Motor Control Platform Software Guide

DRAFT NOT IN DMS

Draft Date: 2011/12/20

SYNOPSIS

The XMOS motor control development platform is provided with a software framework and example control loop. This document provides information relating to the structure, implementation and use of the software modules that are specific to the motor control development platform, and interfacing to associated peripheral modules such as the CAN component.

CONTENTS

1 Motor Control Platform Example Applications

IN THIS CHAPTER

The current release package ships with two example applications.

- An application showing an example Field Oriented Control (FOC) control loop for two motors
- An application showing speed control of two motors using simple sector based commutation

Each application is capable of spinning two motors at speeds between 500 and 3800 RPM. The *A* and *B* buttons on the control board increase and decrease the speed. The display shows the demand speed, and the current speed of each of the two motors.

If an over-current state is detected by the system, or if the motor spins in an unexpected direction, or stalls, then the motor will be shut down, and *FAULT* will appear by the motor in the display.

1.1 Basic BLDC Speed Control Application `app_basic_bldc`

This application makes use of the following functionality.

- PWM
- Hall Input
- Display
- Ethernet & Communications
- Processing Blocks

1.1.1 Motor Control Loop

The main motor control code for this application can be found in the file src/motor/run_motor.xc. The motor control thread is launched using the following function.

```
void run_motor ( chanend c_wd,
      chanend c_pwm,
      chanend c_control,
      port in p_hall,
      port out p_pwm_lo[],
      chanend? c_wd );
```

The core of this function is a continuous loop that receives the position of the rotor as measured by the hall sensor, and selects which coil to energise based on that position.

After initially pausing and starting the watchdog the main loop is entered. The main loop responds to two events. The first event is a change in hall sensor state. This will trigger an update to the low side of the inverters (p_pwm_lo) and also to the PWM side of the inverter based on the hall sensor state. The output states are defined by the lookup arrays declared at the start of the function.

```
/* sequence of low side of bridge */
unsigned bldc_ph_a_lo[6] = {1,1,0,0,0,0};
unsigned bldc_ph_b_lo[6] = {0,0,1,1,0,0};
unsigned bldc_ph_c_lo[6] = {0,0,0,0,1,1};

/* sequence of high side of bridge */
const unsigned bldc_high_seq[6] = {1,2,2,0,0,1};
```

The other event that can be responded to is a command from the c_control channel. This can take the form of two commands. The first command is a request to read the current speed value. The second command is a request to change the PWM value that is being sent to the PWM thread and subsequently the motor.

1.1.2 Speed Control Loop

The speed control loop for this application can be found in the file src/control/speed_control.xc. The thread is launched by calling the following function.

```
void speed_control(chanend c_control, chanend c_lcd, chanend
  ↪ c_can_eth_shared );
```

This thread begins by initialising the PID data structure with the required coefficients. Following this a startup sequence is entered. This triggers open loop control to get the motor to begin rotating. After a sufficient time period the main speed loop is entered into.

The main loop consists of a select statement that responds to three events. The first event is a timed event that triggers the PID control and an update to the motor control threads PWM value. This simply applies the calculated PID error to the set point that is requested.

The second and third events are a request from the LCD and buttons thread or the communication I/O thread. This can either be a request from the display for updated speed, set point and PWM demand values or a change in set point.

1.2 FOC Application `app_dsc_demo`

This application makes use of the following functionality.

- PWM
- QEI
- ADC
- Display
- Ethernet & Communications
- Processing Blocks

1.2.1 Control Loop

The control loop can be found in the file `src/motor/inner_loop.xc`. The thread is launched by calling the following function.

```
void run_motor (
  chanend? c_in,
  chanend? c_out,
  chanend c_pwm,
  streaming chanend c_qei,
  chanend c_adc,
  chanend c_speed,
  chanend? c_wd,
  port in p_hall,
  chanend c_can_eth_shared)
```

The control loop takes input from the encoder, a set speed from the control modules and applies it via PWM. It contains two controllers in one loop, the speed controller and the current controller. The speed controller uses the QEI input to measure the speed of the motor, in order to bring the motor to the correct demand speed. The output of this controller is a tangential torque which is required to acheive that demand speed. The torque is passed through the `iq_set_point` variable. The `id_set_point` variable is always zero, as no force is required in the radial direction. The torque is a direct consequence of current flow in the coils, and therefore the `iq_set_point` is also a measure of the demand current.

The second controller is the torque/current controller. This uses the measured coil currents from the ADC, and tries to make them equal to the `iq_set_point` demand. The output of this controller is the extra current required to deliver the required torque. This is used to set the PWM duty cycles for the three coils.

Because the motor is spinning, and the mathematics for the algorithm is done in the frame of reference of the spinning rotor, the QEI is used to find the rotor angle. A Park transform is used to transform between the fixed coil frame of reference and the spinning rotor frame of reference.

The Clarke transform is used to convert the three currents in the coils into a radial and tangential two component current. This is possible because the coil currents have only two degrees of freedom, the third coil current being the sum of the other two.

This loop is a simple example of how a control loop may be implemented and the function calls that would be used to achieve this.

The first two arguments, `c_in` and `c_out` are used to synchronize the PWMs for multiple motors so that they do not have their ADC dead time in exactly the same time.

Further information on field oriented motor control can be found at:

- http://en.wikipedia.org/wiki/Field-Oriented_Control

1.2.2 Control loop customization

As described, there are two distinct control loops in the FOC design, but they are both coded into a single loop. Separating these into two loops, running in two different threads, may be necessary for designs that have a complex algorithm governing the speed.

The speed control part of the loop uses measurements from the QEI to determine the speed, and a set point that is passed in on a channel from the display or comms threads. To extract the speed control algorithm and put it into another thread, the following actions could be taken.

- Move the speed control PID calculation into a new thread (the speed control thread).
- Move the UI/comms channel processing into the new thread.
- Add a new channel to join the new thread to the torque control thread.
- On a regular timer, send a query to the torque control thread to retreive the rotor speed. Alternatively, the QEI thread could be adjusted to have an extra channel input so that the speed control thread could query the QEI.
- After the speed control thread has performed the algorithm to determine the new demand tangential torque, send the result to the torque control thread through the channel.

In this way, the speed control thread can take advantage of a full 62.5 MIPS. Speed ramping, damping, filtering, or predictive torque control could all be implemented.

2 Processing Blocks

IN THIS CHAPTER

This module provides a number of standard computation functions that are utilised in motor control. These are outlined below.

- PID Calculation Routines
- Clarke & Park Transforms
- Sine & Cosine lookup

2.1 PID Calculation Routines

The processing blocks module provides the following PID calculation routines. The coefficients are signed 16 bit fixed point.

```
#include "pid_regulator.h"

void init_pid( int Kp, int Ki, int Kd, pid_data *d );

int pid_regulator( int set_point, int actual, pid_data *d );

int pid_regulator_delta( int set_point, int actual, pid_data *d );

int pid_regulator_delta_cust_error( int error, pid_data *d );

int pid_regulator_delta_cust_error_speed( int error, pid_data &d );

int pid_regulator_delta_cust_error_Iq_control( int error, pid_data &iq );

int pid_regulator_delta_cust_error_Id_control( int error, pid_data &id );
```

`init_pid` initialises the `pid_data` structure values with the coefficient values for *Kp*, *Ki* and *Kd*. These value are the proportional, integral and differential coefficients controlling the PID controller. The compile time constant `PID_RESOLUTION` determines how many fractional bits are present in these coefficients.

`pid_regulator` performs a standard PID calculation using the `set_point` and `actual` values. It calculates the error and applies the PID coefficients and then returns the result. The returned error will be applied to the `set_point` value.

`pid_regulator_delta` performs a standard PID calculation using the `set_point` and `actual` values. It calculates the error and applies the PID coefficients and then returns the resulting error.

`pid_regulator_delta_cust_error` performs a standard PID calculation using a previously calculated error value. It calculates the error and applies the PID coefficients and then returns the resulting error.

`pid_regulator_delta_cust_error_speed`, `pid_regulator_delta_cust_error_Iq_control` and `pid_regulator_delta_cust_error_Id_control` are customized control PIDs that limit the output to a specific range appropriate to the variable being controlled.

2.2 Clarke & Park Transforms

The processing blocks module provides the following Clarke and park transforms. The internal coefficients are all fixed point values.

```
#include "park.h"
void park_transform( int *Id, int *Iq,
                     int I_alpha, int I_beta,
                     unsigned theta );

void inverse_park_transform( int *I_alpha, int *I_beta,
                             int Id, int Iq,
                             unsigned theta );


#include "clarke.h"
void clarke_transform( int *I_alpha, int *I_beta,
                       int Ia, int Ib, int Ic );

void inverse_clarke_transform( int *Ia, int *Ib, int *Ic,
                               int alpha, int beta );
```

Each function has the calculation outputs passed as references (e.g. as pointers in C) and the inputs passed as normal arguments. The Park transform moves the rotating frame of reference of values relative to the stator (and the QEI and ADCs) into the frame of reference of the rotor. The Clarke transform takes 3-vector values which are gathered by measurement of the three coils and transforms them into a 2-vector value. This is possible because the 3-vectors have only 2 degrees of freedom, the current in one of the coils being the sum of the other two. See a description of Field Oriented Control for more information.

2.3 Sine & Cosine lookup

The sine and cosine functions are largely provided for use in the Park transforms, but may be used by other functions if required. The sine table provided has a 256 entry lookup. This is convenient for a 1024 step full circle QEI on a 4 pole motor, since each angular increment in the QEI represents 4 times the electrical angle.

Thus the 0-1023 range merely needs to be looked up in a 0-255 range, with the upper 2 bits truncated.

The lookup functions provided are as follows.

```
#include "sine_lookup.h"

inline long long sine( unsigned angle );
inline long long cosine( unsigned angle );
```

3 Display and Shared IO Interface

IN THIS CHAPTER

This module provides a details on the display interface and shared IO manager used in the XMOS Motor Control Development Platform.

The shared IO manager interfaces to the following components on the board:

- A Newhaven Display NHD-C12832A1Z-FSW-FBW-3V3 128 x 32 pixel monochrome LCD display via a SPI like interface.
- The 4 push button surface mount switches (marked A-D).

Provision could also be made in this thread to drive the 4 surface mount LEDs next to switches A-D.

3.1 Hardware Interface

The interface is implemented using 11 pins in total, including:

- 1 x 4 bit port to control the display address / data signal.
- 3 x 1-bit ports for the display chip select, serial clock and data signals.
- 1 x 4-bit ports for the buttons A-D.

3.2 Operation

The following files are used for the display and shared IO manager.

`lcd.h`
: Prototypes for LCD functions.

`lcd.xc`
: LCD driver functions.

`lcd_data.h`
: Contains the lcd driver font map.

`lcd_logo.h`
: Contains the XMOS logo as a unsigned char array.

`shared_io.h`
: Header for the main shared IO server and defines commands this thread uses.

`shared_io.xc`
: Contains the main shared IO server routine.

The shared IO manager that interacts with the hardware is a single thread with three channels connecting to it. The function is called from main with parameters passing a structure containing the appropriate ports into it. The server thread prototype is:

```
void display_shared_io_manager( chanend c_speed[],
                                REFERENCE_PARAM(lcd_interface_t, p),
                                in port btns,
                                out port leds )
```

The purpose of each argument is as follows:

`c_speed`
: An array of speed control channel for controlling the motors.

`p` A reference to the control structure describing the LCD interface.

`btns`
: A 4 bit input port attached to the buttons.

`leds`
: A 4 bit output port attached to the leds.

The main shared IO manager is constructed from a `select` statement within a `while(1)` loop, so that it gets executed repeatedly.

`case t when timerafter(time + 10000000) :> time :`
: Timer that executes at 10Hz. This gets the current speed, current Iq and speed setpoint from the motor control loops and updates the display with the new values. It also debounces the buttons.

`case !btn_en => btns when pinsneq(value) :> value:`
: Execute commands if a button is pressed.

The switches are debounced by setting the `but_en` guard signal to two whenever a button is pressed. The 10Hz timer in the select statement decrements the value by one, if the value is not 0, on each iteration though its loop. Therefore, after a minimum of 200ms and a maximum of 300ms the switch is re-enabled.

3.3 LCD Communication

Communication with the LCD is done using a `lcd_byte_out` function. This communicates directly with the ports to the display. The protocol is unidirectional SPI

with a separate command / data pin which specifies if the current data transfer is a command or data word.

The procedure for sending a byte to the display is:

- Select the display using the CS_N signal.
- Set the address / data flag.
- Clock out the 8 bits of data MSB first by: - Setting the data pin to the bit value. - Setting clock high. - Setting clock low.
- Deselect the display using the CS_N signal.

The following functions are provided that use the `lcd_byte_out` function to send data to the display:

`lcd_clear`
: This wipes the display by writing blank characters into the displays output buffer.

`lcd_draw_image`
: This takes an unsigned char array of size 512 bytes and writes it to the display. Hence, it can be used to display images on the display.

`lcd_draw_text_row`
: Writes a row of 21 characters to the display on the row specified by lcd_row (0-3).

The display is configured as 128 columns x 4 byte rows, as the byte writes the data to 8 pixel rows in one transfer. A 5x7 pixel font map is provided for the characters A-z, a-z, 0-9 and standard punctuation.

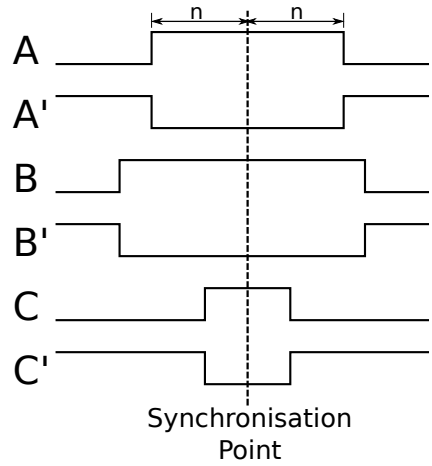
The command set for the display is defined in the datasheet. When sending data to the display it is best to try to send the data as fast as possible. This is because the display has to be turned off, whilst the data is being written to it. Therefore, writing large amounts of data on a regular basis can cause the display to flicker.

4 Pulse Width Modulation

IN THIS CHAPTER

The PWM driver code is written using a *client server* model. The client functions are designed to be run from either the main control loop or a separate thread that sits between the control loop and the PWM server thread (dependant on timing constraints defined by the speed of the control loop). The client and server communicate with each other through a channel and also some shared memory. Consequently, the client and server threads must reside on the same core.

The PWM implementation is centre synchronised. This means that the output is of the form shown in the figure. Having centrally synchronized PWM reduces the number of coincident edges, thus reducing switching noise as fewer FETs are switched simultaneously.



4.1 Configuration

The PWM module has three modes of operation defined, plus a number of other options. The modes are defined in `dsc_config.h` that is part of the application code.

4.1.1 PWM Modes

The PWM operation mode can be one of the following options:

- An inverted mode, whis operates a three leg 180 degree inverter by ensuring that the HI and LO sides of the inverter are switched in a complementary manner
- A simple mode, which operates a three leg inverter by switching the HI side and then applying PWM to the low side of the inverter to achieve simple commutation

4.1.2 Dead Time

The dead time for the inverted mode is defined using the PWM_DEAD_TIME configuration. This is in units of 10ns when using the default reference clock of 100MHz. The dead time is the short period of time between the non-inverted and the inverted PWM lines changing. During this time, neither side of the H-bridge is connected to the motor. The two signals are staggered by the dead time so that the two sides of the H-bridge are never ON at the same time, and do not change simultaneously.

4.1.3 PWM Resolution

PWM resolution is defined using PWM_MAX_VALUE. The value defined here sets the frequency of the PWM. The relationship between PWM_MAX_VALUE, XS1_TIMER_HZ and PWM frequency (PWM_FREQ) is defined in the equation below. XS1_TIMER_HZ is defined at compile time by the ReferenceFrequency identifier in the project XN file. By default this reference frequency is 100MHz so XS1_TIMER_HZ would have a value of 100,000,000.

```
PWM_FREQ = XS1_TIMER_HZ / (PWM_MAX_VAL)
```

So with an example value of PWM_MAX_VALUE being 4096, the PWM_FREQ will be 24,414Hz. Likewise, for a PWM frequency of 25Hz, the PWM_MAX_VAL would be 100000000 / 25 = 4000000. The maximum value for the PWM_MAX_VAL is 0x3FFFFFFF-PWM_DEAD_TIME, because the timestamps used to calculate the triggering of the PWM need to be no more than half of a 32 bit word into the future. This gives a minimum PWM period of around 0.1Hz.

In the FOC example, the ReferenceFrequency is set to 250MHz. This changes the calculation and gives the following:

```
PWM_FREQ = 250000000 / 4096 = 61.035 kHz
```

The PWM_MAX_VALUE is the total length of time which each PWM cycle occupies. Because the PWM is symmetrical, there are only PWM_MAX_VALUE / 2 steps that are available for positioning the rising PWM edge, and likewise for the falling PWM edge. Thus the number of bits available for a PWM_MAX_VALUE of 4096 is actually 11 bits. Note however that the update_pwm client function will shift the input value down by one bit, so that the client function should still provide a duty cycle value in the range of 0 to PWM_MAX_VALUE-1.

The following table gives some values of associated resolution and period for 250MHz and 100MHz clock rates and symmetrical.

Clock / MHz	Period / Hz	Resolution / bits
250	488,281	8
250	122,070	10
250	61,035	11
250	30,517	12
250	1,907	16
100	195,312	8
100	48,828	10
100	24,414	11
100	12,207	12
100	762	16

4.1.4 Locking the ADC trigger to PWM

In some implementations it is desirable to lock the ADC conversion trigger to the PWM. This allows the system to sample the ADC at a specific point in the PWM period (such as when the lower leg is guaranteed to be on). This is enabled using the LOCK_ADC_TO_PWM definition. The PWM server thread function `pwm_service_inv_triggered` should be used, which has extra arguments to include a channel which signals the ADC module, and a dummy port which is used as a timing source for the ADC trigger action. This port is not actually driven, and a port which is not pinned out of the device can be used.

4.2 PWM Server Usage

The usage for each mode is described below. The PWM server needs to be instantiated on the same core as the PWM client. One of the following is required to be included.

- `pwm_service_simple.h`
- `pwm_service_inv.h`

4.2.1 Inverter Mode

To instantiate the PWM service, one of the following function needs to be called. The first is used when ADC synchronization is required, for which `LOCK_ADC_TO_PWM` must be defined.

```
void do_pwm_inv_triggered( chanend c_pwm,
      chanend c_adc_trig,
          in port dummy_port,
          buffered out port:32 p_pwm[],
          buffered out port:32 p_pwm_inv[],
          clock clk);

void do_pwm_inv( chanend c_pwm,
      buffered out port:32 p_pwm[],
      buffered out port:32 p_pwm_inv[],
      clock clk);
```

`chanend c_pwm` is the channel used to communication with the client side.

`chanend c_adc_trig` is the channel used to communicate the triggering of the ADC conversion to the ADC thread.

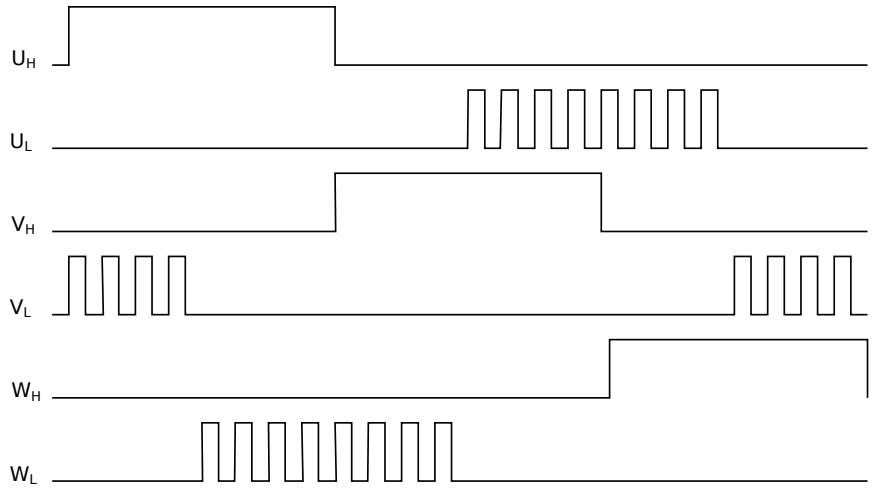
`in port dummy_port` is an unused port that is used to consistently trigger the ADC conversion. This port can overlap other used ports at it is never written to and the input value is never used.

`buffered out port:32 p_pwm[]` and `buffered out port:32 p_pwm_inv[]` are arrays of 1 bit ports with an array length of 3 that are used for the HI and LO sides of inverter respectively.

`clock clk` is the clock block that the PWM thread uses for timing output.

4.2.2 Simple commutation mode

This mode is designed for simple commutation of a brushless DC motor. An example of the output of this mode is shown in the figure below. Only the low sides of the three bridges is driven by the PWM service. It is up to the associated application to drive the high sides of the bridges in conjunction. The application must make sure that the low and high sides of the same bridge are never driven together.



To instantiate the PWM service in this mode the following function needs to be called.

```
void do_pwm_simple( chanend c_pwm,
      buffered out port:32 p_pwm[],
      clock clk);
```

`chanend c_pwm` is the channel used to communication with the client side.

`buffered out port:32 p_pwm[]` is an array of 1 bit ports with an array length of 3 that are used for the HI or LO sides of the inverter respectively.

`clock clk` is the clock block that the PWM thread uses for timing output.

4.3 PWM Client Usage

Because the client and server use shared memory to communicate, the PWM client functions must be operated on the same core as the server. The usage of the client functions in the various operational modes are described below. The following must be included to call the client functions, depending on the commutation mode chosen:

- `pwm_cli_simple.h`
- `pwm_cli_inv.h`

4.3.1 Inverter Mode

The only call required to update the PWM values that are currently being output is listed below. It takes only two arguments, the channel to the PWM server and an array of size three containing unsigned integers that must be between 0 and PWM_MAX_VALUE.

```
void update_pwm_inv( chanend c, unsigned value[]);
```

This function will process the values and pass them to the PWM service thread.

4.3.2 Basic BLDC commutation mode

The basic BLDC commutation mode client operates slightly differently to achieve the waveform shown in the previous figure. The function call listed below must be utilised.

Only a single output is active at any one time and this channel must be identified using the pwm_chan argument, this is a value between 0 and 2. The corresponding inverted leg of the inverter needs to be switched manually in the control thread. Please refer to the app_basic_bldc application and associated documentation.

```
void update_pwm_simple( chanend c,
      unsigned value,
      unsigned pwm_chan );
```

4.4 PWM Service Implementation

The PWM service is designed as a continuously running loop that cannot be blocked. This is important to ensure continuous output as stalling an output on an inverter in any application could result in serious failure of the appliance that is being driven.

To achieve the behaviour needed the PWM services are all written in assembly language. This is done to achieve a fine grained control over the instruction sequences required to load up the buffers in the ports and also the port timers.

The PWM service pulls the required data from a shared memory location. This is a *double buffered* scheme where the client will update the memory area that is not currently in use and then inform the service via a channel which memory location it should look at for the output data. The update sequence is looked at in more detail in the discussion of the client implementation.

4.4.1 PWM service port initialisation `pwm_service_inv.xc`

This file achieves a number of functions. The primary function is a wrapper that is called to start the PWM service running. This configures the port and then enters the main loop for the PWM service.

Firstly three legs of the inverter drive are configured to be attached to the clock block and have an initial output of 0. This is deemed to be a safe start-up configuration as all drives are switched off.

Then, in the loop, the *inverted* ports are configured to output the inverse or complementary of the data that is put into the buffers. This means that only a single data set need be maintained and removes the need for inverting the data using the instruction set as this is done by the port logic.

Following the loop that sets up the individual PWM channels is the configuration for the ADC triggering port. This is an input port that is attached to the same clock block as the PWM output ports. An input port that overlaps other in use ports (as described in the usage section above) will not affect their operation. The dummy port is just used for timing synchronisation when signalling the ADC.

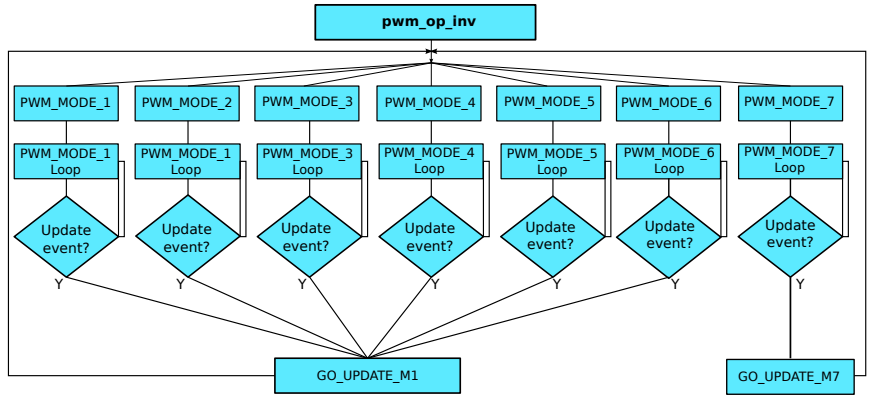
Finally the clock block is started.

Once the ports have been configured the output will remain in the initialised state until the thread receives notification from the client thread that data is available in the shared memory for output. It is important to wait for the first client update otherwise there is a risk of output uninitialised data which may damage the drive circuitry.

Once this information is received the main loop is entered.

4.4.2 PWM service main loop `pwm_op_inv.S`

The operation of the main loop is best described visually as in the flow chart shown in the figure. The entries in the flow chart relate directly to the labels within the main loop. A brief overview of each part of the main loop are given below. These should be consulted alongside the comments that reside in the code itself.



The code begins at the pwm_op_inv entry point. This begins by running a standard callee save. This preserves any registers that we will clobber as part of the operation of this function. The arguments to the function are then stored on the stack itself in sp[8:11]. This ensures we have access to them later.

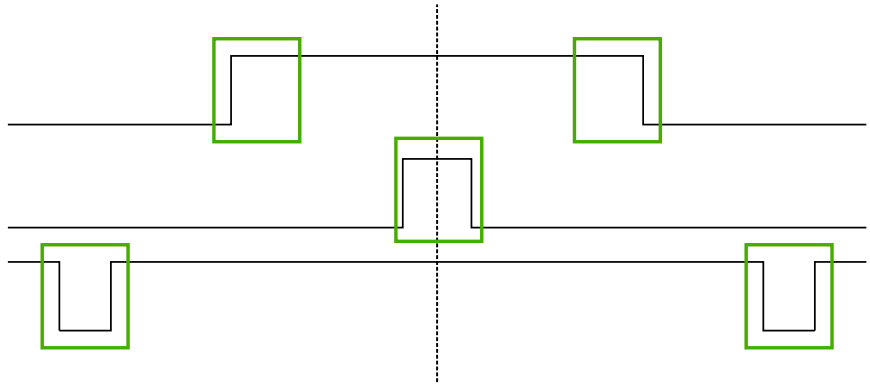
Following this the registers are moved around into the configuration we require and data is read from the t_data_out structure after calculating the appropriate pointers. The port resource IDs are then loaded into registers and the *mode* of operation is read and the port timer read to initialise the synchronisation point.

The code then branches to the appropriate mode according to the mode value that has been read from the data structure provided to it by the client.

4.4.3 Loop modes

By default, the PWM is configured to be unable to do the top and bottom 0.5% of the duty cycle range. This prevents the system having to deal with the unusual cases where the output is a very short or very long pulse. If the constant *PWM_CLIPPED_RANGE* is removed from the *dsc_pwm_common.h* file, then the PWM will be able to cope with the full duty cycle range.

In this case, to achieve the required output, the port buffers are used to create the extremely short or long pulses as shown in the figure. The green boxes indicate a buffer of data that is output from the port.



This method of output requires a combination of one or two buffer outputs depending on the length of these pulses. Rather than calculate these during runtime the client will ascertain the particular combination of outputs required and then will define the mode. The different buffering output modes are individually implemented to reduce branching overhead within the loop.

At the entrance to the loop mode (taking PWM_MODE_4 as the working example) the mode value is replaced with the channel end resource ID. We then enter the core of the PWM service loop. The loop will setup each of the ports in sequence, calculating the appropriate port timer value from the data set that is provided by the client.

When the option to lock the ADC to PWM is required then the system will block on the in instruction while it waits for the timer on the dummy port. Once the port timer reaches the required value the thread will output the token to the ADC thread.

If the ADC to PWM lock is not utilised then the thread will pause on the next setpt instruction until that particular port timer value is met and the data is output. The ports are loaded in reverse order to turn them off at the correct time. Once all of the channels are reloaded the thread will check for data on the update channel. If data is found then it will immediately enter GO_UPDATE_M1 otherwise it will continue through the loop calculating the next synchronisation point and looping back to the top of the output sequence.

If the system branches to update then it will execute a sequence very similar to the entry of the function, reading the data out of the data structure and setting up the relevant memory pointers. The update for PWM_MODE_[1:6] loops are all the same. In the case of PWM_MODE_7 the update sequence is slightly different due to the fact that the event is likely to occur when one of the channels is high. This means that a further output is required before receiving the update from the client.

MODE	PWM pulse pattern
1	3 short
2	2 short + 1 standard
3	1 short + 2 standard
4	3 standard
5	1 short + 1 standard + 1 very long
6	1 very long + 2 standard
7	2 short + 1 very long
unused	1 standard + 2 very long
unused	1 short + 2 very long
unused	3 very long

To drive the ports, the PWM uses the 32 bit buffered ports. The *short* pulse, which is known as a *SINGLE* internally, is one where the duration of the pulse is shorter than 32 reference clock cycles, and the buffer is silled with an appropriate bit pattern to generate the pulse. The *very long* pulses, known as *LONG_SINGLE*, are within 31 reference clocks of the PWM_MAX_VALUE and are therefore similar to the *short* pulses. The *standard* pulses, known as *DOUBLE*, output both the rising edge and falling edge as separate words, hence the name double.

Note that the mode consisting of three very long pulses is not catered for. The client clips the values if this case is attempted.

4.5 PWM Client Implementation

Before a specific client for the inverting mode starts, it needs to let the server thread know where its shared memory control buffers are. A call to `pwm_share_control_buffer_address_with_server` will pass this information to the server. Each client can only talk to one server, but since multiple client/server components can co-exist, each must have its own memory buffer.

The PWM client is required to do a number of functions to provide the correct data to the PWM service that outputs the correct values and timings to the ports. The PWM client must:

- Calculate the output values
- Calculate the timing values (taking into account dead time)
- Sort the ports into time order

- Ascertain the loop mode required
- Maintain the shared data set, including which buffer is in use and which one can be updated

Taking the inverter mode as our working example (located in `module_dsc_pwm/src/dsc_pwm_cli/pwm_cli_inv`) the function `update_pwm_inv` first saves the PWM values for later use and then initialises the channel ordering array to assume a sequential order of output.

If the non-clipped PWM range is being used, then following this the calculation of the timings and output values are done for each of the channel. This is done by passing the relevant PWM value and data set references to `calculate_data_out_ref`. This function also ascertains the type of output which can be one of three values `SINGLE`, `DOUBLE` and `LONG_SINGLE`.

Once the calculations for each of the PWM channels is completed they can be ordered. This is done using the `order_pwm` function. This orders the values in the channel ID buffer and also works out the loop mode that is required.

When the values have been ordered and the loop mode calculated the buffer number is passed to the PWM service to indicate an update.

5 Analogue to Digital Converter (ADC) Interface

IN THIS CHAPTER

The analogue to digital interface currently provided is written for the 7265. This provides a clocked serial output following a sample and hold conversion trigger signal. The physical interface of the ADC is not covered in detail as the interface for ADC's will vary from manufacturer to manufacturer. Examples of the interfaces for the MAX1379 and LTC1408 is also available in this module.

Besides the client and server interfaces the key issue discussed in this section is the synchronisation of the ADC to the PWM and how this is achieved on the ADC side.

The preprocessor define LOCK_ADC_TO_PWM must be 0 or 1 for off and on respectively. This defines whether the ADC readings are triggered by the PWM so that measurements can be taken at the appropriate point in the PWM cycle.

5.1 ADC Server Usage

The header file *adc_7265.h* and function call *adc_7265_triggered* are required to operate the ADC software as a server. This server is utilised in the case where the ADC is locked to the PWM.

See the API section for a full description of the function call.

5.2 ADC Client Usage

The functions below are the primary method of collecting ADC data from the ADC service.

The client can be utilised as follows:

```
#include "adc_client.h"

void do_adc_calibration(chanend c_adc);

{int, int, int} get_adc_vals_calibrated_int16(chanend c_adc);
```

`do_adc_calibration(...)` is used to initialise the ADC and calibrate the 0 point. The server will enter a mode where the next 512 samples are averaged, and the result is considered to be the zero point of further readings.

`get_adc_vals_calibrated_int16(...)` is used to get the three ADC values with the zero calibration, offset and scaling applied to get a signed 16 bit value. This is a multiple return function in channel order.

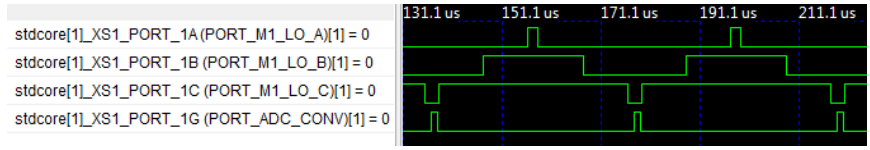
5.3 ADC Server Implementation

The ADC server implementation discussed here is the triggered variant of the ADC code. The ADC server first configures the ports as clocked inputs and outputs. Following this the main loop is entered.

ADC readings are triggered by the receipt of a trigger control token over the channel. A token is used as this offers minimum latency for channel communication. Following the token being received the ADC values are read after a time constant that is calibrated to align with the appropriate measurement point.

ADC values can be requested from the server at any point.

The screenshot shows three PWM ports and the ADC conversion trigger port. It shows that the ADC trigger rising edge, where the sample hold is made, is directly centred in the PWM channel inter-cycle period.



6 Application Level Communications Interfaces

IN THIS CHAPTER

This module provides a details on the higher application level communication interfaces used in the XMOS Motor Control Development Platform.

The motor control platform has been written to take advantage of the XMOS Ethernet and XMOS CAN open source components. With suitable compile time options, the example applications will automatically contain ethernet or CAN control modules. A high level communication server has been written for each, and these interface to the standard MAC and PHY components for the two protocols. These high level server threads are called *do_comms_eth* and *do_comms_can*.

Documentation for the CAN, Ethernet and TCP/IP XMOS components can be found in the software guides for those components, either on the xmos.com website or from the relevent open source respositories.

A LabView runtime based control application, suitable for both CAN and Ethernet control, is included with the software release, in the *gui* subdirectory.

6.1 do_comms_eth

The thread do_comms_eth interfaces to the TCP/IP stack and provides a server interface on the TCP port defined by TCP_CONTROL_PORT (this is typically defined as 9595). See the documentation for the *sc_xtcp* and *sc_ethernet* modules, which describe the use of the TCP/IP service and Ethernet services.

After configuring the TCP port and TCP/IP stack interface, the thread sits in a while(1){} loop processing TCP/IP events. The following actions are performed based on the event type:

- `XTCP_NEW_CONNECTION` - No action is taken.
- `XTCP_RECV_DATA` - Main processing function, described below.
- `XTCP_SENT_DATA` - Closes the send request by sending a 0 byte packet.
- `XTCP_REQUEST_DATA` / `XTCP_RESEND_DATA` - Sends the data generated during the XTCP_RECV_DATA event to the client.
- `XTCP_CLOSED` - Closes the connection.

The main processing function receives a packet from the client and processes it, responding with data as appropriate. The format of the accepted packets are:

`^1|xxxx` - This sets the speed of the motors to the value given in the 4 digit number. The value is given as a hexadecimal number.

`^2|` - This requests the current state of the motor be sent to the remote client. The server replies with the text string

```
^2|aaaa|bbbb|cccc|dddd|eeee|ffff|gggg|hhhh|iiii|jjjj|kkkk|llll|mmmm|nnnn|oo
  ↪ |pp

aaaa - Speed of motor 1
bbbb - Speed of motor 2
cccc - Current Ia for motor 1
dddd - Current Ib for motor 1
eeee - Current Ic for motor 1
ffff - Iq Set Point for motor 1
gggg - Id output for motor 1
hhhh - Iq output for motor 1
iiii - Current Ia for motor 2
jjjj - Current Ib for motor 2
kkkk - Current Ic for motor 2
llll - Iq Set Point for motor 2
mmmm - Id output for motor 2
nnnn - Iq output for motor 2
oo   - Fault flag for motor 1
pp   - Fault flag for motor 2
```

The files for this thread are in `control_comms_eth.xc` and `control_comms_eth.h`

6.1.1 Customizing the TCP/IP packet

The ethernet and TCP/IP communications module is an example of a simple query and reply protocol using a fixed packet format. Many alternatives exist for communication with the device through ethernet, and an incomplete list is given below.

- Remove TCP/IP altogether and send and receive ethernet layer 2 packets of a given fixed format
- Use UDP instead of TCP, with the same packet format
- Alter the packet format to a proprietory alternative
- Integrate an industry standard communication protocol on either Ethernet layer 2 or TCP/IP. Modbus/IP is a good example of this.

6.2 do_comms_can

XMOS provides an independent CAN component in the *sc_can* open source repository. See the documentation for that component for more details of the CAN PHY interface.

This thread is similar in operation to the do_comms_eth thread, and provides the same interface to the speed_control_loop.

It works by configuring the CAN interface and then sitting in a while(1){} loop receiving packets from the CAN interface. Once the thread receives a packet from the client, it looks at the command type, and processes it accordingly.

- If command type (byte 2) equals 1, then this command replies with the current speed and other measured data.
- If command type (byte 2) equals 2, then this command sets the desired speed from the data supplied in the packet.

The format of a received CAN packet is:

- 2 bytes - sender address - used to address the return packet if required.
- 1 byte - command type
- 4 bytes - desired speed in big-endian order if command equals 2.

The format of a transmitted CAN packet is:

- 4 bytes - current speed in big-endian order.
- 2 bytes - the measured Ia for motor 1.
- 2 bytes - the measured Ib for motor 1.

The files for this thread are in `control_comms_can.xc` and `control_comms_can.h`

6.2.1 Customizing the CAN communcation module

The *sc_can* XMOS CAN component supports the CAN PHY, as used in the example communication system given. Many alternative configurations exist, and an incomplete list is given below.

- Use the CAN LLC supported by the *sc_can* module to provide a set of CAN registers containing the read and write parameters of the application.
- Alter the CAN packet format.

6.3 Replacing the communications module

The ethernet/TCP and CAN communications systems are only examples of the many alternatives that can be implemented using the flexible XMOS architecture. Both of these communication modules are implemented as a collection of threads running in the device. They communicate with the motor control thread using channels.

By replacing the threads with alternative physical interface and protocol threads, that communicate with the motor control thread using the same channel messages, it becomes easy to produce a different external control scheme. If the messages

which are currently sent on the channels are not flexible enough, then they too can be altered to provide additional functionality.

An incomplete list of other external control schemes is given below.

- Simple GPIO control
- UART
- SPI
- I2C
- Clocked parallel port
- Bluetooth
- EtherCAT
- RS485

7 Hall Sensor Interface

IN THIS CHAPTER

The hall sensor interface is used for measuring speed and estimating the position of the motor.

7.1 Hall Sensor Usage

The hall sensor input module provides a number of functions and options in its usage. A listing of the available functions is given below.

```
#include "hall_input.h"

void run_hall_speed_timed( chanend c_hall,
      chanend c_speed,
      port in p_hall,
      chanend ?c_logging_0,
      chanend ?c_logging_1 );

void do_hall( unsigned &hall_state,
      unsigned &cur_pin_state,
      port in p_hall );

select do_hall_select( unsigned &hall_state,
      unsigned &cur_pin_state,
      port in p_hall );
```

`run_hall_speed_timed(...)` provides a server thread which measures the time between hall sensor state transitions on a 4 bit port as provided on the motor control platform. This functions implementation is described in more detail below.

`do_hall(...)` simply writes the next hall state into the *hall_state* variable and current pin state into the cur_pin_state variable.

`do_hall_select(...)` is the same as `do_hall` but is a select function. This function is used in the basic BLDC demonstration application.

7.2 Hall Sensor Client

When using the hall sensor server thread as described above, the information may be accessed by using the client functions as listed below.

```
#include "hall_client.h"

{unsigned,unsigned,unsigned} get_hall_pos_speed_delta(chanend c_hall);
```

`get_hall_pos_speed_delta(...)` will request and subsequently return the theta, speed and delta values respectively from the hall input server thread. The theta value is an estimated value, speed is in revolutions per minute (RPM) and delta is currently used for debugging purposes.

7.3 Hall Sensor Server Implementation

This code is currently considered experimental

The function `run_hall_speed_timed(...)` provides a thread that handles hall sensor input functions, speed and angle estimations.

After initialising the ports and initialising the current hall sensor state the code enters a startup phase. This is where an ideal theta value is passed to the client as the motor is not yet actually turning, so no angular estimation can be made. This continues until the hall sensor thread has received two transitions.

Following the initial startup sequence the hall sensor thread enters the main operational loop. This comprises of a select statement that handles either a request for information from the clients, a timeout to detect no rotation or a state transition on hall sensor.

When a new transition is received the new hall state is stored and the current theta base value is updated. This base value is defined as the angular location of the hall sensor within the motor. The system then defines what the next hall sensor state it should wait for will be.

Once the base angle and next state values have been updated the timing calculations are completed to define the speed and angle calculation. Speed calculation is defined by looking for a full mechanical rotation of the motor where it returns to a defined state.

When the thread receives a request for speed and angle information these are calculated and then delivered over the change. The angle estimation is done by considering the time the motor has taken to travel over a hall sensor sector. It assumes that the hall sensor data is requested at a regular time over the sector (which as it is blocked by the PWM it will be in the example FOC implementation).

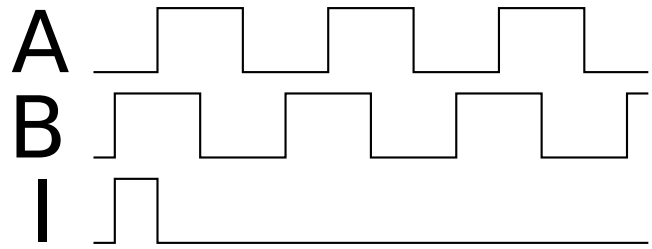
Once the values are calculated they are provided to the client over the channel.

8 Quadrature Encoder Input

IN THIS CHAPTER

The quadrature encoder input (QEI) module is provided with a library for both running the thread that handles the direct interface to the pins and also for retrieving and calculating the appropriate information from that thread.

The particular interface that is implemented utilises three signals comprising of two quadrature output (A and B) and an index output (I). A and B provide incremental information while I indicates a return to 0 or origin. The signals A and B are provided out of phase so that the direction of rotation can be resolved.



8.1 Configuration

The QEI module provides one or multiple QEI device versions of the server. If more than one QEI device is interfaced to the XMOS device, then the designer can opt to use multiple single-device QEI server threads, or one multi-device thread.

The multiple-QEI service loop has a worst-case timing of 1.4us, therefore being able to service two 1024 position QEI devices spinning at 20kRPM.

8.2 QEI Server Usage

To initiate the service the following include is required as well as the function call shown. This defines the ports that are required to read the interface and the channel that will be utilised by the client thread. The compile time constant *NUMBER_OF_MOTORS* is used to determine how many clients and ports are serviced by the multi-device QEI server.

```
#include "qei_server.h"

void do_qei( streaming chanend c_qei, port in pQEI);

void do_qei_multiple( streaming chanend c_qei[NUMBER_OF_MOTORS], port in
  ↪ pQEI[NUMBER_OF_MOTORS]);
```

8.3 QEI Client Usage

To access the information provided by the quadrature encoder the functions listed below can used.

```
#include "qei_client.h"

{ unsigned, unsigned, unsigned } get_qei_data( chanend c_qei );
```

The three values are the speed, position and valid state. The position value is returned as a count from the index zero position and speed is returned in revolutions per minute (RPM).

The third value indicates whether the QEI interface has received an index signal and therefore that the position is valid.

9 API

IN THIS CHAPTER

9.1 Configuration Defines

The file dsc_config.h must be provided in the application source code. This file can set the following defines:

PWM_DEAD_TIME

This is the period, in 10ns intervals, which is not counted towards the PWM time as the PWM is output.

PWM_MAX_VALUE

The PWM input is clamped to this value

LOCK_ADC_TO_PWM

If this is defined, the PWM outputs synchronization information to a channel and dummy port, allowing the ADC module to synchronize the ADC measurement to the dead time when all PWM channels are off.

NUMBER_OF_POLES

This is the number of poles in the motor. It is therefore ratio of the number of electrical rotations to each physical rotation. If a motor has a single winding per coil, then it is called a 2 pole motor. Two sets of windings per coil makes a four pole motor, and so on.

USE_CAN

When defined, the CAN controller in included in the executable. This option is mutually exclusive with Ethernet.

USE_ETH

When defined, the Ethernet controller is included in the executable. This option is mutually exclusive with CAN.

TCP_CONTROL_PORT

When the Ethernet controller is included, this is the TCP port that the server listens on, for receiving control information.

MIN_RPM

The minimum RPM that the controllers can set.

MAX_RPM

The maximum RPM that the controllers can set.

9.2 External modules

For documentation on the Ethernet, CAN and PWM modules, see the relevent XMOS software module documentation.

9.3 ADC

9.3.1 Client functions

`void do_adc_calibration(streaming chanend c_adc)`

ADC calibration sequence.

This switches the ADC server into calibration mode. After a number (512) of samples the server reverts to non-calibration mode, and subsequent calls to the function get_adc_vals_calibrated_int16 will take the measured average of the calibration period as an offset

This function has the following parameters:

`c_adc` the control channel to the ADC server

`{int, int, int} g get_adc_vals_calibrated_int16(streaming chanend c_adc)`

Get values converted from 14 bit unsigned to 16 bit signed and calibrated.

Read a set of current values from the motor, and convert them into a standardized 16 bit scale

This function has the following parameters:

`c_adc` the control channel to the ADC server

9.3.2 Server functions

`void adc_7265_triggered(streaming chanend c_adc[ADC_NUMBER_OF_TRIGGERS],`

```
                    chanend c_trig[ADC_NUMBER_OF_TRIGGERS],
                    clock clk,
                    out port SCLK,
                    port CNVST,
                    in buffered port:32 DATA_A,
                    in buffered port:32 DATA_B,
                    port out MUX)
```

Implements the AD7265 triggered ADC service.

This implements the AD hardware interface to the 7265 ADC device. It has two ports to allow reading two simultaneous current readings for a single motor.

This function has the following parameters:

`c_adc`	the array of ADC server control channels
`c_trig`	the array of channels to recieve triggers from the PWM modules
`clk`	an XCORE clock to provide clocking to the ADC
`SCLK`	the external clock pin on the ADC
`CNVST`	the convert strobe on the ADC
`DATA_A`	the first data port on the ADC
`DATA_B`	the second data port on the ADC
`MUX`	a port to allow the selection of the analogue MUX input

```
void adc_ltc1408_triggered(chanend c_adc[],
                           chanend c_trig[],
                           clock clk,
                           port out SCLK,
                           buffered out port:32 CNVST,
                           in buffered port:32 DATA)
```

Execute the triggered ADC server.

This is the server thread implementation for the LTC1408 ADC device.

This function has the following parameters:

`c_adc`	the array of ADC control channels
`c_trig`	the array of channels to receive triggers from the PWM modules
`clk`	the clock for the ADC device serial port
`SCLK`	the port which feeds the ADC serial clock

CNVST the ADC convert strobe

DATA the ADC data port

```
void run_adc_max1379(chanend c_adc[],
                     clock clk,
                     port out SCLK,
                     port out CNVST,
                     port out SEL,
                     in buffered port:32 DATA)
```

The server thread for the MAX1379 ADC device.

Implements the server thread for the MAX1379 ADC device

This function has the following parameters:

c_adc the array of ADC control channels

clk the clock for the ADC device serial port

SCLK the port which feeds the ADC serial clock

CNVST the ADC convert strobe

SEL the chip select for the ADC device

DATA the ADC data port

9.4 QEI

9.4.1 Client functions

```
{ unsigned, unsigned, unsigned } g get_qei_data(streaming chanend c_qei)
```

Get the position from the QEI server.

This function has the following parameters:

c_qei The control channel for the QEI server

This function returns:

the speed, position and valid state

9.4.2 Server functions

```
void do_qei(streaming chanend c_qei, port in p_qei)
```

Implementation of the QEI server thread.

This function has the following parameters:

c_qei The control channel used by the client

p_qei The hardware port where the quadrature encoder is located

```
void do_multiple_qei(streaming chanend c_qei[], port in p_qei[])
```

Implementation of the QEI server thread that services multiple QEI devices.

This function has the following parameters:

c_qei The control channels used by the client

p_qei The hardware ports where the quadrature encoder is located

9.5 Hall sensors

9.5.1 Client functions

```
{unsigned, unsigned, unsigned} g get_hall_pos_speed_delta(chanend c_hall)
```

Get position, speed and delta from a hall server.

The client library function for a hall sensor server

This function has the following parameters:

c_hall the channel for communicating with the hall server

```
void do_hall(unsigned &hall_state, unsigned &cur_pin_state, port in p_hall)
```

A blocking read of the hall port.

This function has the following parameters:

hall_state the output hall state

cur_pin_state
the last value read from the hall encoder port

p_hall the hall port

```
select do_hall_select(unsigned &hall_state,
                      unsigned &cur_pin_state,
                      port in p_hall)
```

A selectable read of the hall pins.

This selectable function becomes ready when the hall pins change state

This function has the following parameters:

hall_state the output hall state

`cur_pin_state`
: the last value read from the hall encoder port

`p_hall`
: the hall port

9.5.2 Server functions

`void run_hall(chanend c_hall, port in p_hall)`

A basic hall encoder server.

This implements the basic hall sensor server

This function has the following parameters:

`c_hall`
: the control channel for reading hall position

`p_hall`
: the port for reading the hall sensor data

`void run_hall_speed(chanend c_hall, chanend c_speed, port in p_hall)`

A hall encoder server that also calculates motor speed.

This implements the hall sensor server

This function has the following parameters:

`c_hall`
: the control channel for reading hall position

`c_speed`
: the control channel for reading the rotor speed

`p_hall`
: the port for reading the hall sensor data

```
void run_hall_speed_timed_avg(chanend c_hall,
                              chanend c_speed,
                              port in p_hall)
```

A hall encoder server that also calculates motor speed.

This implements the hall sensor server, where the speed is calculated using a timed average of many values.

This function has the following parameters:

`c_hall`
: the control channel for reading hall position

`c_speed`
: the control channel for reading the rotor speed

`p_hall`
: the port for reading the hall sensor data

```
void run_hall_speed_timed(chanend c_hall,
                          chanend c_speed,
```

```
                    port in p_hall,
                    chanend ?c_logging_0,
                    chanend ?c_logging_1)
```

A hall encoder server that also calculates motor speed.

This implements the hall sensor server, where the speed is calculated using a timed average of many values.

This function has the following parameters:

`c_hall`	the control channel for reading hall position
`c_speed`	the control channel for reading the rotor speed
`p_hall`	the port for reading the hall sensor data
`c_logging_0`	an optional channel for logging the hall data on port 0
`c_logging_1`	an optional channel for logging the hall data on port 1

9.6 Computational Blocks

```
void park_transform(int &Id, int &Iq, int I_alpha, int I_beta, unsigned theta)
```

Perform a Park transform.

A Park transform is a 2D to 2D transform which takes the radial and tangential components of a measurement (for instance the magnetic flux or total coil currents) and converts them to a rotating frame of reference. Typically this is the rotating frame of reference attached to the spinning rotor.

This function has the following parameters:

`Id`	the output tangential component
`Iq`	the output radial component
`I_alpha`	the input tangential component
`I_beta`	the input radial component
`theta`	the angle between the fixed and rotating frames of reference

```
void inverse_park_transform(int &I_alpha,
                            int &I_beta,
                            int Id,
                            int Iq,
                            unsigned theta)
```

Perform an inverse Park transform.

A Park transform is a 2D to 2D transform which takes the radial and tangential components of a measurement (for instance the magnetic flux or total coil currents) and converts them to a rotating frame of reference. Typically this is the rotating frame of reference attached to the spinning rotor.

This function has the following parameters:

`I_alpha` the output tangential component

`I_beta` the output radial component

`Id` the input tangential component

`Iq` the intput radial component

`theta` the angle between the fixed and rotating frames of reference

`void clarke_transform(int Ia, int Ib, int Ic, int &I_alpha, int &I_beta)`

Perform a clarke transform.

A Clarke transform is a 3D to 2D transformation where the 3D components have only 2 degrees of freedom. It is used to convert the three current values in the 120 degree separation coils into a radial and tangential component values.

This function has the following parameters:

`Ia` the parameter from coil A

`Ib` the parameter from coil B

`Ic` the parameter from coil C

`I_alpha` the output tangential component

`I_beta` the output radial component

`void inverse_clarke_transform(int &Ia, int &Ib, int &Ic, int alpha, int beta)`

Perform an inverse clarke transform.

The inverse Clarke transform is a 2D to 3D transformation where the 3D components have only 2 degrees of freedom. It is used to convert radial and tangential components of the current vector into the three coil currents.

This function has the following parameters:

`Ia` the output parameter for coil A

`Ib` the output parameter for coil B

`Ic` the output parameter for coil C

`alpha` the input tangential component

`beta` the input radial component

`int sine(unsigned angle)`

Look up the fixed point sine value.

This looks up the sine of a value. The value is the index into the sine table, rather than a particular angular measurement. The sine table has 256 entries, so each index is 1.4 degrees.

A table of 256 entries is suitable for an encoder angle measured in 1024 steps, attached to a 4 pole motor. Each encoder step is 360/1024 = 0.35 physical degrees, but this is worth 4 times as many electrical degrees, or 1.4 electrical degrees.

The result is in fixed point 18.14 format.

This function has the following parameters:

`angle` the index of the sine value to look up

This function returns:

the 18.14 fixed point sine value

`int cosine(unsigned angle)`

Look up the fixed point cosine value.

This looks up the cosine of a value. The value is the index into the sine table, rather than a particular angular measurement.

This function has the following parameters:

`angle` the index of the cosine value to look up

This function returns:

the 18.14 fixed point cosine value

9.7 Watchdog Timer

`void do_wd(chanend c_wd, out port wd)`

Run the watchdog timer server.

The watchdog timer needs a constant stream of pulses to prevent it from shutting down the motor. This is a thread server which implements the watchdog timer output.

The watchdog control port should have two bits attached to the watchdog circuitry. Bit zero will get a rising edge whenever the watchdog is to be reset, and bit one will have the pulse train.

This function has the following parameters:

`c_wd` the control channel for controlling the watchdog

`wd` the control port for the watchdog device

9.8 High level communications

9.8.1 Ethernet control

`void do_comms_eth(chanend c_commands[], chanend tcp_svr)`

Implement the high level Ethernet control server.

This control the motors based on commands from the ethernet/TCP stack

This function has the following parameters:

`c_commands` Array of command channels for motors

`tcp_svr` channel to the TCP/IP thread

9.8.2 CAN control

`void do_comms_can(chanend c_commands[], chanend rxChan, chanend txChan)`

This is a thread which performs the higher level control for the CAN interface.

Use it in conjunction with the thread 'from the module module_can.

This function has the following parameters:

`c_commands` Channel array for interfacing to the motors

`rxChan` Connect to the rxChan port on the canPhyRxTx

`txChan` Connect to the txChan port on the canPhyRxTx

9.9 LCD display and PHY reset

9.9.1 LCD

`lcd_interface_t`

The control structure for the LCD ports.

The display uses an I2C interface with an extra control signal to support selection between a data read/write and a control read/write. This extra signal is bit zero of the p_core1_shared member - which contains no other signals despite the name.

This structure has the following members:

`out port p_lcd_sclk`
i2c serial clock

`out port p_lcd_mosi`
i2c serial data

`out port p_lcd_cs_n`
: i2c chip select

`out port p_core1_shared`
: Display data/control select.

`void reverse(char s[])`

Reverse the order of bytes in the array.

This function has the following parameters:

`s` the byte array to reverse

`void itoa(int n, char s[])`

Convert an integer into a base 10 ASCII string.

This function has the following parameters:

`n` the integer to represent in string form

`s` the output byte array to contain the number

`void lcd_ports_init(lcd_interface_t &p)`

Initialise the LCD device.

This function has the following parameters:

`p` the LCD interface description

`void lcd_byte_out(lcd_interface_t &p, unsigned char i, int is_data)`

Write a byte to the LCD.

This function has the following parameters:

`p` the LCD interface description

`i` the byte to write

`is_data` a boolean indicating if the write is data or control information

`void lcd_clear(lcd_interface_t &p)`

Clear the LCD.

This function has the following parameters:

`p` the LCD interface description

`void lcd_draw_image(const unsigned char image[], lcd_interface_t &p)`

Draw an image on the LCD.

Draw an image on the LCD. The image is assumed to cover the complete LCD. The size of the LCD is 128 wide by 32 high.

This function has the following parameters:

`image` a byte array containing the image data.

`p` the LCD interface description

`void lcd_draw_text_row(const char string[], int lcd_row, lcd_interface_t &p)`

Write text to a row on the LCD.

Display a row of text. The LCD columns beyond the end of the string will be cleared.

This function has the following parameters:

`string` the ASCII string to display on the LCD

`lcd_row` the character row on which to display the string

`p` the LCD interface description

10 Resource usage

IN THIS CHAPTER

The table shows the resource usage for the main components in the system. The transforms are a functional library and thus do not have a thread or port usage.

Component	Threads	Memory	Channel Ends	1b Ports	4b ports
ADC	1	2.2KB	2	4	1
PWM	1	2.8KB	2	6	0
Transforms	0	264B	0	0	0
QEI	1	400B	1	0	1
Watchdog	1	120B	1	1	0
PID	0	300B	0	0	0

See the documentation for the ethernet and CAN software components for their resource usage.

10.1 MIPS

This table shows the FOC control loop worst case timing, against the number of threads running in the motor control core. These values were measured on a 500MHz core.

Number of threads	MIPS per thread	Loop time
4	125	7.9 us
5	100	10 us
6	83.3	12 us
7	71.4	14 us
8	62.5	16 us

For a single motor, using PWM, ADC, QEI and a control loop, only 4 threads are required on the motor core. Another core can be used to provide further functionality, or for a single motor, the remaining 4 threads in the motor core can be used for control and IO, giving a single core FOC motor control solution.

A dual motor, single core FOC solution can be created, by using the dual-QEI mode of the QEI server. The threads for such a solution would be:

- PWM for motor 1

- PWM for motor 2
- dual QEI
- Control loop for motor 1
- Control loop for motor 2
- ADC
- Watchdog and main application
- CAN PHY interface

Table of Contents