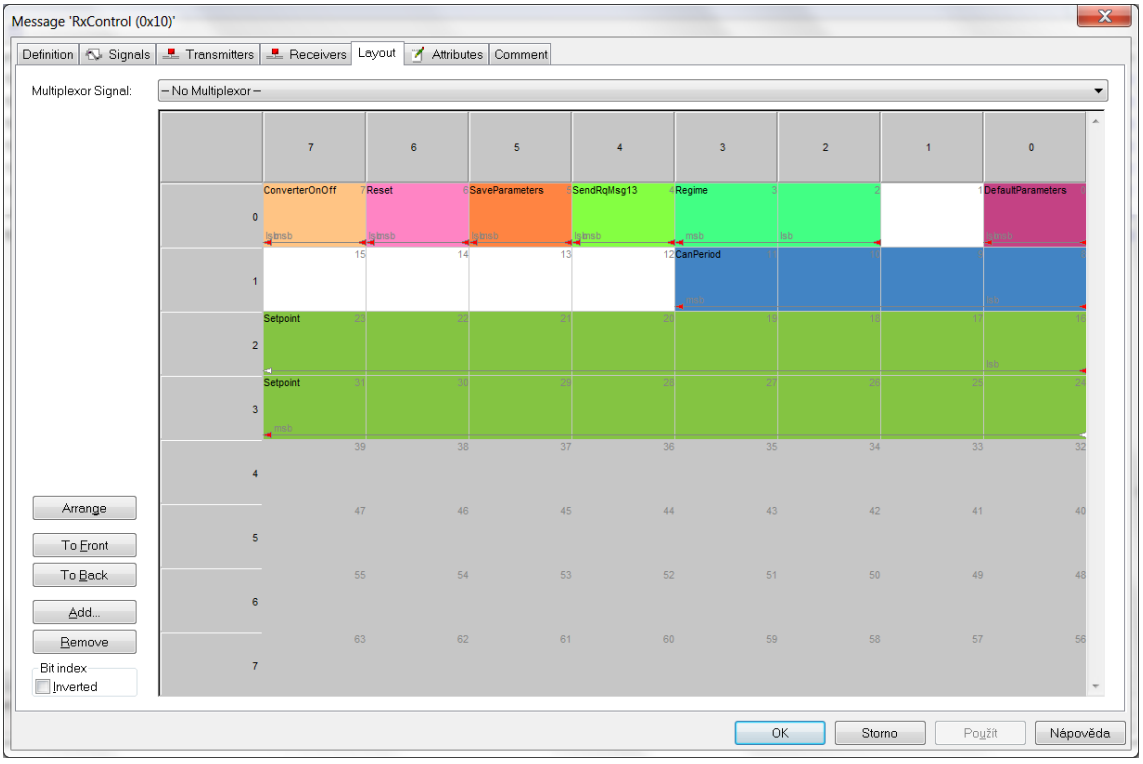
4.6 *Ukládání parametrů regulátorů do paměti*

V bodě zadání č. 8 byly požadavky na realizaci funkce ukládání parametrů do paměti a funkce obnovení defaultních hodnot parametrů regulátorů.

Pro tyto účely byla v komunikaci CAN upravena zpráva RxControl a přidána nová zpráva TxDataflash. Ve zprávě RxControl byly alokovány dva nové bity SaveParameters a DefaultParameters. Pomocí bitu SaveParameters je možné uložit nové hodnoty regulátorů do paměti dataflash. Pomocí bitu DefaultParameters je možné nastavit počáteční parametry regulátorů.



Obr. 4.7 Signály v CAN zprávě RxControl



Obr. 4.8 Uspořádání zprávy RxControl

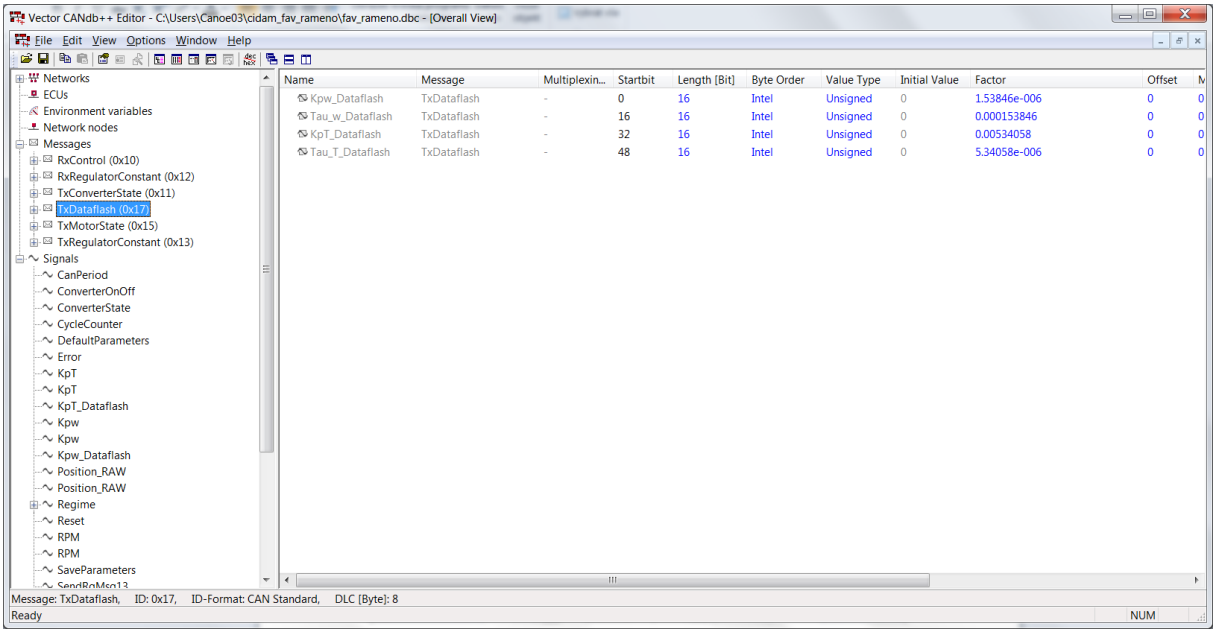
Aktivací bitů SaveParameters nebo DefaultParameters dochází v softwaru k nastavení příznaků FLAG_SaveParameters a FLAG_DefaultParameters. Tyto příznaky jsou datového typu uint8_t a nabývají hodnot 0, 1 nebo 2. Hodna 0 signalizuje, že daný příznak není aktivní (není požadavek na žádný zápis resp. načtení defaultních hodnot). Hodnota 1 signalizuje, že daný příznak je aktivní. Pokud má příznak hodnotu 1, dojde k provedení rutiny zápisu nebo načtení defaultních hodnot. Po dokončení rutiny se automaticky daný příznak nastaví na hodnotu 2. Tato hodnota signalizuje, že příslušná rutina proběhla a čeká se, až přijde nový požadavek. Z uživatelského hlediska to znamená, že pokud zůstane ve zprávě RxControl aktivovaný bit SaveParameters nebo DefaultParameters proběhne daná rutina v procesoru pouze jednou a dále se čeká, až uživatel příslušný bit vynuluje a nastaví znovu do log. 1. Tímto je zabráněno opakovatelnému zápisu stejných parametrů do paměti dataflash. Vyhodnocení příznaků FLAG_SaveParameters a FLAG_DefaultParameters z přijaté zprávy RxControl je znázorněno kódem níže.

Kód:

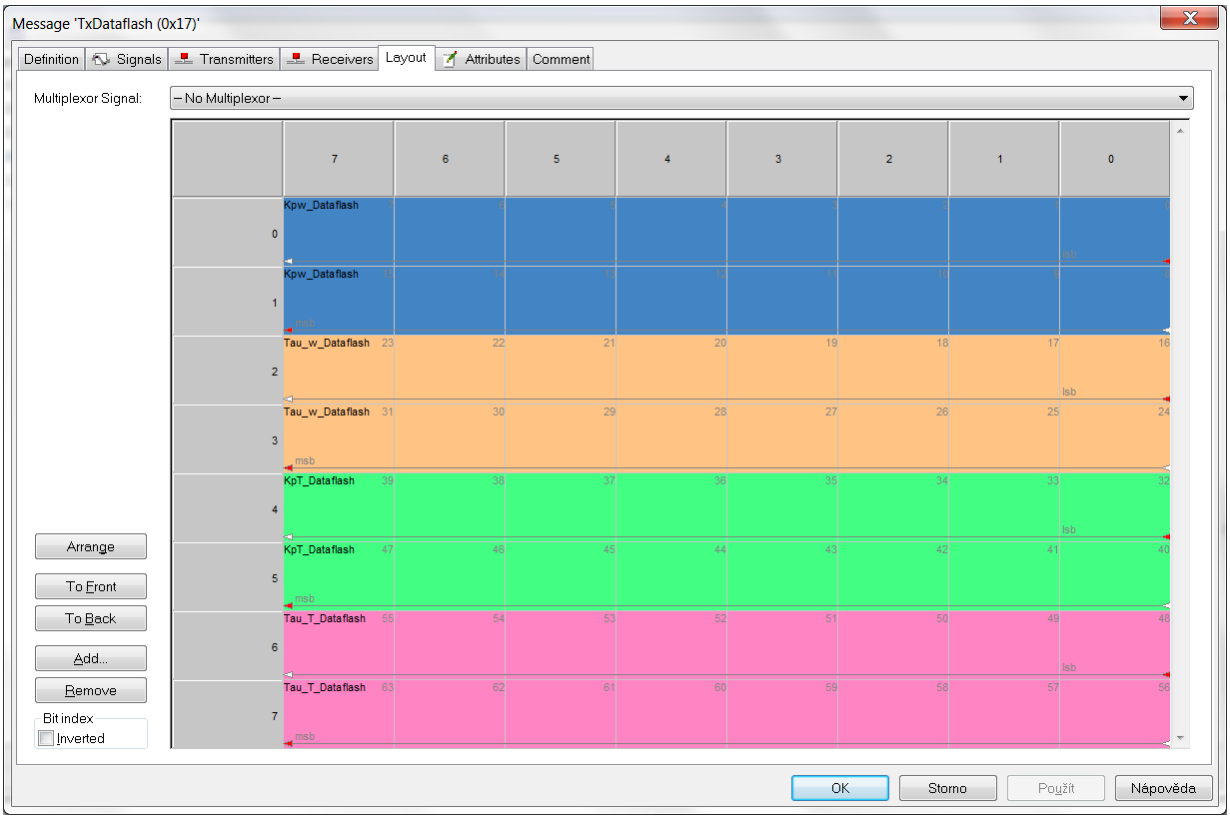
main.cpp (řádek 595):

```
if(CanId_u32==0x10)
             {
                 ProcessDataWatchdog = 0;         //Tím že přijde zpráva RxControl nuluj CounterProcessDataWatchdog

                 OnOff=(((int8_t *)&RxData_u64)[7])>>7;
                  reset=((((int8_t *)&RxData_u64)[7])>>6)&0x1;

                  if(FLAG_SaveParameters == 2)                    //Osetreni, aby nedochazelo k cyklickemu
prepisovani dataflash
                  {
                    FLAG_SaveParameters=((((int8_t *)&RxData_u64)[7])>>5)&0x1;
                    if(FLAG_SaveParameters == 0)
                    {
                        FLAG_SaveParameters = 0;
                    }
                    else
                    {
                      FLAG_SaveParameters = 2;
                    }
                  }
                  else
                  {
                    FLAG_SaveParameters=((((int8_t *)&RxData_u64)[7])>>5)&0x1;
                  }


                if(FLAG_DefaultParameters == 2)                    //Osetreni, aby nedochazelo k cyklickemu
resetovani parametru
                {
                  FLAG_DefaultParameters=(((int8_t *)&RxData_u64)[7])&0x1;

                  if(FLAG_DefaultParameters == 0)
                  {
                      FLAG_DefaultParameters = 0;
                  }
                  else
                  {
                      FLAG_DefaultParameters = 2;
                  }
                }
                else
                {
                    FLAG_DefaultParameters=(((int8_t *)&RxData_u64)[7])&0x1;
                 }
```

Zpráva TxDataflash obsahuje aktuálně uložené parametry regulátorů v paměti dataflash. Uspořádání zprávy TxDataflash je stejné jako uspořádání zprávy TxRegulatorConstant. Rozdíl mezi těmito zprávami je takový, že ve zprávě TxRegulatorConstant se posílají aktuálně používané parametry regulátorů, kdežto ve zprávě TxDataflash se posílají uložené parametry regulátorů, které nemusí být shodné s aktuálně používanými parametry.



Obr. 4.9 Signály v CAN zprávě TxDataflash



Obr. 4.10 Uspořádání zprávy TxDataflash

4.6.1 Upload zprávy TxDataflash

Upload zprávy TxDataflash se provádí v mainu následujícím kódem.

Kód:

main.cpp:

```
//Upload zprávy TxDataflash
        dev_SdfReadPage_e(0, Buffer_RW_dataflash, 0);       //cteni z dataflash

        ((uint16_t *)&CanTx_u64)[3] = ((uint16_t *)&Buffer_RW_dataflash)[0];        //Kpw_dataflash
        ((uint16_t *)&CanTx_u64)[2] = ((uint16_t *)&Buffer_RW_dataflash)[1];        //Tau_w_dataflash

        ((uint16_t *)&CanTx_u64)[1] = ((uint16_t *)&Buffer_RW_dataflash)[2];        //KpT_dataflash
        ((uint16_t *)&CanTx_u64)[0] = ((uint16_t *)&Buffer_RW_dataflash)[3];        //Tau_T_dataflash
        CycleCounter--;

        dev_CanTransmitData_e(DEV_CAN_IFACE_1, 7, CanTx_u64);
```

4.6.2 Inicializace paměti

V inicializační části softwaru dochází k inicializaci paměti dataflash. Tato část kódu nejprve provede inicializační rutinu paměti. Následně se testuje proměnná initOK. Jestliže proběhne inicializace bez problémů, bude se proměnná initOK rovnat nule. Pokud dojde k chybě při inicializaci dataflash, bude proměnná initOK nenulová. V tomto případě, z důvodu bezpečnosti, dojde k přiřazení defaultních hodnot parametrů regulátorů, se kterými se dále pracuje. V případě, že inicializace proběhne bez problému, nastává nejprve počáteční inicializace bufferu, jehož všechny byty se nastaví na nulovou hodnotu. Dále dojde k vyčtení paměti dataflash. Pokud je paměť prázdná, budou mít všechny byty bufferu hodnotu 0xFF. V tomto případě dojde k přiřazení defaultních hodnot parametrů regulátorů. Defaultní hodnoty parametrů regulátorů se následně uloží do paměti dataflash. Pokud při inicializaci dataflash není prázdná, dojde k jejímu přečtení. Pokud přečtené hodnoty jsou v daných mezích, jsou použity jako parametry regulátorů. Pokud přečtené hodnoty nejsou v daných mezích, dochází k smazání paměti dataflash a opětovnému nahrání defaultních hodnot regulátorů. V tomto případě jsou pro další běh programu a regulaci použity defaultní parametry regulátorů.

Kód:

main.cpp, incializace:

```
//Upper limits of regulator from dataflash
#define UPLIM_KP_RegWr 0.11f    //Horni limit proporcni slozky regulatoru W. Po cteni parametru regulatoru z
dataflash se provadi kontrola. Parametry musi byt v urcitych mezich.
#define UPLIM_TAU_RegWr 11.0f   //Horni limit integracni slozky regulatoru W. Po cteni parametru regulatoru z
dataflash se provadi kontrola. Parametry musi byt v urcitych mezich.
#define UPLIM_KP_RegIs 355.0f   //Horni limit proporcni slozky regulatoru Is. Po cteni parametru regulatoru z
dataflash se provadi kontrola. Parametry musi byt v urcitych mezich.
#define UPLIM_TAU_RegIs 0.4f    //Horni limit integracni slozky regulatoru Is. Po cteni parametru regulatoru z
dataflash se provadi kontrola. Parametry musi byt v urcitych mezich.

static uint8_t Buffer_RW_dataflash[512];       //Buffer pro cteni a zapis do dataflash
static int8_t FLAG_RewriteDataflash = 0;
uint16_t init_OK = 65535;      //pokud inicializace probehne OK, nastavi init_OK do 0

init_OK = dev_SdfDataFlashInit_e();  //inicializace dataflash

  if(init_OK != 0)
  {
      //chyba initu dataflash, vezmi pocatecni parametry regulatoru z define v ctr_VectorFloat.h
      RegWr_Kp_shadow = KP_WR;
      RegWr_Tau_shadow = TR_WR;
      RegIs_Kp_shadow = INV_KP_IS;
      RegIs_Tau_shadow = INV_TR_IS;
  }
  else
  {
      for(uint16_t i=0; i<=511; i++)            //Inicializace Bufferu pro cteni a zapis do dataflash
      {
```

```
            Buffer_RW_dataflash[i] = 0;
        }


//Otestovani, zda-li je dataflash prazdna nebo uz jsou v ni ulozeny parametry regulatoru
        dev_SdfReadPage_e(0, Buffer_RW_dataflash, 0);

        for(uint16_t i=0; i<=511; i++)
        {
            if(Buffer_RW_dataflash[i] == 255)
            {
                j++;
            }
        }

        if(j == 512)
        {
            //Dataflash je prazdna, pouzij pro puvodni parametry z define v ctr_VectorFloat.h a do dataflash zapis
puvodni parametry regulatoru ve FP
            ((uint16_t *)&Buffer_RW_dataflash)[0] = KP_WR_FP;
            ((uint16_t *)&Buffer_RW_dataflash)[1] = TR_WR_FP;
            ((uint16_t *)&Buffer_RW_dataflash)[2] = KP_IS_FP;
            ((uint16_t *)&Buffer_RW_dataflash)[3] = TR_IS_FP;

            dev_SdfWritePage_e(0, Buffer_RW_dataflash, 0);  //zapis novych hodnot do dataflash

            RegWr_Kp_shadow = KP_WR;
            RegWr_Tau_shadow = TR_WR;
            RegIs_Kp_shadow = INV_KP_IS;
            RegIs_Tau_shadow = INV_TR_IS;
        }
        else
        {
            //V dataflash jsou nejake hodnoty, otestuj a pouzij tyto hodnoty
            RegWr_Kp_shadow = (((uint16_t *)&Buffer_RW_dataflash)[0]) * SCALE_1_Kpw;
            RegWr_Tau_shadow = (((uint16_t *)&Buffer_RW_dataflash)[1]) * SCALE_1_Tau_w;
            RegIs_Kp_shadow = (((uint16_t *)&Buffer_RW_dataflash)[2]) * SCALE_1_KpT;
            RegIs_Tau_shadow = (((uint16_t *)&Buffer_RW_dataflash)[3]) * SCALE_1_Tau_T;

            //Overeni, pokud prijate parametry regulatoru nebudou v danych mezich, tak pro regulaci pouzij defaultni
parametry a prehraj dataflash
            if((RegWr_Kp_shadow < 0.0) || (RegWr_Kp_shadow >= UPLIM_KP_RegWr))
            {
                dev_SdfErasePage_e(0);            //smaz dataflash
                FLAG_RewriteDataflash = 1;

                RegWr_Kp_shadow = KP_WR;
                RegWr_Tau_shadow = TR_WR;
                RegIs_Kp_shadow = INV_KP_IS;
                RegIs_Tau_shadow = INV_TR_IS;
            }

            if((RegWr_Tau_shadow < 0.0) || (RegWr_Tau_shadow >= UPLIM_TAU_RegWr))
            {
                dev_SdfErasePage_e(0);            //smaz dataflash
                FLAG_RewriteDataflash = 1;

                RegWr_Kp_shadow = KP_WR;
                RegWr_Tau_shadow = TR_WR;
                RegIs_Kp_shadow = INV_KP_IS;
                RegIs_Tau_shadow = INV_TR_IS;
            }

            if((RegIs_Kp_shadow < 0.0) || (RegIs_Kp_shadow >= UPLIM_KP_RegIs))
            {
                dev_SdfErasePage_e(0);            //smaz dataflash
                FLAG_RewriteDataflash = 1;

                RegWr_Kp_shadow = KP_WR;
                RegWr_Tau_shadow = TR_WR;
                RegIs_Kp_shadow = INV_KP_IS;
                RegIs_Tau_shadow = INV_TR_IS;
            }

            if((RegIs_Tau_shadow < 0.0) || (RegIs_Tau_shadow >= UPLIM_TAU_RegIs))
            {
                dev_SdfErasePage_e(0);            //smaz dataflash
                FLAG_RewriteDataflash = 1;

                RegWr_Kp_shadow = KP_WR;
                RegWr_Tau_shadow = TR_WR;
                RegIs_Kp_shadow = INV_KP_IS;
                RegIs_Tau_shadow = INV_TR_IS;
            }
```

```
        if(FLAG_RewriteDataflash == 1)              //pokud vyctene hodnoty z dataflash jsou chybne,
        {                                           //prepis dataflash
            ((uint16_t *)&Buffer_RW_dataflash)[0] = KP_WR_FP;
            ((uint16_t *)&Buffer_RW_dataflash)[1] = TR_WR_FP;
            ((uint16_t *)&Buffer_RW_dataflash)[2] = KP_IS_FP;
            ((uint16_t *)&Buffer_RW_dataflash)[3] = TR_IS_FP;

            dev_SdfWritePage_e(0, Buffer_RW_dataflash, 0);  //zapis novych hodnot do dataflash
            FLAG_RewriteDataflash = 0;
        }
    }
}
```

4.6.3 Zapsání nových parametrů regulátorů

Zápis nových parametrů regulátorů se provádí aktivací bitu SaveParameters do log. 1 ve zprávě RxControl. Pokud je tento bit aktivován, dojde v procesoru k uložení aktuálních hodnot regulátorů, které jsou posílány ve zprávě RxRegulatorConstant, do paměti dataflash. Tato funkce je blokována funkcí *Reset a načtení defaultních hodnot* parametrů regulátorů. Vždy může být aktivována pouze jedna z těchto funkcí.

Zápis nových parametrů regulátorů se provádí na první stránku paměti dataflash, která má index 0. Samotný algoritmus zápisu je následující. Nejprve musí být smazána celá stránka, na kterou chceme zapsat. Dále se ověřuje, zda-li je stránka smazána. Pokud je stránka smazána, budou mít všechny byty bufferu hodnotu 0xFF. V tomto případě dojde k uložení nových hodnot parametrů regulátorů.

Kód:

main.cpp:

```
    if((FLAG_SaveParameters == 1)&&(FLAG_DefaultParameters == 0))
    {
        uint16_t j = 0;

       dev_SdfErasePage_e(0);                           //mazani dataflash
       dev_SdfReadPage_e(0, Buffer_RW_dataflash, 0);    //cteni smazane dataflash

       for(uint16_t i=0; i<=511; i++)          //overeni jestli je dataflash smazana
       {
                if(Buffer_RW_dataflash[i] == 255)
               {
                        j++;
               }
       }

       if(j == 512)        //pokud je dataflash smazana, proved zapis novych hodnot do dataflash
       {
                ((uint16_t *)&Buffer_RW_dataflash)[0] = RegWr_Kp;
               ((uint16_t *)&Buffer_RW_dataflash)[1] = RegWr_Tau;
               ((uint16_t *)&Buffer_RW_dataflash)[2] = RegIs_Kp;
               ((uint16_t *)&Buffer_RW_dataflash)[3] = RegIs_Tau;

               dev_SdfWritePage_e(0, Buffer_RW_dataflash, 0);  //zapis novych hodnot do dataflash

               FLAG_SaveParameters = 2;
       }
    }
```

4.6.4 Reset a načtení defaultních hodnot

Reset a načtení počátečních hodnot parametrů regulátorů se provádí aktivací bitu DafaultParameters ve zprávě RxControl. Pokud je tento bit aktivovaný, nastaví softwarový příznak FLAG_DefaultParameters do log. 1. Tímto dojde ke smazání paměti dataflash a následnému uložení továrních hodnot parametrů regulátorů. Vyvolání této funkce lze pouze tehdy, je-li řízení pohonu vypnuté (bit ON/OFF ve zprávě RxControl má hodnotu nula) a současně není aktivovaný bit pro uložení nových parametrů regulátorů do dataflash. Defaultní hodnoty regulátorů jsou uloženy i do proměnných vektorového řízení. Tudíž při další aktivaci měniče probíhá řízení pohonu s továrními parametry regulátorů.

Kód:

main.cpp:

```
//Resetovani parametru regulatoru do defaultnich hodnot
  if((FLAG_DefaultParameters == 1)&&(OnOff == 0)&&(FLAG_SaveParameters == 0))   //do teto podminky se dostane v c
  pripade:  menic je vypnuty + byl zadan prikaz na reset do defaltnich hodnot par. regulatoru + neni aktivovan
  pozadavek ulozeni parametru regulatoru
  {
      uint16_t j = 0;

      dev_SdfErasePage_e(0);            //smaz dataflash

      RegWr_Kp_shadow = KP_WR;          //Nastav parametry regulatoru z definu (prvotni nastaveni)
      RegWr_Tau_shadow = TR_WR;
      RegIs_Kp_shadow = INV_KP_IS;
      RegIs_Tau_shadow = INV_TR_IS;

      VC_Par.Vect_Contr.Contr_Isd.Kp=RegIs_Kp_shadow;
      VC_Par.Vect_Contr.Contr_Isd.Tau=RegIs_Tau_shadow;

      VC_Par.Vect_Contr.Contr_Isq.Kp=RegIs_Kp_shadow;
      VC_Par.Vect_Contr.Contr_Isq.Tau=RegIs_Tau_shadow;

      VC_Par.Isqw_Calcul.Contr_Wr.Kp=RegWr_Kp_shadow;
      VC_Par.Isqw_Calcul.Contr_Wr.Tau=RegWr_Tau_shadow;

      dev_SdfReadPage_e(0, Buffer_RW_dataflash, 0);       //cteni smazane dataflash

      for(uint16_t i=0; i<=511; i++)          //overeni jestli je dataflash smazana
      {
          if(Buffer_RW_dataflash[i] == 255)
          {
              j++;
          }
      }

      if(j == 512)        //pokud je dataflash smazana, proved zapis novych hodnot do dataflash
      {
          ((uint16_t *)&Buffer_RW_dataflash)[0] = KP_WR_FP;
          ((uint16_t *)&Buffer_RW_dataflash)[1] = TR_WR_FP;
          ((uint16_t *)&Buffer_RW_dataflash)[2] = KP_IS_FP;
          ((uint16_t *)&Buffer_RW_dataflash)[3] = TR_IS_FP;

          dev_SdfWritePage_e(0, Buffer_RW_dataflash, 0);  //zapis novych hodnot do dataflash

          FLAG_DefaultParameters = 2;
      }
  }
```

4.7 Realizace funkce Process data watchdog

V bodě zadání č. 9 byl požadavek na realizaci funkce Process data watchdog. Tato funkce slouží k detekci výpadku komunikace.

Proces data watchdog byl realizován pomocí softwarového čítače, který je odvozen od interruptu regulační smyčky vektorového řízení. Regulační smyčka má frekvenci 10 000Hz, čemuž odpovídá perioda 100us. Při každém přerušení dojde k inkrementaci proměnné ProcessDataWatchdog. Následně v hlavní smyčce programu je při každém přijetí CAN zprávy RxControl tato proměnná ProcessDatatWatchdog nulována. Pokud dojde k načítání v proměnné ProcessDataWatchdog do hodnoty 499, znamená to, že 50ms nebyla přijata zpráva RxControl, tudíž se předpokládá, že vypadla komunikace. V případě vyhodnocení této chyby dochází k nastavení chybového stavu a vypnutí měniče. V softwaru byl rozšířen chybový výčtový typ drive_error_t o další chybu CAN_ERROR, která je označena číslem 10. Process data watchdog hlídá výpadek komunikace jen v případě, že je měnič aktivovaný a probíhá řízení motoru.

Úpravy v kódu:

Ctr_VectorFloat.h:

```
#define THRESHOLD_PROCESS_DATA_WATCHDOG 499     //threshold process data watchdogu,                    (499+1
* perioda_interruptu) = čas kdy watchodg zareaguje (v tomto případě je to 50ms)
```

isr.cpp:

```
extern uint16_t ProcessDataWatchdog;


   if(ProcessDataWatchdog >= THRESHOLD_PROCESS_DATA_WATCHDOG)
   {
         Drive_error = CAN_ERROR;
   }

// Regulace
   if(Drive_error == NO_ERROR)
   {
       if(OnOff)
       {
             ProcessDataWatchdog++;
                           .
                           .
                           .
       }
   else
   {
        fpg_BpcWriteSwBlck(fpg_BpcGetSwBlck_u32() | 0x00fc);

        fpg_LedOff(FPG_LED_RUN);
        foc_Started=0;
        CAN_req=0;
        ProcessDataWatchdog = 0;
   }
```

main.cpp: (řádky 595 - 615)

```
  //************Process data watchdog************************************
  uint16_t ProcessDataWatchdog;

   if(CanId_u32==0x10)
    {
         ProcessDataWatchdog = 0;     //Tím že přijde zpráva RxControl nuluj CounterProcessDataWatchdog

         OnOff=(((int8_t *)&RxData_u64)[7])>>7;
         reset=((((int8_t *)&RxData_u64)[7])>>6)&0x1;
         regime_shadow = (Regime_t)(((((int8_t *)&RxData_u64)[7])>>2)&0x3);
         Send_out2=((((int8_t *)&RxData_u64)[7])>>4)&0x1;
         CAN_period=((((int8_t *)&RxData_u64)[6]))&0xf;
                           .
                           .
                           .
```

```
   }
```

main.cpp: (řádky 519 - 540)

```
switch(state)
  {
  case STATE_RESET:
    {
        ProcessDataWatchdog = 0;          //nuluj CounterProcessDataWatchdog komunikace CAN

        //fpg_BpcWriteSwBlck(0xfefe);//pwm 0 and 8 are not blocked, inverter blocked, synch on and
charging done
        fpg_LedOff(FPG_LED_RUN);
        fpg_LedOff(FPG_LED_ERROR);
        bsp_DoSystemReset();

    state = STATE_IDLE;
    break;
    }
```

5 Závěr

V této zprávě byly podrobně rozepsány úpravy řídicího softwaru pohonu robotického ramene. Robotické rameno je poháněno synchronním motorem s permanentními magnety. Motor je napájen z frekvenčního měniče. Celý pohon je řízen jednou řídicí kartou ze systému REMCS. Řídicí systém využívá pro regulaci PMSM vektorové řízení v kartézských souřadnicích.

Úpravy softwaru byly v této zprávě rozděleny do dvou skupin. První skupina úprav byla zaměřena na detekci a odstranění problémů, které se týkaly kmitání otáček a náhodných proudových špiček při běhu pohonu.

Druhá skupina úprav softwaru se týkala požadavků zákazníka. Jednalo se o úpravy stávajícího kódu, doplnění funkcí CycleCounter a Process data watchdog a úprava komunikace CAN.

Literatura

[1] https://vector.com/portal/medien/cmc/manuals/VN1600_Interface_Family_Manual_EN.pdf

Seznam obrázků

Historie revizí

Rev.	Kapitola	Popis změny	Datum	Jméno
0	Všechny	Publikování dokumentu	29.6.2018	Z. Kehl