Current Loop Control of a brushless DC motor with hall sensors using the ADMC401

AN401-44

Table of Contents

Summary

The inherent advantages of brushless DC motors (BLDC) such as high torque/volume ratio, small heat losses in the rotor or absence of brushes led to increased interest in applications for them. The delay of the success that brushless motors have now was mainly due to the computational burden they put on the controller. Modern processors can easily handle the algorithms that are required. Also, previously time intensive tasks as for instance the PWM are typically eased by partially implementing them in hardware. The ADMC401 combines the power of a DSP with dedicated hardware blocks for common motor control tasks.

This application note demonstrates the use of PWM block for closed loop current control of a brushless DC motor. The organisation of this application note is as follows. In section one, the principle of the torque generation for a DC motor with hall sensors will be discussed. In section two, the application routines for open loop control of the BLDC will be presented. In section three, an example of closed loop current control using a PI controller will be shown. Section four will demonstrate the experimental results.

1 Principle of torque generation with a trapezoidal BLDC

1.1 System overview

The example software is written for controlling a trapezoidal BLDC with three hall-effect sensors. The following figure gives an overview of the complete system. It is only a functional diagram, showing the connections to the ADMC401 mounted on the connector board. Therefore all the power supplies are omitted.

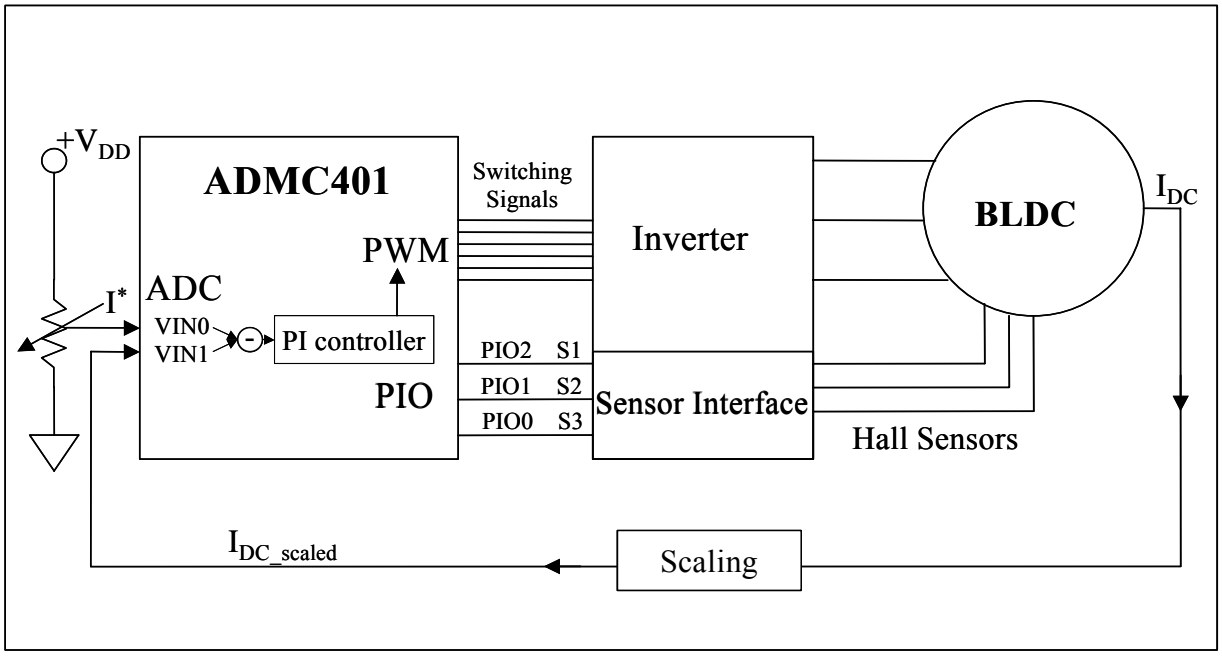


Figure 1 Overall System

The potentiometer that is connected to an analog input VIN0 (through the Analog to Digital Converter, ADC) serves as the current command. Voltage input ranges from –2V to +2V can be applied through the

ADMC connector board. Thus input of +/-2V means commanding maximum current, which corresponds to +1/-1 in digital value. Positive command will generate positive torque while negative command will generate negative torque. The current command is then compared with the measured dc current of the BLDC (this measured current is connected to VIN1 in the ADC), and the calculated error serves as an input for the current controller (I- controller), which makes use of the PI control application note[1]. Finally, the output from the I- controller determines the duty-cycle command of the PWM generation.

1.2 The trapezoidal BLDC and the hall effect sensors

The first step when developing a PWM control for a brushless motor is to determine the commutation sequence of the motor. In general, BLDC may use either 60° or 120° commutation. Figure 2 below shows a 120° commutation scheme between the hall sensor output and the back emf of the BLDC used in this example. When a different commutation scheme is used, the code needs to be modified accordingly.

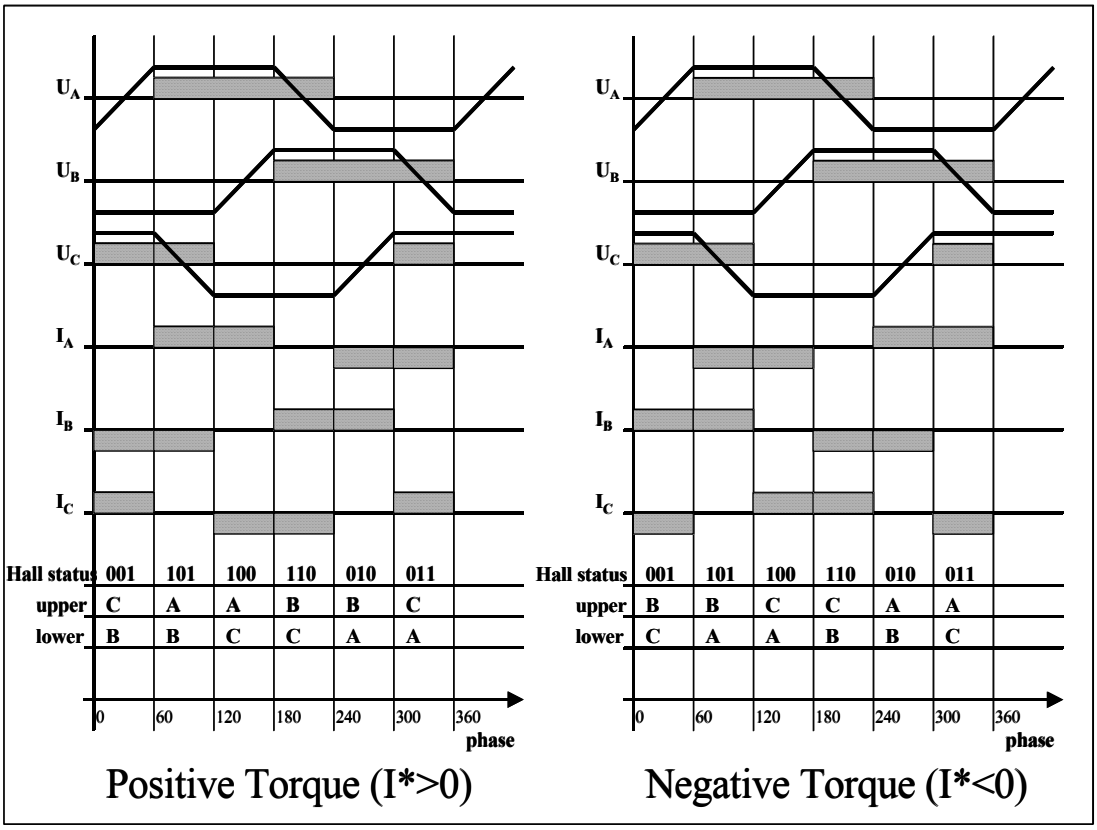


Figure 2 Trapezoidal BLDC operation

The three sensor signals, which serve as inputs into the PIO2, PIO1 and PIO0, are defined to form a binary code, called *Hall status* in the figure, which uniquely identifies the 60° sector the rotor is in. It is noted that the combinations 000 and 111 do not occur. As a result, by reading the value in the PIODATA register, one can identify the particular sector in which the rotor is and perform the control of the BLDC.

[1] Refer to PI Control application note for details.

On receiving the current command I*, the I-controller will generate the appropriate duty-cycle command. Then in each sector, two of the phases are typed with flat back-emf to the DC rails and the remaining 3rd phase is left open. Its sign (or direction) is controlled by the selection of the DC rail a leg is switched to. For example, if the I*>0, and the last three digits of the PIODATA is 101, then, according to Figure 2, phase A will be switched to the positive DC rail while phase B will be switched to the negative DC rail. Phase C will be left open. As a result, only the upper switch of phase A and lower switch of phase B are active, while the rest are inactive. The following figure illustrates the corresponding inverter configuration. The implementation of switching from one phase to another will be discussed in the following sub-section.

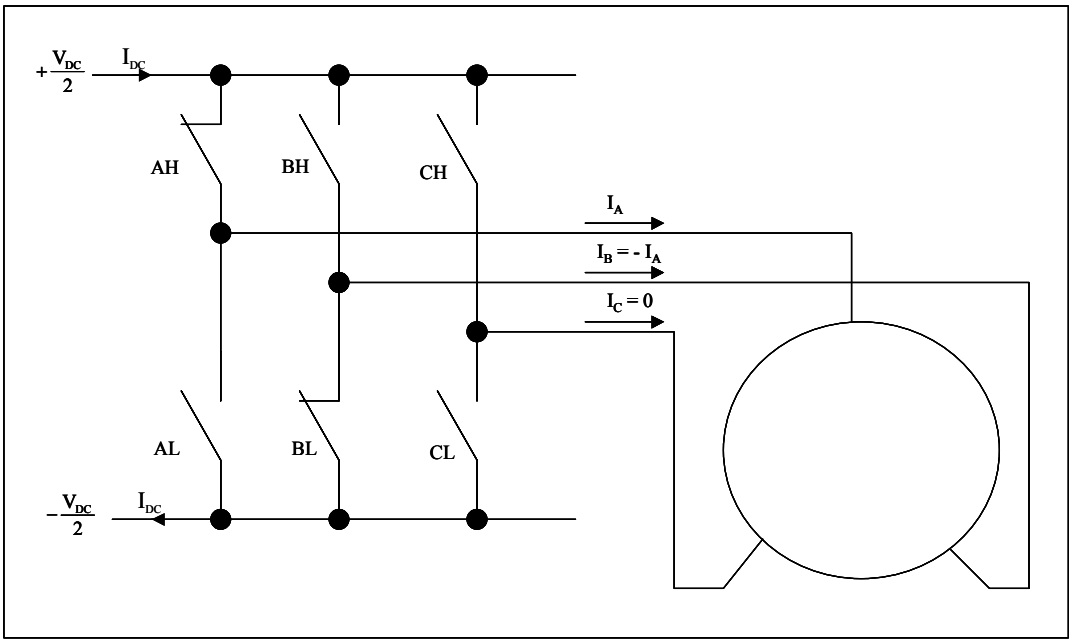


Figure 3 Example Inverter Configuration

1.3 Implementation of switching using the PWM block

As discussed in the previous section, since in each sector four switches are inactive and only two are active, the switching implementation makes extensive use of the PWMSEG register for enabling and disabling the appropriate switches. Besides, the PWM crossover feature is used in this application. It allows normal (from the software's point of view) programming of the duty-cycle while the value is latched to the lower switch instead of the upper one.

In each PWM cycle the current is turned on and off. To achieve this, it is sufficient to switch one of the two involved power devices. Therefore, in this application, the lower active switch is permanently turned on (100% duty-cycle) and the desired duty-cycle is imposed by the upper device. This clearly reduces the number of commutations and therefore the switching losses. The following figure illustrates the PWM signals (active high), the current and the PWMSEG register for some PWM cycles in the sector represented in Figure 3.

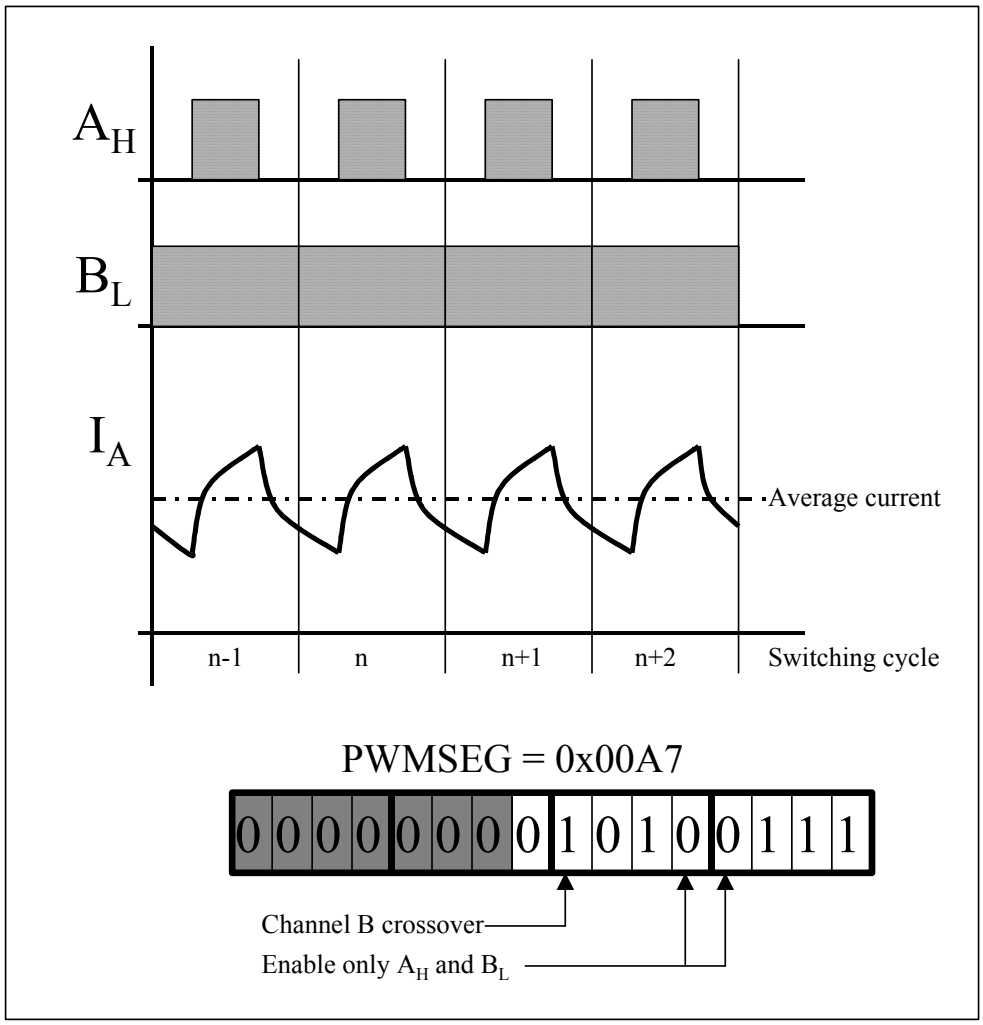


Figure 4 PWM waveforms (active high) and PWMSEG configuration register

2 Implementation of the BLDC Application Routines

2.1 Usage of the BLDC routines

The application is based on two files, the "bldc.dsp" and the "bldc.h". The file "bldc.dsp" contains the assembly code of the subroutines. The user has to include the header file "bldc.h", which provides the function-like calls to the subroutines. The following table summarises the set of macros that are defined in the BLDC application. The example file in Section 3will demonstrate the usage of all the routines.

Operation	*Usage*	*Explanation*
Initialisation	**BLDC_INIT**	**Initialise the PIOs for hall sensors**
BLDC	**BLDC_START**	**Determine the rotor sector and performs open loop control of the BLDC**
PWM configuration	**Set_BLDC_PWM(PWMSEG_config, upper_switch, lower_switch)**	**Configures the PWM and set the desired duty cycle command**

Table 1 Implemented routine

There are three macros in the BLDC application. Since PIOs are used for reading the hall sensor signals, the BLDC_INIT configures PIO0, PIO1 and PIO2 as inputs and disable the PIO interrupts. The control of the BLDC is mainly implemented in the BLDC_START routine, which will be discussed in section 2.4. The macro Set_BLDC_PWM is used internally by the BLDC_START routine, for configuring the PWMSEG register and setting the duty cycle of the active switches, as discussed in Section 1.3.

The routines do not require any configuration constants from the main include-file "main.h" that comes with every application note. Section 3 shows an example of usage of this application. In the following text each routine is explained in detail with the relevant segments of code, which are found in either "bldc.h" or "bldc.dsp". For more information see the comments in those files.

2.2 Usage of DSP registers

The macro listed in Table 1 is based on the subroutines listed in Table 2. Each subroutine will be discussed in the following sections. The following table gives an overview of what DSP registers are used in this macro:

Subroutine	*Inputs*	*Outputs*	*Modified registers*
BLDC_INIT_	No	No	ay0,ar
BLDC_START_	ar	No	ax0, ay0, ar, my0, mr, I7
BLDC_error_	No	No	No

Table 2 Input and output format, modified registers in the BLDC routines

2.3 Access to the BLDC routines: the header file

The BLDC application can be accessed by including the header file "bldc.h" in the application code.

The header file gives external access to the bldc routines. It is mostly self-explaining. It defines the calls shown in Table 1. The BLDC_INIT and BLDC_START are called by users to initialise the PIOs and implement the BLDC control. Please note that Set_BLDC_PWM is called internally by BLDC_START for passing parameters only, in order to implement the switching as described in section 1.3. For example, using the case in section 1.3, the corresponding macro call will be Set_BLDC_PWM(UA_LB, PWMCHA, PWMCHB). The PWMSEG register is then configured by the value defined in UA_LB(0x00A7), for enabling only AH and BL in the PWM outputs and disabling AL, BH, CH and CL. Besides, the outputs for BH and BL are cross-over. Also, the desired duty cycle command is imposed on AH and 100% duty cycle is imposed on BL.

```
{*************************************************************************************
*                                                                                    *
* Type: Macro                                                                        *
* Call: BLDC_INIT_;                                                                  *
* Implementing BLDC initialization                                                   *
* Inputs  :          None                                                            *
* Outputs  :         None                                                            *
* Modified:          ay0, ar                                                         *
*                                                                                    *
{*************************************************************************************}

.MACRO BLDC_INIT;
          Call BLDC_INIT_;
.ENDMACRO;

{*************************************************************************************
*                                                                                    *
* Type: Macro                                                                        *
* Call: BLDC_START_;                                                                 *
* Implementing basic BLDC control                                                    *
* Inputs  :          ar      (duty cycle command input)                              *
* Outputs  :         None                                                            *
* Modified:          ax0,ay0,ar,my0,mr,I7                                            *
*                                                                                    *
{*************************************************************************************}

.MACRO BLDC_START;
          Call BLDC_START_;
.ENDMACRO;

{*************************************************************************************
*                                                                                    *
* Type: Macro                                                                        *
*Setting  PWM configurations and perform switching                                   *
* Inputs  :          %0 :  appropriate PWMSEG configuration                          *
*                    %1 :  active upper switch                                       *
*                    %2 :  :  active lower switch                                    *
* Outputs  :         None                                                            *
* Modified:          ax0                                                             *
*                                                                                    *
{*************************************************************************************}

.MACRO set_BLDC_PWM(%0,%1,%2);     { updates the segment and crossover settings}
      ax0 = %0;                                  { and the duty-cycle commands of the PWM block     }
      dm(PWMSEG) = ax0;
      ax0 = mr1;
      dm(%1) = ax0;
      ax0 = dm(PWMTM);
      dm(%2) = ax0;
.ENDMACRO;
```

2.4 The program code

The code contained in the file "bldc.dsp" defines the routines listed in Table 2.

Some constants are defined to configure the PWMSEG register, in order to enable and disable the appropriate switches. Please refer to Section 1.3 for details. In the example shown in Section 1.3, the upper and lower active switches are at phase A and B respectively, hence the corresponding constant is denoted as UA_LB. The notation of other constants follows the same manner. Another constant defined is for initialising the PIOs, such that PIO0, PIO1 and PIO2 are set as inputs for reading the hall sensor values.

```
{*****************************************************************************
* Constants Defined in the Module                                            *
*****************************************************************************}
{PWM_SEG configurations for each of the segments}
.CONST UB_LA = 0x011b;
.CONST UC_LA = 0x011e;
.CONST UC_LB = 0x00b6;
.CONST UA_LB = 0x00a7;
.CONST UA_LC = 0x006d;
.CONST UB_LC = 0x0079;


.CONST  PIO_hall_sense      = 0x0FF8;          {configure PIO pins:    PIO0 Hall sensor S3 as input
                                                                       PIO1 Hall sensor S2 as input
                                                                       PIO2 Hall sensor S1 as input }
```

The following code initialises the PIOs as inputs for the hall sensors, and disable the PIO interrupts.

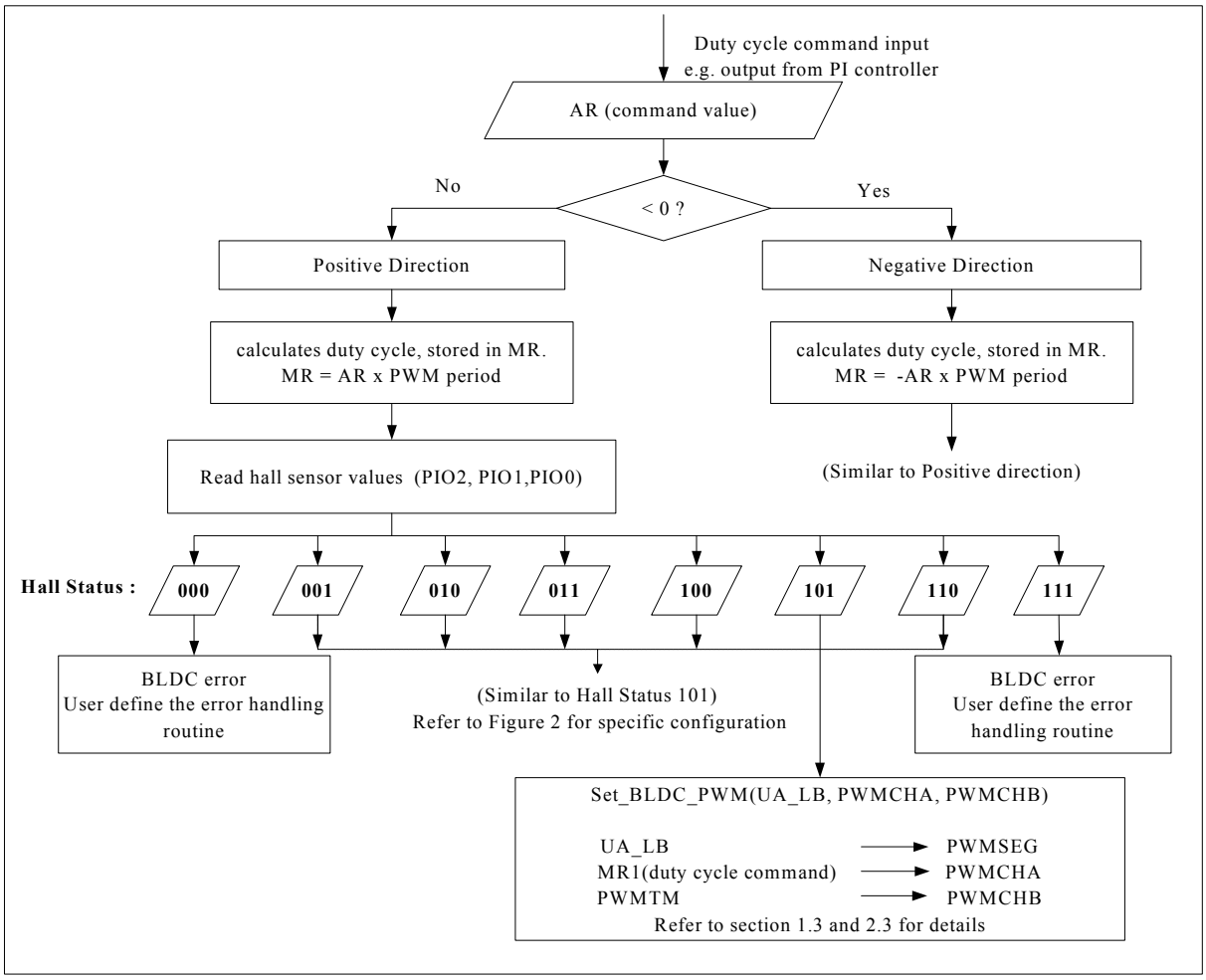
```
BLDC_INIT_:

        AY0=PIO_hall_sense;
        AR=DM(PIODIR);
        AR=AR and AY0;
        dm(PIODIR)=AR;

        AY0 = 0x063F;       {Disable interrupt for PIO0-PIO2 in PICMASK}
        AR = DM(PICMASK);
        AR = AR and AY0;
        dm(PICMASK) = AR;
rts;
```

The main control (for just open loop) of the BLDC is implemented in the following code, please note that there is one input into this routine, which is the duty cycle command value and this value should be stored in ar register before entering this routine. In this example, since this command value comes from the output of the I-controller, this controller output value is stored in ar before entering the BLDC_START routine, as will be shown in the main.dsp program in Section 3.3. The idea of the BLDC_START routine can be summarized in FlowChart 1.

FlowChart 1 Details of the BLDC_START_ routine




```
BLDC_START_:

if LT jump NEGATIVE_DIR;

POSITIVE_DIR:                   { Generate positive torque          }

     my0= dm(PWMTM);
     mr= ar * my0 (SU);         { trasform command value in duty-cycle  }

     ax0 = dm(PIODATA);                 { read status of hall-sensors           }
     ay0 = 0x0007;
     ar  = ax0 and ay0;
     ay0 = ^jump_table_posdir;  { load ay0 with base address of jump-table }

     ar = ar + ay0;                     { interpret hall status as offset       }
     i7 = ar;
     jump(i7);                  { jump to appropriate location in table  }

jump_table_posdir:                      { call the corresponding PWM setup macro  }
     jump BLDC_ERROR_;
```

```
        jump posdir_001;
        jump posdir_010;
        jump posdir_011;
        jump posdir_100;
        jump posdir_101;
        jump posdir_110;
        jump BLDC_ERROR_;

{ bit code in rev_xxx labels indicate status of phase CBA respectively                                  }
posdir_001: set_BLDC_PWM(UC_LB,PWMCHC,PWMCHB);                  { set PWM and return to main loop
}
                rts;
posdir_010: set_BLDC_PWM(UB_LA,PWMCHB,PWMCHA);
                rts;
posdir_011: set_BLDC_PWM(UC_LA,PWMCHC,PWMCHA);
                rts;
posdir_100: set_BLDC_PWM(UA_LC,PWMCHA,PWMCHC);
                rts;
posdir_101: set_BLDC_PWM(UA_LB,PWMCHA,PWMCHB);
                rts;
posdir_110: set_BLDC_PWM(UB_LC,PWMCHB,PWMCHC);
                rts;


NEGATIVE_DIR:                                           { Generate negative torque              }
            mr=0;
            my0=dm(PWMTM);          { load with   -full cycle                   }

            mr= mr-ar * my0 (SS);                       { trasform command value in duty-cycle    }

        ax0 = dm(PIODATA);                              { read status of hall-sensors                   }
        ay0 = 0x0007;
        ar  = ax0 and ay0;
        ay0 = ^jump_table_negdir;               { load ay0 with base address of jump-table }

        ar = ar + ay0;                                  { interpret hall status as offset           }
        i7 = ar;
        jump(i7);                                       { jump to appropriate location in table    }

jump_table_negdir:                                      { call the corresponding PWM setup macro   }
        jump BLDC_ERROR_;
        jump negdir_001;
        jump negdir_010;
        jump negdir_011;
        jump negdir_100;
        jump negdir_101;
        jump negdir_110;
        jump BLDC_ERROR_;

{ bit code in negdir_xxx labels indicate status of phase CBA respectively
}
negdir_001: set_BLDC_PWM(UB_LC,PWMCHB,PWMCHC);                  { set PWM and return to main loop
}
                rts;
negdir_010: set_BLDC_PWM(UA_LB,PWMCHA,PWMCHB);
                rts;
negdir_011: set_BLDC_PWM(UA_LC,PWMCHA,PWMCHC);
```

```
            rts;
negdir_100: set_BLDC_PWM(UC_LA,PWMCHC,PWMCHA);
            rts;
negdir_101: set_BLDC_PWM(UB_LA,PWMCHB,PWMCHA);
            rts;
negdir_110: set_BLDC_PWM(UC_LB,PWMCHC,PWMCHB);
            rts;


BLDC_ERROR_:
nop; {This is the error handling routine, e.g.when the hall status is invalid (e.g.000, 111)}
      {Here, the user can define what is to be done in case of error}
rts;
```

3 Software Example: Current Loop Control of BLDC using a PI controller

3.1 Closed loop control of BLDC

DC machines are widely used in closed loop motion control systems because of their excellent control properties and fast response. In this example, current loop control of a BLDC is demonstrated using a PI controller together with the BLDC routines.

3.2 Current measurement and I-controller settings

As shown in Figure 1, the current flowing through the BLDC, I_{DC} is measured and it serves as the feedback signal for the I-controller. I_{DC_scaled} is the value of amplifying the voltage drop across a shunt resistor at the DC rails. Figure 5 shows I_{DC_scaled}, and actual I_{DC} value measured with a current probe, with respect to the PWMSYNC signal (for single update mode). As shown in Figure 5, current flows through the BLDC between each PWMSYNC, due to the center based PWM generation. As a result, if the PWM single update mode is used and the I_{DC_scaled} value is acquired at each PWMSYNC, the acquired current value will always be zero. Thus, in this example, double update mode is used, such that I_{DC_scaled} can be acquired at the second half of the PWMSYNC. Figure 6 shows I_{DC_scaled} and the acquired I_{DC_scaled} value ($I_{DC\_acquired}$) with respect to the PWMSYNC, when double update mode is used. Note that $I_{DC\_acquired}$ is acquired at the second half of each PWMSYNC, but read at the next half of the PWMSYNC (which is the first half), as will be shown in the main.dsp.

For the BLDC used here, the nominal current is 2.5A, so the maximum feedback current is set to be 3A. Since the maximum input voltage into the ADC through the ADMC connector board is 2V, the I_{DC_scaled} value is scaled such that when the ADC reads a digital value of +1, that corresponds to I_{DC} having a value of 3A. It is possible to change the scaling factors for both the feedback current and the command current, in this example, they have the same scaling factor.

A 32-bit, zero-order hold approach PI controller is used as the current controller. This controller is tuned empirically, with proportional gain K_p=0.75 and bandwidth ω_{pi} =160Hz. Sampling frequency is at 20kHz. The calculated parameters are n=0, B_0=1, A_1^{sc}=0x6000 (16-bit, 1.15 format) and A_0^{sc}=0xAC44 (16-bit, 1.15 format). Since A_0^{sc} and A_1^{sc} are stored in program memory (24-bit), they are declared as A_1^{sc} =0x600000 and A_0^{sc} =0xAC4400. These constants are defined in the main.dsp program as shown in Section 3.3.

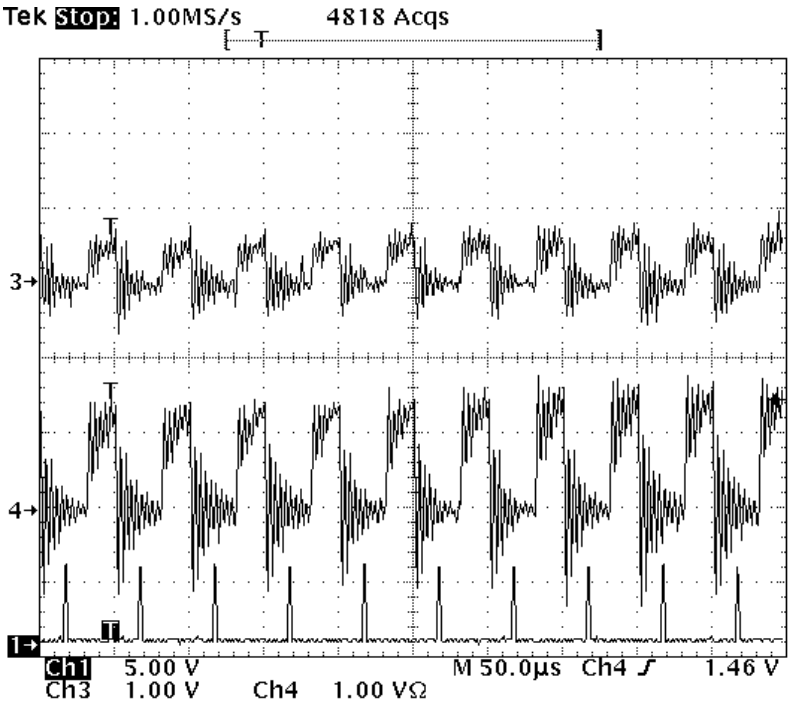


Figure 5 Measured current with respect to PWMSYNC (single update mode)

Ch1: PWMSYNC, Ch3: I_{DC_scaled} (1.5A/div) Ch4: I_{DC} from current probe(1A/div)

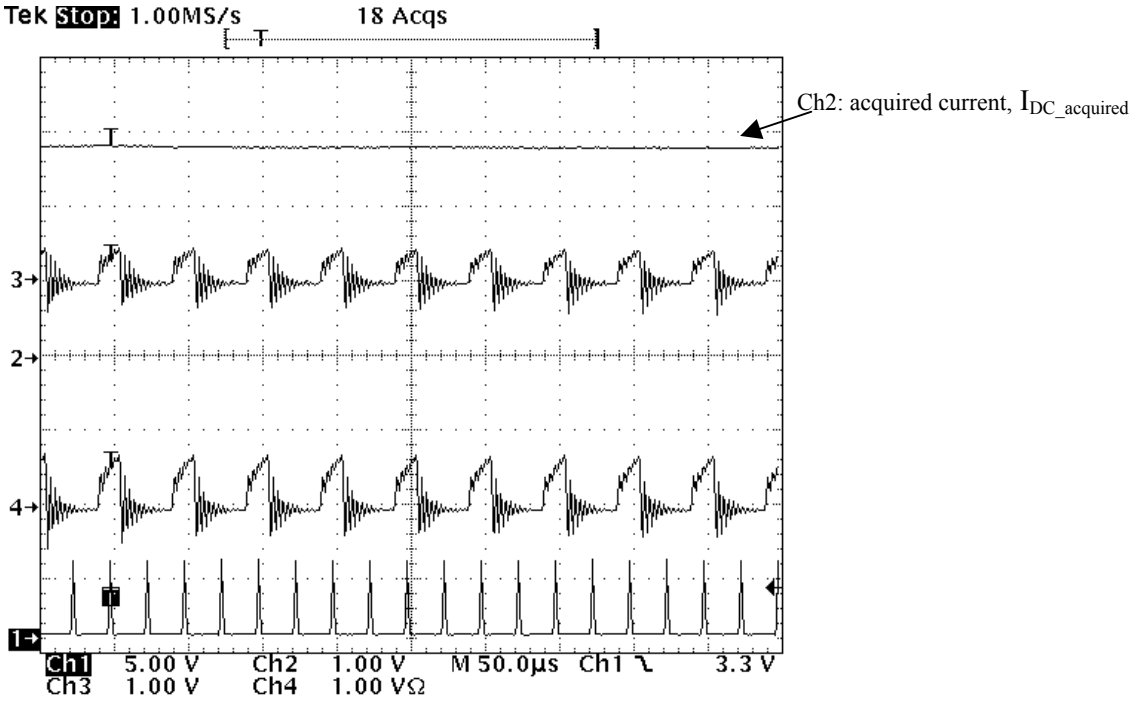


Figure 6 Measured current with respect to PWMSYNC (double update mode)

Ch1: PWMSYNC, Ch2: $I_{DC\_acquired}$[2], Ch3: I_{DC_scaled} (1.5A/div) Ch4: I_{DC} from current probe (1A/div)

[2] $I_{DC\_acquired}$ is the output from the DAC, the output voltage ranges from 0V-5V representing –1 to +1 digital value. Please refer to the DAC application note for details.

3.3 The main program: main.dsp

The file "main.dsp" contains the initialisation and PWM Sync and Trip interrupt service routines. The batch file **build.bat** may be applied either within DOS prompt or by double-clicking on it from Windows Explorer, it compiles every file of the application, links them together and builds the executable file **main.exe**. This file can be run on the Motion Control Debugger. A brief description of the code is given as follows:

Start of code - declaring start location in program memory

```
.MODULE/RAM/SEG=USERPM1/ABS=0x60     Main_Program;
```

*Next, the general systems constants and PWM configuration constants (main.h – see the next section) are included. Also included are the PWM library, the ADC and DAC interface library, the pi control library and the **BLDC routines**.*

```
{*****************************************************************************
* Include General System Parameters and Libraries                            *
*****************************************************************************}
#include <main.h>;
#include <pwm401.h>;
#include <bldc.h>;
#include <adc401.h>;
#include <dac401.h>;
#include <pi.h>;
```

Some variables are defined hereafter. Here is where the I-controller is initialised.

```
{*****************************************************************************
* Local Variables Defined in this Module  ( .Var )                           *
*****************************************************************************}
{ ---------------- for 32 bit Current controller ---------------------- }
#define PI_SF32 0                                   { n}

.VAR/RAM/PM/CIRC/SEG=USER_PM1 PI_Coef32[2];
.INIT PI_Coef32: 0xA4CC00, 0x600000;                {kp=.75, wpi=160hz}

.VAR/RAM/DM/CIRC/SEG=USER_DM1 PI_Delay32[3];     { Ik, Uk_M, Uk_L}
.INIT PI_Delay32: 0x0000, 0x0000, 0x0000;
```

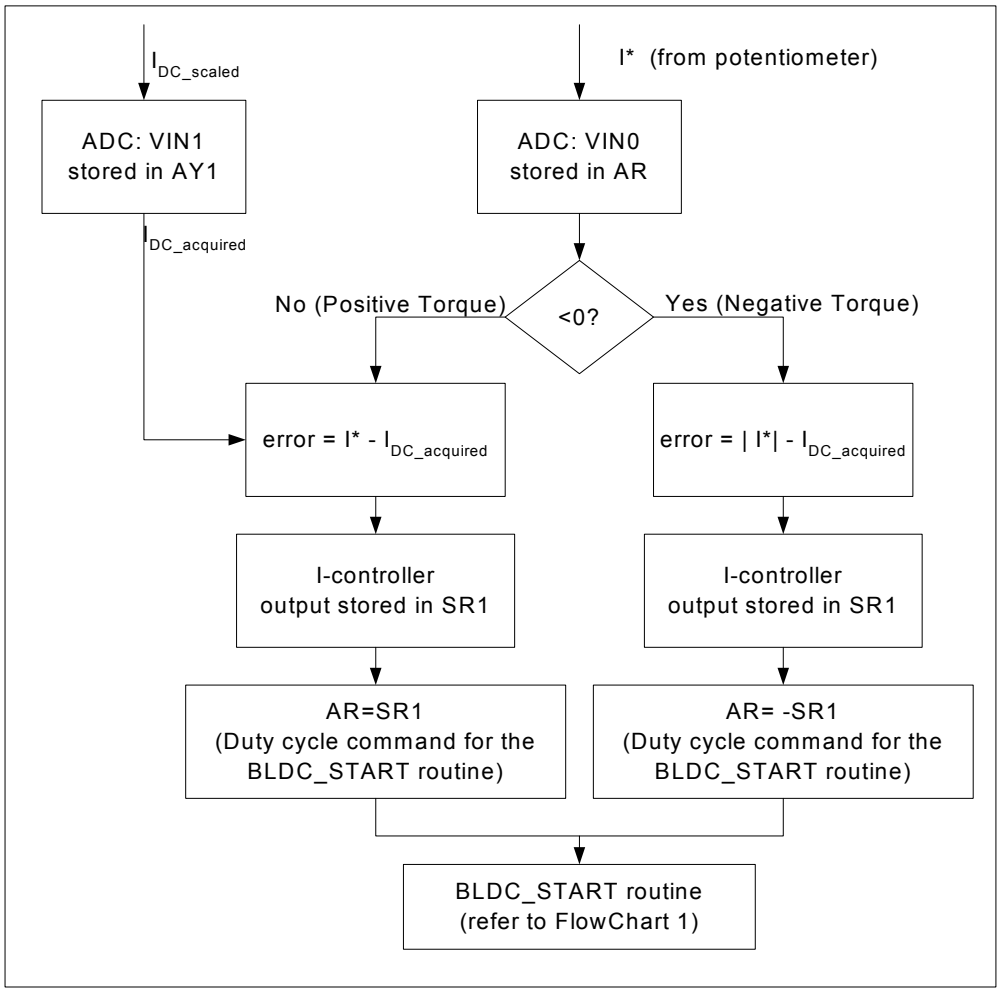
The initialisation of the PWM block is executed. Note how the interrupt vectors for the PWMSync and PWMTrip service routines are passed as arguments. Double update mode is set. Then the interrupt IRQ2 is enabled by setting the corresponding bit in the IMASK register. The ADC, DAC blocks are initialised. The PIOs are initialised as inputs using the BLDC_INIT and the I-controller is initialised. The main loop just waits for interrupts.

```
{*****************************************************************************}
{ Start of program code                                                       }
{*****************************************************************************}
Startup:
            PWM_Init(PWMSYNC_ISR  , PWMTRIP_ISR);
            Set_Bit_DM(MODECTRL, 6);       { Set into Double Update Mode          }
           DAC_Init;
           IFC = 0x80;              { Clear any pending IRQ2 inter.}
           ay0 = 0x200;     { unmask irq2 interrupts.}
           ar = IMASK;
           ar = ar or ay0;
```

```
                IMASK = ar;          { IRQ2 ints fully enabled here}
                ADC_Init;
                BLDC_Init;
               INIT_PI32(PI_Delay32, 0x0000);                    { reset Current PI    }
Main:
jump Main;          { Wait for interrupt to occur}
rts;
```

The I-controller implemented in the PWMSYNC_ISR can be summarized in FlowChart 2. The feedback current from the BLDC and the current command values are read through the ADC. Note that the correct value of the I_{dc_scaled} is acquired at the second half of the PWMSYNC_ISR and is ready to be used in the next half (that means the first half) of the PWMSYNC_ISR. As discussed before, positive command means positive torque generation while negative command means negative torque generation, so the sign of the command determines only the direction of torque generation. As a result, when calculating the error for the I-controller, the sign of the current command is omitted and the absolute value of the current command is used. The output from the I-controller then serves as the duty cycle command for the BLDC routine, and the sign of the command is restored at this point for torque generation.

FlowChart 2 I-controller implemented in the PWMSYNC_ISR routine



The following is a piece of code to demonstrate the current loop control of a BLDC, using the PI routines and the BLDC routines. As shown below, it is executed within the PWM interrupt service routine, so that Tsample is determined by the PWM frequency (20kHz in this example). It is worth pointing out how the error signal is computed. Assuming both the reference value and the feedback signal being in 1.15 format, the difference may assume values comprised between –2 and 2. Clearly, those values can no longer be represented in 1.15 format. However, by enabling the ALU saturation mode, the difference is saturated to values in the range of –1 to 1. This should normally have only minimal effects on the overall behaviour, since errors that are larger than full scale should appear only in small transition times. If nonetheless the saturation of the error is unacceptable, it is necessary to divide it by two, call the PI routine and multiply the output by two.

```
{*************************************************************************************************}
{ PWM Interrupt Service Routine                                                                   }
{*************************************************************************************************}
PWMSYNC_ISR:
                Test_Bit_DM(SYSSTAT, 3); {Check if it is in the second half of the PWMCYNC}
                if_Set_Jump(Second_Halve);


                ADC_Read(ADC1, Offset_0to3); DAC_Put(2, ar);   {read the current feedback, max current limited to
                3Amp}
                ay1=ar;  {Passing the feedback value into ay1 for error calculation}


                ADC_Read(ADC0, Offset_0to3);DAC_Put(1, ar); {Read command value} {command is stored in ar}
                ar=pass ar; {passing current command into I-controller}


                {This part is for PI for the current loop, feedback current at VIN1}
                if LT jump Negative_Torque_; {Applying negative values mean negative torque generation, measured
                current always >=0}


                        ena AR_SAT;
                        ar=ar-ay1;DAC_Put(3, ar);    {error signal}
                        dis AR_SAT;
                        Pi32(PI_Delay32, PI_Coef32, PI_SF32);            { compute controller output   }
                        ar=pass sr1; DAC_Put(4, sr1);{since controller output is sr1, but into to the BLDC_Start is AR}
                        jump BLDC_;


Negative_Torque_:
                        ar=abs ar;
                        if AV ar=pass 0x7fff;
                        ena AR_SAT;
                        ar=ar-ay1;DAC_Put(3, ar);    {error signal}
                        dis AR_SAT;
                        Pi32(PI_Delay32, PI_Coef32, PI_SF32);            { compute controller output   }
```

```
                    ar=0x0-sr1; DAC_Put(4, sr1);{since controller output is sr1, but into to the BLDC_Start is AR}
BLDC_:
                BLDC_Start; {Start the BLDC routine, details refer to FlowChart1}
                RTI;

Second_Halve:
                DAC_Update;
                rti;
```

3.4 The main include file: main.h

This file contains the definitions of ADMC401 constants, general-purpose macros and the configuration parameters of the system and library routines. It should be included in every application. For more information refer to the Library Documentation File.

This file is mostly self-explaining. As already mentioned, the BLDC application routines do not require any configuration parameters. The following defines the parameters for the PWM ISR used in this example.

```
{*********************************************************************************************}
{ Library: PWM block                                                                          }
{ file   : PWM401.dsp                                                                 }
{ Application Note: Usage of the ADMC401 Pulse Width Modulation Block                         }
.CONST  PWM_freq        = 20000;                        {Desired PWM switching frequency [Hz]       }
.CONST  PWM_deadtime    = 1000;         {Desired deadtime [nsec] }
.CONST  PWM_minpulse    = 1000;           {Desired minimal pulse time [nsec]    }
.CONST  PWM_syncpulse   = 1540;           {Desired sync pulse time [nsec]        }
{*********************************************************************************************}
```

4 Experimental Results

The following results show the step response of the I-controller when load is absent. Figure 7 show the traces of the command current I*, the acquired current $I_{DC\_cquired}$, the error and the output from the current controller. From Figure 7, one can clearly see that it takes about 2.5ms for the current to reach its command value. In the same figure, the difference between I* and $I_{DC\_acquired}$ at the first 1ms is due to the fact that I* is negative, while $I_{DC\_acquired}$ is positive. If one looks closely, I* and $I_{DC\_acquired}$ do have the same current values, the difference is the sign. Note that these curves are the outputs from the DAC. Hence, a value of 2.5V means zero in digital value.

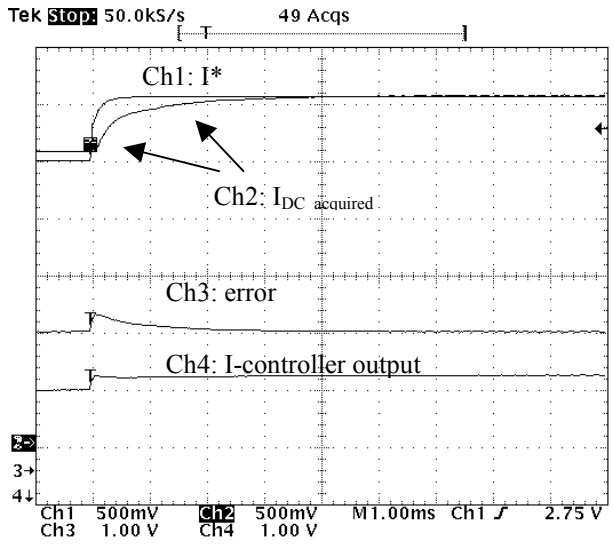


Figure 7 Step response of the I-controller (no load is applied)

Ch1: current command I*, Ch2: $I_{DC\_acquired}$, Ch3: error signal I*-I_{DC}_acquired, Ch4: I-controller output

The following figure shows the response of the I-controller in the case of disturbance (achieved by holding the motor shaft manually). As shown in the figure, the I-controller reacts to the disturbance such that the current flows into the BLDC $I_{DC\_acquired}$ still follows the command current I*.

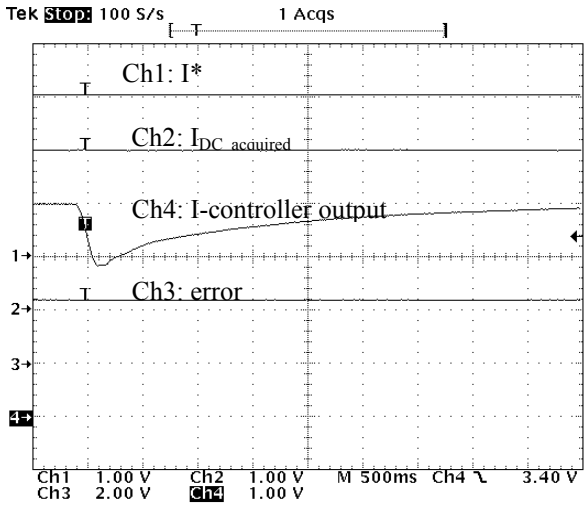


Figure 8 Response of the I-controller in case of disturbance

Ch1: current command I*, Ch2: I_{DC}_acquired, Ch3: error signal I*-I_{DC}_acquired, Ch4: I-controller output