```
//
// This is the motor control code that will run on PRU 0
//

#include <stdint.h>
#include "pru_cfg.h"
#include "pru_intc.h"
#include "mem.h"
#include "pru0Lib.h"
#include "motorLib.h"
#include "pru0.h"

// Define input and output registers

volatile register uint32_t __R30;
volatile register uint32_t __R31;

/* Mapping Constant table register to variable */

volatile pruCfg CT_CFG __attribute__((cregister("PRU_CFG", near), peripheral));
volatile far pruIntc CT_INTC __attribute__((cregister("PRU_INTC", far), peripheral));

// Global pointer to memory stucture

shared_memory_t   *mem ;

// Global variables that allow us to handle GPIO

int  *clrGPIO1_reg ;
int  *setGPIO1_reg ;
int  *readGPIO1_reg ;

int  *clrGPIO3_reg ;
int  *setGPIO3_reg ;
int  *readGPIO3_reg ;

// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Subroutine to perform PRU initialization
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

void initPRU(void) {
        CT_INTC.SICR = PRU1_PRU0_EVT ;
        initGPIO();
        return ;
}

// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Subroutine to wait for an interrupt.
// It also clears the interrupt
// Toggles the PRU LED at interrupt rate
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

void waitForInterrupt(void) {
   while (!(__R31 & HOST0_MASK)) { } ;  // wait for interrupt
   CT_INTC.SICR = PRU1_PRU0_EVT ;       // clear interrupt
   TOGGLE_PRU_LED ;
   return ;
}

// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Routine to kill some time
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
void killTime(int32_t delay) {
```

```
   int i ;
   for (i=0; i<delay; i++) ;
   return ;
} // end killTime()


// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// MAIN PROGRAM STARTS HERE
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

void main() {

// Point to 12 kB of shared memory

   mem = (shared_memory_t *) PRU_SHARED_MEM_ADDR ;

// Perform some PRU initialization tasks

   initPRU() ;

// Enable the motor driver signals

   enableBuffers() ;

// Keep implementing commands until we are told to exit
// Put in some delay.
// Don't want to check the status too rapidly so we
// kill some time ...

   while (!mem->exitFlag) {
       killTime(KILL_TIME) ;
       switch (mem->command.status) {
           case  IDLE:      break ;

           case  START:     doCommand(mem->command.code) ;
                            break ;

           case  ACTIVE:    break ;

           case  COMPLETED: break ;

           case  ABORTED:   break ;

       } // end switch
    } // end while

   doCommand(HALT_PRU) ;

// Disable the motor driver signals

   disableBuffers() ;

// Turn the PRU LED off

   OFF_PRU_LED;

//   GPIO1pin(LED_PIN, OFF) ;

   __R31 = 35;    // PRU 0 to ARM interrupt
   __halt();      // halt the PRU

} // end main
```

```
//
// Defines used by PRU 0
//

// 70 ms for 1,000,000

#define   KILL_TIME   1000000

// For convenience

#define   TRUE       1
#define   FALSE      0

//#define PRU addresses

/*
#define PRU0
#define HOST1_MASK              (0x80000000)
#define HOST0_MASK              (0x40000000)
#define PRU0_PRU1_EVT           (16)
#define PRU1_PRU0_EVT           (18)
#define PRU0_ARM_EVT            (34)
#define SHARE_MEM               (0x00010000)
*/



// Bit 15 is P8-11
// Bit 14 is p8-16
// Bit 07 is p9-25
// Bit 05 is p9-27

#define TOGGLE_PRU_LED          (__R30 ^= (1 << 15))      //Bit 15
#define OFF_PRU_LED             (__R30 &= (0xFFFF7FFF))
#define ON_PRU_LED                      (__R30 |= (0x00008000))
#define DISABLE_DRV                     (__R30 |= (1 << 5))       //Bit 5 Neg logic
#define ENABLE_DRV                      (__R30 &= (0xFFFFFFDF))   //Bit 5 Neg logic
#define TOGGLE_DRV                      (__R30 &= (1 << 5))       //Bit 5 Neg logic
#define PRU_SW_VALUE            (__R31  & (1 << 14))      //Bit 14
#define ACC_IN1_VAL                     (__R31  & (1 << 7))           //Bit 7
```

```
//
// Library of routines which run on PRU0
// for handling the motors
//

#include <stdio.h>
#include <stdint.h>
#include <math.h>
#include "mem.h"
#include "fix.h"
#include "motorLib.h"

// Pointer to shared memory is a global variable

  extern   shared_memory_t   *mem ;

// Wait for interrupt

  void    waitForInterrupt(void) ;

// ************************************************************
// Routine to halt PRU #1
// Sets halt flag bit in state variable
// We also have to clear the run flag.
// ************************************************************

void  haltPRU(void) {
   mem->command.status = ACTIVE ;
   mem->state = M_HALT ;
   return ;
}

// ************************************************************
// Routine to apply hard brake
// ************************************************************

void hardBrake(void) {
   mem->command.status = ACTIVE ;
   mem->state = M_HARD_BRAKE ;
   return ;
}

// *************************************************************
// Routine to coast to a stop
// *************************************************************

void coast(void) {
   mem->command.status = ACTIVE ;
   mem->state = M_COAST ;
   return ;
}

// ***************************************************************
// Routine to implement PID loop on a single DC motor
// Using the velocity or differential PID
//
// delta_p = Kp * error
// delta_i = Ki * (error - past_error)
// delta_d = Kd * (error - past_error + 2 * past_past_error)
//
// output = previous_output + (delta_p + delta_i + delta_d)
//
// Errors in Q0 format.
```

```
// enc in Q0 format.
// Kp, Ki, Kd are in Q12 format,
// delta_p, delta_i, and delta_d are in Q format
// Output in Q0 format.
//
// ***************************************************************

int32_t PID(DCmotor_t * motor, int32_t enc) {
   int32_t   delta_p, delta_i, delta_d ;
   int32_t   scr ;
   int32_t   out ;

// Update past_error and past_past_error

   motor->e2 = motor->e1 ;
   motor->e1 = motor->e0 ;

// Compute new error term

   motor->e0 = FSUB(motor->setpoint, enc) ;

// Compute delta_p, delta_i, and delta_d

   delta_p = FMUL(motor->Kp, motor->e0, 0) ;
   scr = FSUB(motor->e0, motor->e1) ;
   delta_i = FMUL(motor->Ki, scr, 0) ;
   scr = FADD(scr, motor->e2 << 1) ;
   delta_d = FMUL(motor->Kd, scr, 0) ;
   scr = FADD(delta_i, delta_d) ;
   scr = FADD(scr, delta_p) ;

// Convert the delta from Q to 0 format

   scr = FCONV(delta_p, Q, 0) ;
   out = FADD(motor->PWMout, scr)  ;

// Make sure "out" is in range

   if (out > motor->PWMmax) {
       out = motor->PWMmax ;
   } else {
       if (out < motor->PWMmin) {
           out = motor->PWMmin ;
       } // end if
   } // end if-then-else

// Save the output and also return it

   motor->PWMout = out ;
   return   out ;
}

// *******************************************************************
// Routine to move.
// **********************************************************************

void move(void) {
   int        i ;
   DCmotor_t  *motor ;
   int32_t    state ;

// Status is ACTIVE
```

```
   mem->command.status = ACTIVE ;

// Set the errors to zero
// Also set the distance traveled to 0
// The setpoint, brake mode, wheel direction
// and target distance all get set by the routines
// in robotLib. Slso clear out pwm array.
// enc array is cleared in PRU 1 asm code

   for (i=0; i<NUM_MOTORS; i++) {
      mem->motor[i].e0 = 0 ;
      mem->motor[i].e1 = 0 ;
      mem->motor[i].e2 = 0 ;
      mem->motor[i].distance = 0 ;
      mem->motor[i].PWMout = 0 ;
      mem->pwm[i] = 0 ;
   }

// Look at direction field so we can set the state correctly
// Also look at the breaking mode so when we stop
// we do so either by braking hard or by coasting.

   state = 0 ;

   if (mem->motorENA[M1]) {
        if (mem->motor[M1].wheelDirection == CW) {
           state |= M1_CW ;
        } else {
           state |= M1_CCW ;
        } // end if-then-else
        if (mem->motor[M1].brakeType == HARD) {
           state |= M_HARD_BRAKE ;
        } // end if
   } // end if

   if (mem->motorENA[M2]) {
        if (mem->motor[M2].wheelDirection == CW) {
           state |= M2_CW ;
        } else {
           state |= M2_CCW ;
        } // end if-then-else
        if (mem->motor[M2].brakeType == HARD) {
           state |= M_HARD_BRAKE ;
        } // end if
   } // end if

   if (mem->motorENA[M3]) {
        if (mem->motor[M3].wheelDirection == CW) {
           state |= M3_CW ;
        } else {
           state |= M3_CCW ;
        } // end if-then-else
        if (mem->motor[M3].brakeType == HARD) {
           state |= M_HARD_BRAKE ;
        } // end if
   } // end if
   if (mem->motorENA[M4]) {
        if (mem->motor[M4].wheelDirection == CW) {
           state |= M4_CW ;
        } else {
           state |= M4_CCW ;
        } // end if-then-else
        if (mem->motor[M4].brakeType == HARD) {
```

```
           state |= M_HARD_BRAKE ;
        } // end if
   } // end if

// Set the run bit

   state |= M_RUN ;

// Write state out to shared memory
// PRU1 should start generating PWM outputs

   mem->state = state ;

// Main loop
// Keep looping until distance traveled on a single
// motor exceeds the target distance.
// PRU 1 will interrupt us at the desired sample rate.
// The waitForInterrupt routine toggles the PRU LED
// at the sample rate to provide user with visual feedback.
//

   int32_t  scr ;
   int   loop = TRUE ;
   while (loop) {
      waitForInterrupt() ;
      for (i=0; i < NUM_MOTORS; i++) {
         if (mem->motorENA[i]) {
            motor = &mem->motor[i] ;
            motor->distance = motor->distance + mem->enc[i] ;
            scr = motor->setpoint + motor->deltaSetpoint ;
            if (scr <= motor->targetSetpoint) {
                motor->setpoint = scr ;
            }
            if ((motor->distance) < (motor->targetDistance)) {
                mem->pwm[i] = PID(motor, mem->enc[i]) ;
            } else {
                loop = FALSE ;
                mem->state = M_STOP & mem->state ; // clear run bit
            } // end if-then-else
         } // end if
      } // end for
   } // end while

   return  ;

} // end move()

// *****************************************************************
//  Executes the command
//  The move() routine will look at the state and call appropriate
//  subroutine.  Upon return from the called routine, the status
//  will be updated to reflect that the commmand has COMPLETED.
//  The move() routine looks at the mem->motor struct to figure
//  out exactly what is must do.
// ****************************************************************

void doCommand(int32_t  command_code) {

   switch (command_code) {
       case  NOP:        mem->command.status = IDLE ;
                         break ;

       case  FWD:        move() ;
```

```
                         mem->command.status = COMPLETED ;
                         break ;

       case  BWD:        move() ;
                         mem->command.status = COMPLETED ;
                         break ;

       case  ROT:        move() ;
                         mem->command.status = COMPLETED ;
                         break ;

       case BRAKE_HARD:  hardBrake() ;
                         mem->command.status = COMPLETED ;
                         break ;

       case BRAKE_COAST: coast() ;
                         mem->command.status = COMPLETED ;
                         break ;

       case HALT_PRU:    haltPRU() ;
                         mem->command.status = COMPLETED ;
                         break ;

       default:          mem->command.status = IDLE ;
                         break ;

   } // end switch

  return ;

} // end doCommand()
```

```
// $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
// Define some useful subroutines
// $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$

#include <stdint.h>
#include "pru_cfg.h"
#include "pru_intc.h"
#include "pru0Lib.h"

// Global variables that allow us to handle GPIO

extern   int   *clrGPIO1_reg ;
extern   int   *setGPIO1_reg ;
extern   int   *readGPIO1_reg ;

extern   int   *clrGPIO3_reg ;
extern   int   *setGPIO3_reg ;
extern   int   *readGPIO3_reg ;


// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// GPIO 0 intialization
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
void initGPIO(void) {
   clrGPIO1_reg = (int *) (GPIO1 | GPIO_CLEARDATAOUT) ;
   setGPIO1_reg = (int *) (GPIO1 | GPIO_SETDATAOUT) ;
   readGPIO1_reg = (int *) (GPIO1 | GPIO_DATAIN) ;

   clrGPIO3_reg = (int *) (GPIO3 | GPIO_CLEARDATAOUT) ;
   setGPIO3_reg = (int *) (GPIO3 | GPIO_SETDATAOUT) ;
   readGPIO3_reg = (int *) (GPIO3 | GPIO_DATAIN) ;
   return ;
}

// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Subroutine to write to GPIO1 pin
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
void GPIO1pin(int pin, int value) {
   switch (value) {
      case OFF:   *clrGPIO1_reg = *readGPIO1_reg  | (1 << pin) ;
                  break ;
      case ON:    *setGPIO1_reg = *readGPIO1_reg |  (1 << pin) ;
                  break ;
   }
   return ;
}

// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Subroutine to write to GPIO3 pin
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
void GPIO3pin(int pin, int value) {
   switch (value) {
      case OFF:   *clrGPIO3_reg = *readGPIO3_reg  | (1 << pin) ;
                  break ;
      case ON:    *setGPIO3_reg = *readGPIO3_reg |  (1 << pin) ;
                  break ;
   }
   return ;
}

// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Routine to enable buffers
```

```
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

void enableBuffers(void) {
   GPIO3pin(DRV_PIN, 0) ;
   return ;
}

// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Routine to disable buffers
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

void disableBuffers(void) {
   GPIO3pin(DRV_PIN, 1) ;
   return ;
}

// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Subroutine slowly blink LED
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
void blinkLED(void) {
   static int LEDstate = OFF ;
   if (LEDstate == ON) {
       GPIO1pin(LED_PIN, OFF) ;
       LEDstate = OFF ;
   } else {
       GPIO1pin(LED_PIN, ON) ;
       LEDstate = ON ;
   } // end else

   return ;
} // blink LED
```

```
//
//  Defines
//

// Operating Modes

#define  LOOPBACK     1
#define  BROADCAST    2
#define  VAD          3
#define  SINE         4

// Fixed point operationss

#define  FADD(op1,op2)      ( (op1) + (op2) )
#define  FSUB(op1,op2)      ( (op1) - (op2) )
#define  FMUL(op1,op2,q)    ((int32_t) (((int64_t) ((int64_t) (op1) * (int64_t) (op2)))
 >> q))

// #define  FDIV(op1,op2,q)    ( (int32_t) (((int64_t)(op1) << q)/ ((int64_t) op2 )) )

// Convert from a q1 format to q2 format

#define  FCONV(op1,q1,q2)     (((q2) > (q1)) ? ((op1) << ((q2)-(q1))) : ((op1) >> ((q1)
-(q2))))

// Convert a float to a fixed-point representation in q format

#define  TOFIX(op1, q)       ((int32_t) ((op1) * ((float) (1 << (q)))))

// Convert a fixed-point number back to a float

#define  TOFLT(op1, q)       ( ((float) (op1)) / ((float) (1 << (q))) )

// Misc defines

#define PRU0
#define HOST1_MASK              (0x80000000)
#define HOST0_MASK              (0x40000000)

//#define PRU0_PRU1_EVT (16)

#define PRU1_PRU0_EVT           (18)

// Bit 3 is P9-28

#define TOGGLE_LED   (__R30 ^= (1 << 3))

// The magic pi

#define   PI     3.14159

// Fixed-point Q values

#define   Q14        14
#define   Q15        15
#define   Q28        28
#define   Q30        30
#define   Q20        20

// Shared memory address

#define   PRU_SHARED_MEM_ADDR     0x00010000
```

```
// GPIO bank addresses

#define      GPIO0                0x44e07000
#define      GPIO1                0x4804c000
#define      GPIO2                0x481ac000
#define      GPIO3                0x481ae000

// These can be OR’ed with the above bank addresses

#define      GPIO_DATAOUT         0x13C
#define      GPIO_DATAIN          0x138
#define      GPIO_CLEARDATAOUT    0x190
#define      GPIO_SETDATAOUT      0x194

#define      TP_PIN       27
#define      LED_PIN      26

#define      OFF          0
#define      ON           1
//
// Ears Library function prototype declarations
//

void       initGPIO(void) ;
void       GPIO0pin(int pin, int value) ;
void       blinkLED(void) ;

int16_t    *pwrap(uint32_t bufLen, int16_t * buf, int16_t * ptr) ;
// int16_t    fir(int M,  int16_t *buf,  int16_t *ptr, int Ntaps, int16_t *h) ;
void       initPRU(void) ;
int32_t    sineGen(osc_t *osc) ;
void       storeInput(void) ;
int16_t    iir_1(iir_1_t * filt) ;
void       updatePointers(void) ;
void       doLPF(void) ;
int32_t    measureEnergy(uint32_t M, int16_t *data, int32_t past_energy, int16_t *ptr,
uint32_t N) ;
void       doEnergy(void) ;

// Routine for the 4 operating modes

void doMode(int mode) ;
```

```
//
//  Defines
//

// Misc defines

#define PRU0
#define HOST1_MASK              (0x80000000)
#define HOST0_MASK              (0x40000000)

//#define PRU0_PRU1_EVT (16)

#define PRU1_PRU0_EVT           (18)

// Shared memory address

#define   PRU_SHARED_MEM_ADDR     0x00010000

// GPIO bank addresses

#define      GPIO0                0x44e07000
#define      GPIO1                0x4804c000
#define      GPIO2                0x481ac000
#define      GPIO3                0x481ae000

// These can be OR'ed with the above bank addresses

#define      GPIO_DATAOUT         0x13C
#define      GPIO_DATAIN          0x138
#define      GPIO_CLEARDATAOUT    0x190
#define      GPIO_SETDATAOUT      0x194

// GPIO LED GPIO1[12]

#define      LED_PIN      12

// GPIO LED GPIO3[19]

#define      DRV_PIN      19


#define      OFF          0
#define      ON           1

//
// Library function prototype declarations
//

void       initGPIO(void) ;
void       GPIO1pin(int pin, int value) ;
void       blinkLED(void) ;

void       GPIO3pin(int pin, int value) ;
void       enableBuffers(void) ;
void       disableBuffers(void) ;
```

```
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Defines
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

// BUF_LEN is length of data buffer
// STR_LEN is length of string buffer

#define     BUF_LEN         200
#define     STR_LEN         250

// Motor defines

#define     NUM_MOTORS      4

#define     M1              0
#define     M2              1
#define     M3              2
#define     M4              3

#define     CRASH           -9999

// Wheel directions

#define   CW      0
#define   CCW     1

// Brake types

#define   COAST   0
#define   HARD    1
//
// Commands we can give PRU 0
//

#define   NOP            0
#define   FWD            1
#define   BWD            2
#define   ROT            3
#define   BRAKE_HARD     4
#define   BRAKE_COAST    5
#define   HALT_PRU       6

//
// Here the codes for status of commands
//

#define   CMD            1
#define   STATUS         2

#define   IDLE          0
#define   START         1
#define   ACTIVE        2
#define   COMPLETED     3
#define   ABORTED       4

// These are used when we query the motor structure

#define   SETPOINT       1
#define   DISTANCE       2
#define   TARGET_DIST    3
#define   WHEEL_DIR      4
#define   BRAKE_TYPE     5
```

```
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// A structure that decribes a command from
// the ARM to PRU 0
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

typedef struct {
    int32_t     code ;
    int32_t     status ;
} command_t ;


// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Declare a structure to hold the GUI variables
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

typedef struct {
    int      exitFlag ;
    int      sonarEna ;
    int      lineEna ;
    int      rtcEna ;
    int      accelEna ;
    int      motorType ;
    float    Kp ;
    float    Ki ;
    float    Kd ;
    float    samplePeriod ;
    float    wheelDiam ;
    float    turnRad ;
    float    ticsPerRev ;
    int      M1Ena ;
    int      M2Ena ;
    int      M3Ena ;
    int      M4Ena ;
    int      PWMresMode ;
} GUIvars_t ;


// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// A DC motor structure
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

typedef struct {
    int32_t     setpoint ;          // desired velocity (in tics)
    int32_t     targetSetpoint ;    // will rammp up intil this is reached
    int32_t     deltaSetpoint ;     // steps we will take in ramping up
    int32_t     distance ;          // dist in tics (actual)
    int32_t     targetDistance ;    // dist in tics (desired)
    int32_t     wheelDirection ;    // CW or CCW
    int32_t     brakeType ;         // COAST or HARD
    int32_t     e0 ;                // current error
    int32_t     e1 ;                // past error
    int32_t     e2 ;                // past "past error"
    int32_t     Kp ;                // proportional gain (Q)
    int32_t     Ki ;                // integral gain (Q)
    int32_t     Kd ;                // deriviative gain (Q)
    int32_t     PWMmin ;            // minumum PWM out allowed
    int32_t     PWMmax ;            // maximum PWM out allowed
    int32_t     PWMout ;            // PWM output
}   DCmotor_t;

// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Our shared memory structure
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

```
typedef struct {
    int32_t     pwm[NUM_MOTORS] ;      // shared mem byte os of 0
    int32_t     enc[NUM_MOTORS] ;      // os of 16
    int32_t     delay ;                // os of 32
    int32_t     state ;                // os of 36
    int32_t     PWMclkCnt ;            // os of 40
    int32_t     PWMres ;               // os of 44
    int32_t     exitFlag ;             // exit when true
    int32_t     interruptCounter ;     // sample counter
    int32_t     motorType ;            // DC or stepper
    int32_t     motorENA[NUM_MOTORS] ; // Motor enables
    int32_t     scr ;                  // scratchpad register
    int32_t     wheelDiam ;            // diameter in inches (Q)
    int32_t     ticsPerInch;           // encoder tics per inch (Q)
    int32_t     enc_data[BUF_LEN] ;    // Buffer of encoder data
    command_t   command ;              // Motor command structure
    DCmotor_t   motor[NUM_MOTORS] ;    // DC motor structure
}   shared_memory_t ;
```

```
//
// Defines for PRU #1 assembly code
//

#define         PRU_R31_VEC_VALID         32          // allows notification of program
completion
#define         PRU_EVTOUT_1              4           // event number that is sent back
for PRU 1 to ARM interrupt
#define         PRU0_PRU1_INTERRUPT       17          // PRU0->PRU1 interrupt number
#define         PRU1_PRU0_INTERRUPT       18          // PRU1->PRU0 interrupt number
#define         ARM_PRU1_INTERRUPT        37              //  ARM->PRU1 interrupt number

// Shared memory base addres AND
// Byte offsets for shared memory accesses

#define         PRU_SHARED_MEM_ADDR    0x00010000
#define         PWM_OS          0
#define         ENC_OS          16
#define         DELAY_OS        32
#define         STATE_OS        36
#define         CLK_CNT_OS      40
#define         PWM_RES_OS      44

// Define linux space GPIO access

#define         GPIO0                   0x44e07000
#define         GPIO1                   0x4804C000
#define         GPIO2                   0x481ac000
#define         GPIO3                   0x481ae000

#define         GPIO_CLEARDATAOUT    0x190           //Clearing GPIO
#define         GPIO_SETDATAOUT      0x194           //Setting GPIO
#define         GPIO_DATAOUT         0x138           //reading GPIO

/* gle
#define        GPIO_DATAOUT         0x13C
#define        GPIO_DATAIN          0x138
#define        GPIO_CLEARDATAOUT    0x190
#define        GPIO_SETDATAOUT      0x194
*/

#define        GPIO1_15_MASK     0x80       //SWITCH
#define        GPIO1_12_MASK     0x10       //LED

// Motor control signals

#define         M1_0     r30.t2
#define         M1_1     r30.t3

#define         M2_0     r30.t4
#define         M2_1     r30.t5

#define         M3_0     r30.t0
#define         M3_1     r30.t1

#define         M4_0     r30.t6
#define         M4_1     r30.t7

// Define register aliases

#define         nopReg                          r0.b0

#define         enc1                            r1
```

```
#define         enc2                            r2
#define         enc3                            r3
#define         enc4                            r4

#define         pwm1                            r5
#define         pwm2                            r6
#define         pwm3                            r7
#define         pwm4                            r8

#define         encNEW                          r9
#define         encOLD                          r10
#define         encEDGE                         r11

#define     GPIO_LED_STATE      r12
#define     GPIO_BUTTON         r13
#define     GPIO_LED            r14

#define     read_gpio1          r15
#define     set_gpio1           r16
#define     clr_gpio1           r17

#define     pwmResReg           r18
#define         stateReg                        r19
#define         clkCntReg                       r20

#define         i                                   r21
#define         j                                   r22

//Can be used to temporally hold values if needed
// scratchpad register

#define         scr                                 r23

// Currently not using the dela value

#define         delayValue                      r24

// Shared memory base address

#define         sharedMem                       r25

// R29 is used for subroutine calls

#define         M1_ctrl                         stateReg.b0.t2
#define         M2_ctrl                         stateReg.b0.t4
#define         M3_ctrl                         stateReg.b0.t0
#define         M4_ctrl                         stateReg.b0.t6

#define     run_flag            stateReg.b1.t0
#define     brake_flag          stateReg.b1.t1
#define     update_flag         stateReg.b1.t2
#define     halt_flag           stateReg.b1.t3

#define         enc1_bit                        encEDGE.b1.t0
#define         enc2_bit                        encEDGE.b1.t1
#define         enc3_bit                        encEDGE.b1.t2
#define         enc4_bit                        encEDGE.b1.t3

// Use r29 for subroutine calls
// Since r30.w0 is our output port!!!!!
// Else we get very odd behavior!

.setcallreg   r29.w0
```

```
// Define Macros

// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// read_delay_value
//
// Description:
//      read in the delay loop value from sharedMemory (written by host)
//
// At a byte offset of 32
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro          read_delay_value
                        lbbo    delayValue, sharedMem, DELAY_OS, 4
.endm

//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// get_state
//
// Description:
//      update the status register value in pru1 Memory
//      The state value can be updated by pru0 and the ARM
//      system if necesary it is used to determine wheel direction
//      braking system, if a update of PWM values need to happen,
//      or if the system should stop
//
//      Status byte structure:  Wheel control:     b0
//                              Status Flags:      b1-b2
//                                      b1.t0 - run flag    (1 if in run mode)
//                                      b1.t1 - brake flag  (1 if in hard brake)
//                                      b1.t2 - update flag (1 if an update for pwm)
//                                      b1.t3 - halt flag  (1 if we want to halt pru)
//                                 Nop (stays zero):  b3
//
//      At a byte offset of 36
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro          get_state
                        lbbo    stateReg, sharedMem, STATE_OS, 4
.endm

//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// read_clk_cnt
//
// Description:
//   Tells us how many pwm clock cycles we should run
//   before interrupting PRU0
//
// At a byte offset of 40
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro          read_clk_cnt
                        lbbo    clkCntReg, sharedMem, CLK_CNT_OS, 4
.endm

//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// read_pwm_res
//
// Description:
//      read in the pwm maximum count from sharedMemory (written by host)
// Either 255 (8 bit), 1023 (10 bit), or 4095 (12 bit)
//
// At a byte offset of 44
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro          read_pwm_res
                        lbbo    pwmResReg, sharedMem, PWM_RES_OS, 4
```

```
.endm

// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// pwm_timer
//
// Description:
//      This decrements the PWM register given the motor
//      timer register and will stop the pwm signal if
//      necessary
//
// Usage:
//      pwm_timer       M1_ctrl, pwm1, M1_1, M1_1, NEXT_LINE
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro          pwm_timer
.mparam         M0_ctrl, pwm0, M0_0, M0_1

                   qbbc         CCW, M0_ctrl            //Check if we are clockwise or
counter clockwise
CW:                qbne         PWM_JMP, pwm0, 0        //Check if time to bring low
                   clr      M0_0                            //Set bit low
                   qba      NEXT                            //Jump to the next instruct
ion

CCW:       qbne     PWM_JMP, pwm0, 0    //counter clockwise case
                   clr      M0_1
                   qba      NEXT

PWM_JMP:   dec      pwm0                        //If it is not time to bring low decrem
ent
                   NO_OP                                        //NO_OP to mimic jump
NEXT:
.endm

// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// check_encoder_edges
//
// Description:
//      Transfers the old encoder values read the new ones
//      and xor to see if there is an edge
//
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro          check_encoder_edges
                        mov     encOLD.b1, encNEW.b1
                        mov     encNEW.b1, r31.b1
                        xor     encEDGE.b1, encNEW.b1, encOLD.b1
.endm

// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// zero_encoder_regs
//
// Description:
//   Clears the enc1, enc2, enc3, enc4 registers
//
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro          zero_encoder_regs
            zero &enc1, 16
.endm

// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// enc_cnt
//
// Description:
//      Reads the encoder tics and increments if need be
```

```
//
// Usage:
//      enc_cnt         enc1, enc1_bit
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro          enc_cnt
.mparam         enc0, enc0_bit
            qbbc        ENC_JMP, enc0_bit       //If clear jump to ENC_JMP
            inc     enc0                            //If there is an edge increment
                    qba     NEXT
ENC_JMP:        NO_OP
                    NO_OP
NEXT:
.endm

//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// store_encoder_values
//
// Description:
//      Stores the current encoder values to shared sharedMemory
//
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro          store_encoder_values
                        sbbo    &enc1, sharedMem, ENC_OS, 16
.endm

//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// pwm_start
//
// Description:
//      Uses the state register to start the pwm values
//      stored in b0
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro          pwm_start
                        mov     r30.b0, stateReg.b0
.endm

//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// brake
//
// Description:
//      Stops the pwm by setting outputs to zero or to one
//      depending on the brake bit
//
// Usage:
//      stop_pwm  NEXT_LINE
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro          brake
                        qbbs    HARD_BR, brake_flag
                        mov         r30.b0, 0x00
                        qba         NEXT
HARD_BR:        mov         r30.b0, 0xFF
                        NO_OP
NEXT:
.endm

//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// brake
//
// Description:
//      Hard brake
//
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro          hard_brake
```

```
                        mov         r30.b0, 0xff
.endm

//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// read_pwm_values
//
// Usage:
//      read in PWM values from sharedMemory
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro          read_pwm_values
                        lbbo    pwm1, sharedMem, PWM_OS, 16
.endm

// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// No operation
// (1 clock cycle)
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro          NO_OP
                        mov    nopReg, nopReg
.endm

//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Clear All registers (R0-R28)
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro      clear_REGS
                zero    &r0, 116
.endm

//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Turn the GPIO LED on (p8.12, GPIO1[12], GPIO:44)
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro          led_ON
            set    GPIO_LED_STATE, 12
            sbbo   GPIO_LED, set_gpio1, 0, 4
.endm

//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Turn the GPIO LED off (p8.12, GPIO1[12], GPIO:44)
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro          led_OFF
            clr     GPIO_LED_STATE, 12
            sbbo    GPIO_LED, clr_gpio1, 0, 4
.endm

//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Toggle the GPIO LED (p8.12, GPIO1[12], GPIO:44)
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro          led_TOGGLE
            xor      GPIO_LED_STATE.b1, GPIO_LED_STATE.b1, 1<<4
            qbbc     L0, GPIO_LED_STATE, 12
            led_ON
            qba      L1
L0:                     led_OFF
L1:
.endm

//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// QBPR : Quick branch if button press
//
// Usage:
//      qbpr LOCATION
//
// Branches to target location if GPIO button is pressed
```

```
//      (Green Button, p8.15, GPIO1[15], GPIO:47)
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro          qbpr
.mparam         LOCATION
                lbbo   GPIO_BUTTON, read_gpio1, 0, 4
            qbbs   LOCATION, GPIO_BUTTON.t15
.endm

// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// movi32 : Move a 32bit value to a register
//
// Usage:
//     movi32   dst, src
//
// Sets dst = src. Src must be a 32 bit immediate value.
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro          movi32
.mparam         dst, src
            mov     dst.w0, src & 0xFFFF
            mov     dst.w2, src >> 16
.endm

// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// dec: Decrement value/register
//
// Usage:
//      dec value
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro      dec
.mparam     value
            sub     value, value, #1
.endm

// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// inc: Increment value/register
//
// Usage:
//      inc value
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro      inc
.mparam         value
                        add     value,value, #1
.endm

// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Copy a bit from source register to the destination register
// dreg is the destination register
// dbit is the bit number in the destination register
// sreg is the source register
// sbit is the bit number in the source register
// (4 clock cycles)
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro      copy_bit
.mparam     dreg, dbit, sreg, sbit
            clr     dreg, dbit
            qbbc    END, sreg, sbit
            set     dreg, dbit
END:
.endm

//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Macro to set up the GPIO utils
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

```
.macro          gpio_SETUP
            mov     read_gpio1, GPIO1 | GPIO_DATAOUT
            mov     set_gpio1, GPIO1 | GPIO_SETDATAOUT
            mov     clr_gpio1, GPIO1 | GPIO_CLEARDATAOUT
            set     GPIO_LED, 12
.endm

//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// send_ARM_interrupt
//
// Description:
//      Send an interrupt to the arm from pru1
//
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro          send_ARM_interrupt
                        mov     r31.b0, PRU_R31_VEC_VALID | PRU_EVTOUT_1
.endm

//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// send_pru0_interrupt
//
// Description:
//      Send an interrupt to pru0 from pru1
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro          send_pru0_interrupt
                        ldi     r31, PRU1_PRU0_INTERRUPT + 16
.endm

// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Macro used to enable to OCP port
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.macro          ocp_port_ENABLE
            lbco    r0, C4, 4, 4     // load SYSCFG reg into r0 (use c4 const addr)
                clr     r0, 4            // clear bit 4 (STANDBY_INIT)
                sbco    r0, C4, 4, 4     // store the modified r0 back at the load addr
.endm
```

```
//
// Routines that make doing fixed point operations easy
//

#define     PI         3.14159

#define     Q          6
#define     twoQ       12

#define     Q24        24
#define     Q12        12
#define     Q0         0

// Fixed point operationss

#define  FADD(op1,op2)      ( (op1) + (op2) )
#define  FSUB(op1,op2)      ( (op1) - (op2) )
#define  FMUL(op1,op2,q)    ((int32_t) (((int64_t) ((int64_t) (op1) * (int64_t) (op2)))
 >> q))

// #define  FDIV(op1,op2,q)    ( (int32_t) (((int64_t)(op1) << q)/ ((int64_t) op2 )) )

// Convert from a q1 format to q2 format

#define  FCONV(op1,q1,q2)     (((q2) > (q1)) ? ((op1) << ((q2)-(q1))) : ((op1) >> ((q1)
-(q2))))

// Convert a float to a fixed-point representation in q format

#define  TOFIX(op1, q)       ((int32_t) ((op1) * ((float) (1 << (q)))))

// Convert a fixed-point number back to a float

#define  TOFLT(op1, q)       ( ((float) (op1)) / ((float) (1 << (q))) )
```

```
//
// Function prototype declarations
//
//
// Defines used by motorLib
//
#define   FALSE   0
#define   TRUE    1

#define   M_RUN                  (1 << 8)
#define   M_STOP         (~M_RUN)
#define   M_HARD_BRAKE   (1 << 9)
#define   M_UPDATE               (1 << 10)
#define   M_HALT                 (1 << 11)
#define   M_COAST                0

// Wheel directions

#define   CW      0
#define   CCW     1

// Motor control for state register

#define   M1_CW         (0x00000004)
#define   M1_CCW        (0x00000008)
#define   M2_CW         (0x00000010)
#define   M2_CCW        (0x00000020)
#define   M3_CW         (0x00000001)
#define   M3_CCW        (0x00000002)
#define   M4_CW         (0x00000040)
#define   M4_CCW        (0x00000080)

void      haltPRU(void) ;
void      hardBrake(void) ;
void      coast(void) ;
int32_t   PID(DCmotor_t *motor, int32_t enc) ;
void      move(void) ;
void      doCommand(int32_t  command_code) ;
```