```
//#define TEST

#include "ES_Configure.h"
#include "ES_Framework.h"
#include "ES_DeferRecall.h"
#include "TestHarnessService0.h"
#include "Hardware.h"

#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "inc/hw_gpio.h"
#include "inc/hw_sysctl.h"
#include "inc/hw_nvic.h"
#include "inc/hw_uart.h"
#include "driverlib/sysctl.h"
#include "driverlib/pin_map.h"      // Define PART_TM4C123GH6PM in project
#include "driverlib/gpio.h"
#include "driverlib/uart.h"
#include "ES_ShortTimer.h"
#include "driverlib/pwm.h"
#include "inc/hw_pwm.h"
#include "inc/hw_i2c.h"
#include "inc/hw_nvic.h"
#include "termio.h"

#define clrScrn()       printf("\x1b[2J")
#define goHome()  printf("\x1b[1,1H")
#define clrLine() printf("\x1b[K")

#include "ADMulti.h"
#include "Constants.h"

static uint8_t LastDirThrust = FORWARD;
static uint8_t LastDirDiscoBall = FORWARD;

static void IO_Init(void);
static void AD_Init(void);
static void PWM_Init(void);
static void UART_Init(void);
static void UART_PIC_Init(void);
static void I2C_Init(void);
static void UART_PIC_Init(void);

void Hardware_Init(void)
{
      IO_Init();
      AD_Init();
      PWM_Init();
      UART_Init();
      I2C_Init();
      UART_PIC_Init();


}

static void IO_Init(void)
{
      // connect clock to ports B
      HWREG(SYSCTL_RCGCGPIO) |= (SYSCTL_RCGCGPIO_R1);
      // wait for clock to connect to ports B and F
      while ((HWREG(SYSCTL_PRGPIO) & (SYSCTL_PRGPIO_R1)) != (SYSCTL_PRGPIO_R1)) {}
      // digitally enable IO pins
      HWREG(GPIO_PORTB_BASE + GPIO_O_DEN) |= (THRUST_FAN_DIR_B);
      // set direction of IO pins
```

```
	HWREG(GPIO_PORTB_BASE + GPIO_O_DIR) |= (THRUST_FAN_DIR_B);

}

static void AD_Init(void)
{
	// Connect clock to port E
	HWREG(SYSCTL_RCGCGPIO)|=SYSCTL_RCGCGPIO_R4;
	// wait for clock to connect to port E
	while((HWREG(SYSCTL_PRGPIO)& SYSCTL_PRGPIO_R4)!=SYSCTL_PRGPIO_R4){}
	// digitally enable Anaolog Pins (I realize this doesn't make any sense, it's
2 am leave me alone)
	HWREG(GPIO_PORTE_BASE + GPIO_O_DEN) |= (DOG_TAG_E);
	// set direction of Analog Pins
	HWREG(GPIO_PORTE_BASE + GPIO_O_DIR) &= ~(DOG_TAG_E);
	ADC_MultiInit(NUMBER_OF_ANALOG_PINS);
}

static void PWM_Init(void)
{

	// Enable the clock to the PWM Module
	HWREG(SYSCTL_RCGCPWM) |= (SYSCTL_RCGCPWM_R0);
	while ((HWREG(SYSCTL_PRPWM) & (SYSCTL_PRPWM_R0)) != (SYSCTL_PRPWM_R0)) {}
	// Enable the clock to Port B and F
	HWREG(SYSCTL_RCGCGPIO) |= (SYSCTL_RCGCGPIO_R1);
	while ((HWREG(SYSCTL_PRGPIO) & (SYSCTL_PRGPIO_R1)) != (SYSCTL_PRGPIO_R1)) {}
	// digitially enable the PWM pins
	HWREG(GPIO_PORTB_BASE+GPIO_O_DEN) |= (LEFT_SERVO_PIN_B | RIGHT_SERVO_PIN_B |
INDICATOR_PIN_B | THRUST_FAN_PWM_PIN_B);
	HWREG(GPIO_PORTB_BASE+GPIO_O_DIR) |= (LEFT_SERVO_PIN_B | RIGHT_SERVO_PIN_B |
INDICATOR_PIN_B | THRUST_FAN_PWM_PIN_B);
	// Select the system clock/32
	HWREG(SYSCTL_RCC) = (HWREG(SYSCTL_RCC) & ~SYSCTL_RCC_PWMDIV_M) |
(SYSCTL_RCC_USEPWMDIV | SYSCTL_RCC_PWMDIV_32);
	// Disable the PWM generator while initializing
	HWREG(PWM0_BASE + PWM_O_0_CTL) = 0;
	HWREG(PWM0_BASE + PWM_O_1_CTL) = 0;
	// Set initial generator values: motors should be stopped, servos at idle
	HWREG(PWM0_BASE + PWM_O_0_GENA) = PWM_0_GENA_ACTZERO_ZERO;
	HWREG(PWM0_BASE + PWM_O_0_GENB) = PWM_0_GENB_ACTZERO_ZERO;
	HWREG(PWM0_BASE + PWM_O_1_GENA) = GenA_1_Normal;
	HWREG(PWM0_BASE + PWM_O_1_GENB) = GenB_1_Normal;
	// Set the load to ½ the desired period since going up and down
	HWREG(PWM0_BASE + PWM_O_0_LOAD) = ((MOTOR_PWM_PERIOD) >> 1);
	HWREG(PWM0_BASE + PWM_O_1_LOAD) = ((SERVO_PWM_PERIOD) >> 1);
	// Set the initial duty cycle on the servos
	HWREG(PWM0_BASE + PWM_O_1_CMPA) = LEFT_SERVO_IDLE_DUTY;
	HWREG(PWM0_BASE + PWM_O_1_CMPB) = RIGHT_SERVO_IDLE_DUTY;
	// Enable the PWM outputs
	HWREG(PWM0_BASE + PWM_O_ENABLE) |= (PWM_ENABLE_PWM0EN | PWM_ENABLE_PWM1EN |
PWM_ENABLE_PWM2EN | PWM_ENABLE_PWM3EN);
	// Select the alternate function for PWM Pins
	HWREG(GPIO_PORTB_BASE + GPIO_O_AFSEL) |= (LEFT_SERVO_PIN_B |
RIGHT_SERVO_PIN_B | INDICATOR_PIN_B | THRUST_FAN_PWM_PIN_B);
	// Choose to map PWM to those pins
	HWREG(GPIO_PORTB_BASE + GPIO_O_PCTL) = (HWREG(GPIO_PORTB_BASE + GPIO_O_PCTL)
& PWM_PIN_M_B) + (4<<(LEFT_SERVO_BIT*BitsPerNibble)) +
(4<<(RIGHT_SERVO_BIT*BitsPerNibble)) + (4<<(INDICATOR_BIT*BitsPerNibble)) +
(4<<(THRUST_FAN_PWM_BIT*BitsPerNibble));
	// Set the up/down count mode
	// Enable the PWM generator
	// Make generator updates locally synchronized to zero count
	HWREG(PWM0_BASE + PWM_O_0_CTL) = (PWM_0_CTL_MODE | PWM_0_CTL_ENABLE |
```

```
PWM_0_CTL_GENAUPD_LS | PWM_0_CTL_GENBUPD_LS);
	HWREG(PWM0_BASE + PWM_O_1_CTL) = (PWM_1_CTL_MODE | PWM_1_CTL_ENABLE |
PWM_1_CTL_GENAUPD_LS | PWM_0_CTL_GENBUPD_LS);


	SetLeftBrakePosition(LEFT_SERVO_UP);
	SetRightBrakePosition(RIGHT_SERVO_UP);
}

static void UART_Init(void)
{
	//Enable the clock for the UART module
	HWREG(SYSCTL_RCGCUART) |= SYSCTL_RCGCUART_R1;

	//Wait for the UART to be ready
	while((HWREG(SYSCTL_PRUART)& SYSCTL_PRUART_R1)!=SYSCTL_PRUART_R1){}

	//Enable the clock to the appropriate gpio module via the RCGCGPIO - port C
	HWREG(SYSCTL_RCGCGPIO) |= SYSCTL_RCGCGPIO_R2;

	//Wait for the GPIO module to be ready
	while((HWREG(SYSCTL_PRGPIO)& SYSCTL_PRGPIO_R2)!=SYSCTL_PRGPIO_R2){}

	//Configure the GPIO pins for in/out/drive-level/drive-type
	HWREG(GPIO_PORTC_BASE+GPIO_O_DEN) |= (GPIO_PIN_4 | GPIO_PIN_5);
	HWREG(GPIO_PORTC_BASE+GPIO_O_DIR) |= GPIO_PIN_5;
	HWREG(GPIO_PORTC_BASE+GPIO_O_DIR) &= ~GPIO_PIN_4;

	//Select the Alternate function for the UART pins
	HWREG(GPIO_PORTC_BASE+GPIO_O_AFSEL) |= (BIT4HI | BIT5HI);

	//Configure the PMCn fields in the GPIOPCTL register to assign the UART pins
	HWREG(GPIO_PORTC_BASE+GPIO_O_PCTL) = (HWREG(GPIO_PORTC_BASE+GPIO_O_PCTL) &
0Xfff0ffff) + (RX_ALT_FUNC<<(RX_PIN*BITS_PER_NIBBLE));
	HWREG(GPIO_PORTC_BASE+GPIO_O_PCTL) = (HWREG(GPIO_PORTC_BASE+GPIO_O_PCTL) &
0Xff0fffff) + (TX_ALT_FUNC<<(TX_PIN*BITS_PER_NIBBLE));

	//Disable the UART by clearing the UARTEN bit in the UARTCTL register
	HWREG(UART1_BASE+UART_O_CTL) = HWREG(UART1_BASE + UART_O_CTL) &
~UART_CTL_UARTEN;

	//Write the integer portion of the BRD
	HWREG(UART1_BASE + UART_O_IBRD) = BAUD_RATE_INT;

	//Write the fraction portion of the BRD
	HWREG(UART1_BASE + UART_O_FBRD) = BAUD_RATE_FRAC;

	//Write the desired serial parameters
	HWREG(UART1_BASE + UART_O_LCRH) = HWREG(UART1_BASE + UART_O_LCRH) |
UART_LCRH_WLEN_8;

	//Enable RX and TX interrupts in mask
	HWREG(UART1_BASE + UART_O_IM) = HWREG(UART1_BASE + UART_O_IM) | UART_IM_RXIM;

	//Configure the UART operation
	//Enable the UART
	HWREG(UART1_BASE + UART_O_CTL) = HWREG(UART1_BASE + UART_O_CTL) |
UART_CTL_UARTEN;

	//Enable interrupt in the NVIC
	HWREG(NVIC_EN0) |= BIT6HI;

	//Enable interrupts globally
	__enable_irq();
```

```
	//Print successful initialization
	printf("UART 1 Successfully Initialized! :)\r\n");
}

static void UART_PIC_Init(void)
{
	//Enable the clock for the UART module
	HWREG(SYSCTL_RCGCUART) |= SYSCTL_RCGCUART_R3;

	//Wait for the UART to be ready
	while((HWREG(SYSCTL_PRUART)& SYSCTL_PRUART_R3)!=SYSCTL_PRUART_R3){}

	//Enable the clock to the appropriate gpio module via the RCGCGPIO - port C
	HWREG(SYSCTL_RCGCGPIO) |= SYSCTL_RCGCGPIO_R2;

	//Wait for the GPIO module to be ready
	while((HWREG(SYSCTL_PRGPIO)& SYSCTL_PRGPIO_R2)!=SYSCTL_PRGPIO_R2){}

	//Configure the GPIO pins for in/out/drive-level/drive-type
	HWREG(GPIO_PORTC_BASE+GPIO_O_DEN) |= GPIO_PIN_7;
	HWREG(GPIO_PORTC_BASE+GPIO_O_DIR) |= GPIO_PIN_7;

	//Select the Alternate function for the UART pins
	HWREG(GPIO_PORTC_BASE+GPIO_O_AFSEL) |= BIT7HI;

	//Configure the PMCn fields in the GPIOPCTL register to assign the UART pins
	HWREG(GPIO_PORTC_BASE+GPIO_O_PCTL) = (HWREG(GPIO_PORTC_BASE+GPIO_O_PCTL) &
0X0fffffff) + (TX_PIC_ALT_FUNC<<(TX_PIC_PIN*BITS_PER_NIBBLE));

	//Disable the UART by clearing the UARTEN bit in the UARTCTL register
	HWREG(UART3_BASE+UART_O_CTL) = HWREG(UART3_BASE + UART_O_CTL) &
~UART_CTL_UARTEN;

	//Write the integer portion of the BRD
	HWREG(UART3_BASE + UART_O_IBRD) = PIC_BAUD_RATE_INT;

	//Write the fraction portion of the BRD
	HWREG(UART3_BASE + UART_O_FBRD) = PIC_BAUD_RATE_FRAC;

	//Write the desired serial parameters
	HWREG(UART3_BASE + UART_O_LCRH) = HWREG(UART3_BASE + UART_O_LCRH) |
UART_LCRH_WLEN_8;

	//Configure the UART operation
	//Enable the UART
	HWREG(UART3_BASE + UART_O_CTL) = HWREG(UART3_BASE + UART_O_CTL) |
UART_CTL_UARTEN;

	//Print successful initialization
	printf("UART PIC Successfully Initialized! :)\r\n");
}

void SetThrustFan(uint8_t DriveCtrl)
{
	uint8_t DutyCycle;


	if(DriveCtrl < 127) //If we are less than 127, we are going in reverse
	{
		//set the direction to reverse
		printf("REVERSE\r\n");
		SetDirectionThrust(REVERSE);
```

```
            //scale the ctrl value to be between 0 and 100 (where 127 corresponds
to 0, and 0 corresponds to 100 duty)
            DutyCycle = ((126 - DriveCtrl)*100)/126;
            printf("Thrust Fan DutyCycle = %i \r\n", DutyCycle);
            //write the value to the fan
            SetDutyThrustFan(DutyCycle);
      }
      else if((DriveCtrl >= 127) && (DriveCtrl <= 255)) //If we are greater than 127,
we are going forward
      {
            //set the direction to forward
            printf("FORWARD\r\n");
            SetDirectionThrust(FORWARD);

            //scale the ctrl value to be between 0 and 100 (where 127 corresponds
to 0, and 255 corresponds to 100 duty)
            DutyCycle = ((DriveCtrl-127)*100)/128;
            printf("Thrust Fan DutyCycle = %i \r\n", DutyCycle);

            //write the value to the fan
            SetDutyThrustFan(DutyCycle);
      }
      else
      {
            printf("HARDWARE ---- UNEXPECTED DUTY = %i \r\n", DriveCtrl);
      }

}

void SetDutyThrustFan(uint8_t duty)
{

      // Motor starts at rest
      static bool restoreMotor = true;

      // New Value for comparator to set duty cycle
      static uint32_t newCmp;
      if (LastDirThrust == REVERSE) duty = 100 - duty;
      // set new comparator value based on duty cycle
      newCmp = HWREG(PWM0_BASE + PWM_O_0_LOAD)*(100-duty)/100;
      if (duty == 100 | duty == 0)
      {
            restoreMotor = true;
            if (duty == 100)
            {
                  // To program 100% DC, simply set the action on Zero to set the
output to one
                  HWREG( PWM0_BASE+PWM_O_0_GENA) = PWM_0_GENA_ACTZERO_ONE;
            }
            else
            {
                  // To program 0% DC, simply set the action on Zero to set the output
to zero
                  HWREG( PWM0_BASE+PWM_O_0_GENA) = PWM_0_GENA_ACTZERO_ZERO;
            }
      }
      else
      {
            // if returning from 0 or 100
            if (restoreMotor)
            {
                  restoreMotor = false;
                  // restore normal operation
                  HWREG(PWM0_BASE+PWM_O_0_GENA) = GenA_0_Normal;
```

```
		}
		// write new comparator value to register
		HWREG( PWM0_BASE+PWM_O_0_CMPA) = newCmp;
	}

}

void SetDutyIndicator(uint8_t duty)
{

	// Motor starts at rest
	static bool restoreIndicator = true;

	// New Value for comparator to set duty cycle
	static uint32_t newCmp;
	// set new comparator value based on duty cycle
	if(LastDirDiscoBall == REVERSE)
	{
		duty = 100 - duty;
	}
	newCmp = HWREG(PWM0_BASE + PWM_O_0_LOAD)*(100-duty)/100;
	if (duty == 100 | duty == 0)
	{
		restoreIndicator = true;
		if (duty == 100)
		{
			// To program 100% DC, simply set the action on Zero to set the
output to one
			HWREG(PWM0_BASE+PWM_O_0_GENB) = PWM_0_GENB_ACTZERO_ONE;
		}
		else
		{
			// To program 0% DC, simply set the action on Zero to set the output
to zero
			HWREG(PWM0_BASE+PWM_O_0_GENB) = PWM_0_GENB_ACTZERO_ZERO;
		}
	}
	else
	{
		// if returning from 0 or 100
		if (restoreIndicator)
		{
			restoreIndicator = false;
			// restore normal operation
			HWREG(PWM0_BASE + PWM_O_0_GENB) = GenB_0_Normal;
		}
		// write new comparator value to register
		HWREG(PWM0_BASE + PWM_O_0_CMPB) = newCmp;
	}

}

void SetDirectionThrust(uint8_t dir)
{
	//THIS CODE APPEARS TO BE RIGHT, BUT STILL BREAKS WHEN WRITING DUTY?
	if (dir==REVERSE) {
		HWREG(PWM0_BASE + PWM_O_0_GENA) = GenA_0_Invert;
		HWREG(GPIO_PORTB_BASE + (GPIO_O_DATA + ALL_BITS)) |=
(THRUST_FAN_DIR_B);
	}
	else if (dir==FORWARD) {
		HWREG(PWM0_BASE + PWM_O_0_GENA) = GenA_0_Normal;
		HWREG(GPIO_PORTB_BASE + (GPIO_O_DATA + ALL_BITS)) &=
```

```
~(THRUST_FAN_DIR_B);
	}
	LastDirThrust=dir;

}

void SetDirectionDiscoBall(uint8_t dir)
{
	//THIS CODE APPEARS TO BE RIGHT, BUT STILL BREAKS WHEN WRITING DUTY?
	if (dir==REVERSE) {
		HWREG(PWM0_BASE + PWM_O_0_GENB) = GenB_0_Invert;
		HWREG(GPIO_PORTB_BASE + (GPIO_O_DATA + ALL_BITS)) |= (GPIO_PIN_1);
	}
	else if (dir==FORWARD) {
		HWREG(PWM0_BASE + PWM_O_0_GENB) = GenB_0_Normal;
		HWREG(GPIO_PORTB_BASE + (GPIO_O_DATA + ALL_BITS)) &= ~(GPIO_PIN_1);
	}
	LastDirThrust=dir;

}

void SetLeftBrakePosition(uint16_t position)
{

	// New Value for comparator to set duty cycle
	// max is 1600, min 300
	uint32_t newCmp = HWREG(PWM0_BASE+PWM_O_1_LOAD)*(12500-position)/12500;
	// write new comparator value to register
	HWREG(PWM0_BASE+PWM_O_1_CMPA) = newCmp;

}

void SetRightBrakePosition(uint16_t position)
{
	// max is 1600, min is 300
	// New Value for comparator to set duty cycle
	uint32_t newCmp = HWREG(PWM0_BASE+PWM_O_1_LOAD)*(12500-position)/12500;
	// write new comparator value to register
	HWREG(PWM0_BASE+PWM_O_1_CMPB) = newCmp;

}

uint8_t ReadDOGTag(void)
{
	uint32_t TagVal[1];
	ADC_MultiRead(TagVal);
	if (TagVal[0] < DOG_3_THRESHOLD)
	{
		return DOG_3;
	}
	else if (TagVal[0] < DOG_2_THRESHOLD)
	{
		return DOG_2;
	}
	else
	{
		return DOG_1;
	}
}

static void I2C_Init(void)
{
	// enable the I2C clock for I2C module 2
	HWREG(SYSCTL_RCGCI2C) |= SYSCTL_RCGCI2C_R2;
```

```
	while ((HWREG(SYSCTL_PRI2C) & SYSCTL_PRI2C_R2) != SYSCTL_PRI2C_R2) {}
	// enable clock to GPIO pins on I2C2 (E4, E5)
	HWREG(SYSCTL_RCGCGPIO) |= SYSCTL_RCGCGPIO_R4;
	while ((HWREG(SYSCTL_PRGPIO) & SYSCTL_PRGPIO_R4) != SYSCTL_PRGPIO_R4) {}
	//enable internal pullups
	HWREG(GPIO_PORTE_BASE + GPIO_O_PUR) |= (I2C_SDA_PIN | I2C_SCL_PIN);
	// digitally enable maybe?
	HWREG(GPIO_PORTE_BASE + GPIO_O_DEN) |= (I2C_SDA_PIN | I2C_SCL_PIN);
	// select alternate functions for B2, B3
	HWREG(GPIO_PORTE_BASE + GPIO_O_AFSEL) |= (I2C_SDA_PIN | I2C_SCL_PIN);
	// set SDA to Open Drain
	HWREG(GPIO_PORTE_BASE + GPIO_O_ODR) |= I2C_SDA_PIN;
	// select I2C function
	HWREG(GPIO_PORTE_BASE + GPIO_O_PCTL) = ((HWREG(GPIO_PORTE_BASE + GPIO_O_PCTL)
& I2C_PIN_M) | ((3 << (I2C_SDA_BIT*BitsPerNibble)) | (3 <<
(I2C_SCL_BIT*BitsPerNibble))));
	// initialize the TIVA as Master
	HWREG(I2C2_BASE + I2C_O_MCR) |= I2C_MCR_MFE;
	// set the SCL clock (there is a fancy equation, I'm just using the provided
10KBPS val given)
	HWREG(I2C2_BASE + I2C_O_MTPR) = ((HWREG(I2C2_BASE + I2C_O_MTPR) &
~(I2C_MTPR_TPR_M)) | I2C_COMM_SPEED);
	// Load Slave address
	HWREG(I2C2_BASE + I2C_O_MSA) = IMU_SLAVE_ADDRESS;
	// set up ISR
	HWREG(NVIC_EN2) |= BIT4HI;
	HWREG(I2C2_BASE + I2C_O_MIMR) |= I2C_MIMR_IM;
}


void sendToPIC(uint8_t value)
{
	printf("Sent To PIC: %i\r\n",value);
	if((HWREG(UART3_BASE+UART_O_FR) & UART_FR_TXFE) != 0)
	{
		if(value > 25)
		{
			//PIC expects value 0 to 25, if higher value gets sent then saturate
the rails
			HWREG(UART3_BASE+UART_O_DR) = 25;
		}
		else
		{
			HWREG(UART3_BASE+UART_O_DR) = value;
		}
	}
}


#ifdef TEST

int main(void)
{
	SysCtlClockSet(SYSCTL_SYSDIV_5 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN
			| SYSCTL_XTAL_16MHZ);
	TERMIO_Init();
	clrScrn();

	Hardware_Init();
	//SetLeftBrakePosition(1000);
	uint8_t Dog = ReadDOGTag();
	printf("\r\nDOG Tag: %d\r\n", Dog);
	while(1)
	{
```

```
//          uint32_t TagVal[1];
//          ADC_MultiRead(TagVal);
//          printf("\r\n%d\r\n", TagVal[0]);
      }

      return 0;
}

#endif
```