//#############################################################################

//

// FILE: cm\_common\_config\_c28x.c

//

// TITLE: C28x Common Configurations to be used for the CM Side.

//

//! \addtogroup driver\_example\_list

//! <h1>C28x Common Configurations</h1>

//!

//! This example configures the GPIOs and Allocates the shared peripherals

//! according to the defines selected by the users.

//!

//

//#############################################################################

// $TI Release: F2838x Support Library v3.04.00.00 $

// $Release Date: Fri Feb 12 19:08:49 IST 2021 $

// $Copyright:

// Copyright (C) 2021 Texas Instruments Incorporated - http://www.ti.com/

//

// Redistribution and use in source and binary forms, with or without

// modification, are permitted provided that the following conditions

// are met:

//

// Redistributions of source code must retain the above copyright

// notice, this list of conditions and the following disclaimer.

//

// Redistributions in binary form must reproduce the above copyright

// notice, this list of conditions and the following disclaimer in the

// documentation and/or other materials provided with the

// distribution.

//

// Neither the name of Texas Instruments Incorporated nor the names of

// its contributors may be used to endorse or promote products derived

// from this software without specific prior written permission.

//

// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS

// "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT

// LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR

// A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT

// OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,

// SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT

// LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,

// DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY

// THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT

// (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE

// OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

// $

//#############################################################################

//

// Included Files

//

**#include** "driverlib.h"

**#include** "device.h"

**#ifdef** \_\_TMS320C28XX\_CLA\_\_

**#pragma** CODE\_SECTION(PI\_init,"Cla1Prog2");

**#endif**

//#########################edit at 0627

**#define** IPC\_CMD\_READ\_MEM 0x1001

**#define** IPC\_CMD\_RESP 0x2001

**#define** TEST\_PASS 0x5555

**#define** TEST\_FAIL 0xAAAA

**#pragma** DATA\_SECTION(readData, "MSGRAM\_CPU\_TO\_CM")

**float** readData[4];

uint32\_t pass;

**float** recData\_c28[10];

**int** count = 0;

//

//adc

//

////

// Defines

//

**#define** EX\_ADC\_RESOLUTION 12

// 12 for 12-bit conversion resolution, which supports single-ended signaling

// Or 16 for 16-bit conversion resolution, which supports single-ended or

// differential signaling

**#define** EX\_ADC\_SIGNALMODE "SINGLE-ENDED"

//"SINGLE-ENDED" for ADC\_MODE\_SINGLE\_ENDED:

// Sample on single pin (VREFLO is the low reference)

// Or "Differential" for ADC\_MODE\_DIFFERENTIAL:

// Sample on pair of pins (difference between pins is converted, subject to

// common mode voltage requirements; see the device data manual)

//

// Globals

//

uint16\_t adcAResult0;

uint16\_t adcAResult1;

uint16\_t adcAResult2;

uint16\_t adcCResult0;

uint16\_t adcCResult1;

uint16\_t adcCResult2;

uint16\_t adcFinalRe;

uint16\_t adcRe[3];

uint16\_t adcFinalReT;

uint16\_t adcReT[3];

uint16\_t adcFinalReT1;

uint16\_t adcReT1[3];

uint16\_t adcFinalReT2;

uint16\_t adcReT2[3];

uint16\_t dt2;

//iiR

uint16\_t adc\_iiR;

uint16\_t iiR[2];

uint16\_t dif;

**float** F;

**int** fb = 200;

**float** k;

**float** k1=5;

**float** k2=0.001;

**float** adCR=100000;

**float** adCC=0.1;

**float** adCL=0.02;

**int** iiR\_T =0;

**int** iiR\_T2 =0;

//iiR

**float** B0=1;

**float** B1=2;

**float** B2=1;

**float** A0=1;

**float** A1=-1.521690426072245916344627403304912149906;

**float** A2= 0.600000000000000199840144432528177276254;

**float** gain = 0.019577393481938570873879257305816281587 ;

**float** wx0=0;

**float** wx1=0;

**float** wx2=0;

**float** wy0=0;

**float** wy1=0;

**float** wy2=0;

//

// Function Prototypes

//

**void** **configureADC**(uint32\_t adcBase);

**void** **initEPWM7**();

**void** **initADCSOC**(**void**);

**\_\_interrupt** **void** **adcA1ISR**(**void**);

//

// Defines

//

**#define** TB\_CLK DEVICE\_SYSCLK\_FREQ / 2 // Time base clock is SYSCLK / 2

**#define** PWM\_CLK 5e3 // We want to output at 5 kHz

// (300 rpm)

**#define** PRD\_VAL (TB\_CLK / (PWM\_CLK \* 2)) // Calculate value period value

// for up-down count mode

**#define** EPWM\_TIMER\_TBPRD 500UL

**#define** CPU\_CLK 200e6

**#define** PERIOD CPU\_CLK/PWM\_CLK

//fncs

**void** **initEPWMGPIO**(**void**);

**\_\_interrupt** **void** **IPC\_ISR1**();

//enet\_timer

**\_\_interrupt** **void** **cpuTimer1ISR**(**void**);

//cpu timer

**\_\_interrupt** **void** **cpuTimer0ISR**(**void**);

**\_\_interrupt** **void** **cpuTimer2ISR**(**void**);

**void** **initCPUTimers**(**void**);

**void** **configCPUTimer**(uint32\_t cpuTimer, uint32\_t period);

**void** **updateLimitS**(**void**);

**void** **CmdToDuty**(**void**);

**void** **initEPWM**(uint32\_t base);

//mov\_cmd list

**float** x\_rj = 0;

**float** y\_rj = 0;

**float** z\_rj = 0;

**float** up\_cmd = 0;

**float** down\_cmd = 0;

**float** left\_cmd = 0;

**float** right\_cmd = 0;

**float** front\_cmd;

**float** back\_cmd;

**float** spin\_cmd;

**float** rcm\_en\_cmd;

**float** ee\_en\_cmd;

**float** STOP\_cmd;

//call interrupt to turn on the motor

**int** on\_up\_motor;

**int** on\_down\_motor;

**int** on\_right\_motor;

**int** on\_left\_motor;

**const** **float** steady\_joystick = 20000;

**const** **float** max\_allow\_joystick = 5000;

**const** **float** min\_allow\_joystick = 500;

**const** **float** max\_allow\_Hz = 50;

**const** **float** min\_allow\_Hz = 20;

**const** **float** min\_step\_angle = 0.72;

**const** **float** timer0\_period = 0.0001;

**float** up\_rm;

**float** down\_rm;

**float** left\_rm;

**float** right\_rm;

**float** up\_Hz;

**float** down\_Hz;

**float** left\_Hz;

**float** right\_Hz;

**int** lr\_interrupt\_times = 0;

**int** ud\_interrupt\_times = 0;

//

**float** freq = 0;

**int** freq1 = 0;

**float** duty = 0;

//

**const** **float** Xdo = 5;

**float** Xd1 = 0;

**const** **float** K = -1;

**const** **float** tt = 0.1;

**float** preloc = 0;

**float** vv;

**float** delX = 0;

**const** **float** F1 = 1000;

**float** vsc;

**float** curloc=0;

**float** oral\_dpt = 0;

**int** dpt = 0;

//limit switch state

**float** limitUp = 0;

**float** limitDown = 0;

**float** limitLeft = 0;

**float** limitRight = 0;

uint16\_t cpuTimer0IntCount;

uint16\_t cpuTimer1IntCount;

uint16\_t cpuTimer2IntCount;

//for\_test

**int** tf=0;

**int** test0 = 0;

**int** test1 = 0;

**int** test2 = 0;

**int** test3 = 0;

**int** test4 = 0;

**int** test5 = 0;

**int** test6 = 0;

**int** test7 = 0;

**int** test8 = 0;

**int** test9 = 0;

**int** test10 = 0;

**int** test11 = 0;

**int** test12 = 0;

**int** test13 = 0;

**int** test14 = 0;

**int** qqq=0;

**void** **main**(**void**)

{

//

// Initialize device clock and peripherals

//

Device\_init();

//

// Boot CM core

//

**#ifdef** \_FLASH

Device\_bootCM(BOOTMODE\_BOOT\_TO\_FLASH\_SECTOR0);

**#else**

Device\_bootCM(BOOTMODE\_BOOT\_TO\_S0RAM);

**#endif**

//

// Disable pin locks and enable internal pull-ups.

//

Device\_initGPIO();

**#ifdef** ETHERNET

//

// Set up EnetCLK to use SYSPLL as the clock source and set the

// clock divider to 2.

//

// This way we ensure that the PTP clock is 100 MHz. Note that this value

// is not automatically/dynamically known to the CM core and hence it needs

// to be made available to the CM side code beforehand.

SysCtl\_setEnetClk(*SYSCTL\_ENETCLKOUT\_DIV\_2*, *SYSCTL\_SOURCE\_SYSPLL*);

//

// Configure the GPIOs for ETHERNET.

//

//

// MDIO Signals

//

**GPIO\_setPinConfig**(GPIO\_105\_ENET\_MDIO\_CLK);

**GPIO\_setPinConfig**(GPIO\_106\_ENET\_MDIO\_DATA);

//

// Use this only for RMII Mode

//GPIO\_setPinConfig(GPIO\_73\_ENET\_RMII\_CLK);

//

//

//MII Signals

//

**GPIO\_setPinConfig**(GPIO\_109\_ENET\_MII\_CRS);

**GPIO\_setPinConfig**(GPIO\_110\_ENET\_MII\_COL);

**GPIO\_setPinConfig**(GPIO\_75\_ENET\_MII\_TX\_DATA0);

**GPIO\_setPinConfig**(GPIO\_122\_ENET\_MII\_TX\_DATA1);

**GPIO\_setPinConfig**(GPIO\_123\_ENET\_MII\_TX\_DATA2);

**GPIO\_setPinConfig**(GPIO\_124\_ENET\_MII\_TX\_DATA3);

//

//Use this only if the TX Error pin has to be connected

//GPIO\_setPinConfig(GPIO\_46\_ENET\_MII\_TX\_ERR);

//

**GPIO\_setPinConfig**(GPIO\_118\_ENET\_MII\_TX\_EN);

**GPIO\_setPinConfig**(GPIO\_114\_ENET\_MII\_RX\_DATA0);

**GPIO\_setPinConfig**(GPIO\_115\_ENET\_MII\_RX\_DATA1);

**GPIO\_setPinConfig**(GPIO\_116\_ENET\_MII\_RX\_DATA2);

**GPIO\_setPinConfig**(GPIO\_117\_ENET\_MII\_RX\_DATA3);

**GPIO\_setPinConfig**(GPIO\_113\_ENET\_MII\_RX\_ERR);

**GPIO\_setPinConfig**(GPIO\_112\_ENET\_MII\_RX\_DV);

**GPIO\_setPinConfig**(GPIO\_44\_ENET\_MII\_TX\_CLK);

**GPIO\_setPinConfig**(GPIO\_111\_ENET\_MII\_RX\_CLK);

//

//Power down pin to bring the external PHY out of Power down

//

**GPIO\_setDirectionMode**(108, *GPIO\_DIR\_MODE\_OUT*);

**GPIO\_setPadConfig**(108, GPIO\_PIN\_TYPE\_PULLUP);

GPIO\_writePin(108,1);

//

//PHY Reset Pin to be driven High to bring external PHY out of Reset

//

**GPIO\_setDirectionMode**(119, *GPIO\_DIR\_MODE\_OUT*);

**GPIO\_setPadConfig**(119, GPIO\_PIN\_TYPE\_PULLUP);

GPIO\_writePin(119,1);

**#endif**

**#ifdef** MCAN

//

// Setting the MCAN Clock.

//

SysCtl\_setMCANClk(SYSCTL\_MCANCLK\_DIV\_4);

//

// Configuring the GPIOs for MCAN.

//

GPIO\_setPinConfig(DEVICE\_GPIO\_CFG\_MCANRXA);

GPIO\_setPinConfig(DEVICE\_GPIO\_CFG\_MCANTXA);

**#endif**

**#ifdef** CANA

//

// Configuring the GPIOs for CAN A.

//

GPIO\_setPinConfig(DEVICE\_GPIO\_CFG\_CANRXA);

GPIO\_setPinConfig(DEVICE\_GPIO\_CFG\_CANTXA);

//

// Allocate Shared Peripheral CAN A to the CM Side.

//

SysCtl\_allocateSharedPeripheral(SYSCTL\_PALLOCATE\_CAN\_A,0x1U);

**#endif**

**#ifdef** CANB

//

// Configuring the GPIOs for CAN B.

//

GPIO\_setPinConfig(DEVICE\_GPIO\_CFG\_CANRXB);

GPIO\_setPinConfig(DEVICE\_GPIO\_CFG\_CANTXB);

//

// Allocate Shared Peripheral CAN B to the CM Side.

//

SysCtl\_allocateSharedPeripheral(SYSCTL\_PALLOCATE\_CAN\_B,0x1U);

**#endif**

**#ifdef** UART

//

// Configure GPIO85 as the UART Rx pin.

//

GPIO\_setPinConfig(GPIO\_85\_UARTA\_RX);

GPIO\_setDirectionMode(85, GPIO\_DIR\_MODE\_IN);

GPIO\_setPadConfig(85, GPIO\_PIN\_TYPE\_STD);

GPIO\_setQualificationMode(85, GPIO\_QUAL\_ASYNC);

//

// Configure GPIO84 as the UART Tx pin.

//

GPIO\_setPinConfig(GPIO\_84\_UARTA\_TX);

GPIO\_setDirectionMode(84, GPIO\_DIR\_MODE\_OUT);

GPIO\_setPadConfig(84, GPIO\_PIN\_TYPE\_STD);

GPIO\_setQualificationMode(84, GPIO\_QUAL\_ASYNC);

**#endif**

**#ifdef** USB

**#ifdef** USE\_20MHZ\_XTAL

//

// Set up the auxiliary PLL so a 60 MHz output clock is provided to the USB module.

// This fixed frequency is required for all USB operations.

//

SysCtl\_setAuxClock(SYSCTL\_AUXPLL\_OSCSRC\_XTAL |

SYSCTL\_AUXPLL\_IMULT(48) |

SYSCTL\_REFDIV(2U) | SYSCTL\_ODIV(4U) |

SYSCTL\_AUXPLL\_DIV\_2 |

SYSCTL\_AUXPLL\_ENABLE |

SYSCTL\_DCC\_BASE\_0);

**#else**

//

// Set up the auxiliary PLL so a 60 MHz output clock is provided to the USB module.

// This fixed frequency is required for all USB operations.

//

SysCtl\_setAuxClock(SYSCTL\_AUXPLL\_OSCSRC\_XTAL |

SYSCTL\_AUXPLL\_IMULT(48) |

SYSCTL\_REFDIV(2U) | SYSCTL\_ODIV(5U) |

SYSCTL\_AUXPLL\_DIV\_2 |

SYSCTL\_AUXPLL\_ENABLE |

SYSCTL\_DCC\_BASE\_0);

**#endif**

//

// Allocate Shared Peripheral USB to the CM Side.

//

SysCtl\_allocateSharedPeripheral(SYSCTL\_PALLOCATE\_USBA, 1);

GPIO\_setPinConfig(GPIO\_0\_GPIO0);

GPIO\_setPadConfig(0, GPIO\_PIN\_TYPE\_STD);

GPIO\_setDirectionMode(0, GPIO\_DIR\_MODE\_OUT);

GPIO\_setMasterCore(0, GPIO\_CORE\_CM);

//

// Set the master core of GPIOs to CM.

//

GPIO\_setMasterCore(42, GPIO\_CORE\_CM);

GPIO\_setMasterCore(43, GPIO\_CORE\_CM);

GPIO\_setMasterCore(46, GPIO\_CORE\_CM);

GPIO\_setMasterCore(47, GPIO\_CORE\_CM);

GPIO\_setMasterCore(120, GPIO\_CORE\_CM);

GPIO\_setMasterCore(121, GPIO\_CORE\_CM);

//

// Set the USB DM and DP GPIOs.

//

GPIO\_setAnalogMode(42, GPIO\_ANALOG\_ENABLED);

GPIO\_setAnalogMode(43, GPIO\_ANALOG\_ENABLED);

//

// Set the direction for VBUS and ID.

//

GPIO\_setDirectionMode(46, GPIO\_DIR\_MODE\_IN);

GPIO\_setDirectionMode(47, GPIO\_DIR\_MODE\_IN);

//

// Configure the Power Fault.

//

GPIO\_setMasterCore(120, GPIO\_CORE\_CM);

GPIO\_setDirectionMode(120, GPIO\_DIR\_MODE\_IN);

//

// Configure the External Power Signal Enable.

//

GPIO\_setMasterCore(121, GPIO\_CORE\_CM);

GPIO\_setDirectionMode(121, GPIO\_DIR\_MODE\_OUT);

GPIO\_writePin(121, 1);

//

// Set the CM Clock to run at 120MHz.

// The CM Clock is a fractional multiple of the AUXPLL Clock (120 Mhz) from

// which the USB Clock (60 MHz) is derived.

//

SysCtl\_setCMClk(SYSCTL\_CMCLKOUT\_DIV\_1, SYSCTL\_SOURCE\_AUXPLL);

**#endif**

//ethernet\_end

//

initEPWMGPIO();

//// init gpio0~3,5 to create low freq(40hz) pulse

**GPIO\_setPadConfig**(6, GPIO\_PIN\_TYPE\_STD); // Enable pullup on GPIO6

GPIO\_writePin(6, 0); // Load output latch

**GPIO\_setPinConfig**(GPIO\_6\_GPIO6); // GPIO6 = GPIO6

**GPIO\_setDirectionMode**(6, *GPIO\_DIR\_MODE\_OUT*); // GPIO6 = output

**GPIO\_setPadConfig**(1, GPIO\_PIN\_TYPE\_STD); // Enable pullup on GPIO8

GPIO\_writePin(1, 0); // set\_low

**GPIO\_setPinConfig**(GPIO\_1\_GPIO1); // GPIO8 = GPIO8

**GPIO\_setDirectionMode**(1, *GPIO\_DIR\_MODE\_OUT*); // GPIO8 = output

**GPIO\_setPadConfig**(2, GPIO\_PIN\_TYPE\_STD); // Enable pullup on GPIO8

GPIO\_writePin(2, 0); // set\_low

**GPIO\_setPinConfig**(GPIO\_2\_GPIO2); // GPIO8 = GPIO8

**GPIO\_setDirectionMode**(2, *GPIO\_DIR\_MODE\_OUT*); // GPIO8 = output

**GPIO\_setPadConfig**(3, GPIO\_PIN\_TYPE\_STD); // Enable pullup on GPIO8

GPIO\_writePin(3, 0); // set\_low

**GPIO\_setPinConfig**(GPIO\_3\_GPIO3); // GPIO8 = GPIO8

**GPIO\_setDirectionMode**(3, *GPIO\_DIR\_MODE\_OUT*); // GPIO8 = output

**GPIO\_setPadConfig**(5, GPIO\_PIN\_TYPE\_STD); // Enable pullup on GPIO8

GPIO\_writePin(5, 0); // set\_low

**GPIO\_setPinConfig**(GPIO\_5\_GPIO5); // GPIO8 = GPIO8

**GPIO\_setDirectionMode**(5, *GPIO\_DIR\_MODE\_OUT*); // GPIO8 = output

//gpio28\_for\_spinning

**GPIO\_setPadConfig**(10, GPIO\_PIN\_TYPE\_STD); // Enable pullup on GPIO8

GPIO\_writePin(10, 0); // set\_low

**GPIO\_setPinConfig**(GPIO\_10\_GPIO10); // GPIO8 = GPIO8

**GPIO\_setDirectionMode**(10, *GPIO\_DIR\_MODE\_OUT*); // GPIO8 = output

//limit\_switch\_gpio24,25,26,27

**GPIO\_setPadConfig**(24, GPIO\_PIN\_TYPE\_STD); // Enable pullup on GPIO24

**GPIO\_setPinConfig**(GPIO\_24\_GPIO24); // GPIO24 = GPIO24

**GPIO\_setDirectionMode**(24, *GPIO\_DIR\_MODE\_IN*); // GPIO24 = input

**GPIO\_setPadConfig**(25, GPIO\_PIN\_TYPE\_STD); // Enable pullup on GPIO34

**GPIO\_setPinConfig**(GPIO\_25\_GPIO25); // GPIO34 = GPIO34

**GPIO\_setDirectionMode**(25, *GPIO\_DIR\_MODE\_IN*); // GPIO34 = input

**GPIO\_setPadConfig**(26, GPIO\_PIN\_TYPE\_STD); // Enable pullup on GPIO34

**GPIO\_setPinConfig**(GPIO\_26\_GPIO26); // GPIO34 = GPIO34

**GPIO\_setDirectionMode**(26, *GPIO\_DIR\_MODE\_IN*); // GPIO34 = input

**GPIO\_setPadConfig**(27, GPIO\_PIN\_TYPE\_STD); // Enable pullup on GPIO34

**GPIO\_setPinConfig**(GPIO\_27\_GPIO27); // GPIO34 = GPIO34

**GPIO\_setDirectionMode**(27, *GPIO\_DIR\_MODE\_IN*); // GPIO34 = input

**Interrupt\_initModule**();

**Interrupt\_initVectorTable**();

//ADC\_timer

//

// Set up ADCs, initializing the SOCs to be triggered by software

//

//

// Interrupts that are used in this example are re-mapped to ISR functions

// found within this file.

//

Interrupt\_register(INT\_ADCA1, &adcA1ISR);

//

// Set up the ADC and the ePWM and initialize the SOC

//

configureADC(ADCA\_BASE);

configureADC(ADCC\_BASE);

initEPWM7();

initADCSOC();

//

// Enable ADC interrupt

//

**Interrupt\_enable**(INT\_ADCA1);

//

//ipc\_timers

**IPC\_registerInterrupt**(*IPC\_CPU1\_L\_CM\_R*, IPC\_INT1, IPC\_ISR1);

IPC\_clearFlagLtoR(*IPC\_CPU1\_L\_CM\_R*, IPC\_FLAG\_ALL);

IPC\_sync(*IPC\_CPU1\_L\_CM\_R*, IPC\_FLAG31);

//other\_timers

Interrupt\_register(INT\_TIMER0, &cpuTimer0ISR);

Interrupt\_register(INT\_TIMER1, &cpuTimer1ISR);

Interrupt\_register(INT\_TIMER2, &cpuTimer2ISR);

initEPWM(EPWM3\_BASE);

initEPWM(EPWM5\_BASE);

initCPUTimers();

//

// Configure CPU-Timer 0, 1, and 2 to interrupt every second:

// 1 second Period (in uSeconds)

//

configCPUTimer(CPUTIMER0\_BASE, timer0\_period \* 1000000);

configCPUTimer(CPUTIMER1\_BASE, 1000);

configCPUTimer(CPUTIMER2\_BASE, 1000);

//

// To ensure precise timing, use write-only instructions to write to the

// entire register. Therefore, if any of the configuration bits are changed

// in configCPUTimer and initCPUTimers, the below settings must also

// be updated.

//

CPUTimer\_enableInterrupt(CPUTIMER0\_BASE);

CPUTimer\_enableInterrupt(CPUTIMER1\_BASE);

CPUTimer\_enableInterrupt(CPUTIMER2\_BASE);

//

// Enable interrupts required for this example

// Enables CPU int1, int13, and int14 which are connected to CPU-Timer 0,

// CPU-Timer 1, and CPU-Timer 2 respectively.

// Enable TINT0 in the PIE: Group 1 interrupt 7

//Interrupt\_enable(INT\_EPWM1);

**Interrupt\_enable**(INT\_TIMER0);

**Interrupt\_enable**(INT\_TIMER1);

**Interrupt\_enable**(INT\_TIMER2);

//

// Starts CPU-Timer 0, CPU-Timer 1, and CPU-Timer 2.

//

CPUTimer\_startTimer(CPUTIMER0\_BASE);

CPUTimer\_startTimer(CPUTIMER1\_BASE);

CPUTimer\_startTimer(CPUTIMER2\_BASE);

//

// Enable Global Interrupt (INTM) and realtime interrupt (DBGM)

//

EINT;

ERTM;

//

// Start ePWM1, enabling SOCA and putting the counter in up-count mode

//

EPWM\_enableADCTrigger(EPWM7\_BASE, *EPWM\_SOC\_A*);

EPWM\_setTimeBaseCounterMode(EPWM7\_BASE, *EPWM\_COUNTER\_MODE\_UP*);

//

// Take conversions indefinitely in loop

//

**do**

{

//

// Wait while ePWM causes ADC conversions.

// ADCA1 ISR processes each new set of conversions.

//

}

**while**(1);

}

**\_\_interrupt** **void** **IPC\_ISR1**() {

uint32\_t command, addr, data;

**int** i;

**bool** status = **false**;

**IPC\_readCommand**(*IPC\_CPU1\_L\_CM\_R*, IPC\_FLAG1, IPC\_ADDR\_CORRECTION\_ENABLE,

&command, &addr, &data);

**if**(command == IPC\_CMD\_READ\_MEM)

{

**for**(i=0; i<data; i++)

{

recData\_c28[i] = \*((**float** \*)addr + i);

}

x\_rj = recData\_c28[0];

y\_rj = recData\_c28[1];

front\_cmd = recData\_c28[4];

back\_cmd = recData\_c28[5];

spin\_cmd = recData\_c28[6];

rcm\_en\_cmd = recData\_c28[7];

ee\_en\_cmd = recData\_c28[8];

STOP\_cmd = recData\_c28[9];

**if** (x\_rj != 0)

x\_rj = x\_rj - steady\_joystick;

**if** (y\_rj != 0)

y\_rj = y\_rj - steady\_joystick;

}

**if**(status)

{

IPC\_sendResponse(*IPC\_CPU1\_L\_CM\_R*, TEST\_PASS);

}

**else**

{

IPC\_sendResponse(*IPC\_CPU1\_L\_CM\_R*, TEST\_FAIL);

}

IPC\_ackFlagRtoL(*IPC\_CPU1\_L\_CM\_R*, IPC\_FLAG1);

Interrupt\_clearACKGroup(INTERRUPT\_ACK\_GROUP11);

}

//

// cpuTimer1ISR - Counter for CpuTimer1

//

**\_\_interrupt** **void**

**cpuTimer1ISR**(**void**)

{

count += 1;

readData[0]=F;

readData[1]=curloc;

readData[2]=oral\_dpt;

readData[3]=limitRight;

**IPC\_sendCommand**(*IPC\_CPU1\_L\_CM\_R*, IPC\_FLAG0, IPC\_ADDR\_CORRECTION\_ENABLE,

IPC\_CMD\_READ\_MEM, (uint32\_t)readData, 4);

IPC\_waitForAck(*IPC\_CPU1\_L\_CM\_R*, IPC\_FLAG0);

**if**(IPC\_getResponse(*IPC\_CPU1\_L\_CM\_R*) == TEST\_PASS){

pass = 1;

}

**else**{

pass = 0;

}

//

// The CPU acknowledges the interrupt.

//

cpuTimer1IntCount++;

}

//

// cpuTimer0ISR - Counter for CpuTimer0

//

**\_\_interrupt** **void**

**cpuTimer0ISR**(**void**)

{

cpuTimer0IntCount++;

}

//

// cpuTimer2ISR - Counter for CpuTimer2

//

**\_\_interrupt** **void**

**cpuTimer2ISR**(**void**)

{

F=(adc\_iiR-745.58)/4.8347;

//

// The CPU acknowledges the interrupt.

//

cpuTimer2IntCount++;

/\*

EPWM\_setTimeBasePeriod(EPWM5\_BASE, 3125000/tf);//100kHz

EPWM\_setCounterCompareValue(EPWM5\_BASE, EPWM\_COUNTER\_COMPARE\_A, 3125000/tf/2);//duty

if(tf!=0){

if (GPIO\_readPinDataRegister(10) == 1){

GPIO\_writePin(10,0);

qqq=2;

}

}

else{

if (GPIO\_readPinDataRegister(10) == 0){

GPIO\_writePin(10,1);

qqq=2;

}

}

\*/

CmdToDuty();

}

**void** **CmdToDuty**(**void**)

{

//change\_start

**if** (front\_cmd != 0)

{

freq=3125000/1000;

duty = 0.5\*freq;

EPWM\_setTimeBasePeriod(EPWM3\_BASE, freq);//100kHz

EPWM\_setCounterCompareValue(EPWM3\_BASE, *EPWM\_COUNTER\_COMPARE\_A*, duty);//duty

GPIO\_writePin(5,1);

}

**else** **if** (back\_cmd != 0)

{

test1++;

EPWM\_setTimeBasePeriod(EPWM3\_BASE, 3125000/back\_cmd);//100kHz

EPWM\_setCounterCompareValue(EPWM3\_BASE, *EPWM\_COUNTER\_COMPARE\_A*, 3125000/back\_cmd/2);//duty

GPIO\_writePin(5,0);

}

**else**

{

EPWM\_setCounterCompareValue(EPWM3\_BASE, *EPWM\_COUNTER\_COMPARE\_A*, 0);//duty

}

**if** (spin\_cmd != 0)

{

test3++;

EPWM\_setTimeBasePeriod(EPWM5\_BASE, 3125000/50);//100kHz

EPWM\_setCounterCompareValue(EPWM5\_BASE, *EPWM\_COUNTER\_COMPARE\_A*, 3125000/50/2);//duty

}

**else**

{

qqq=1;

EPWM\_setCounterCompareValue(EPWM5\_BASE, *EPWM\_COUNTER\_COMPARE\_A*, 0);//duty

}

**if** (ee\_en\_cmd == 1)

{

**if** (GPIO\_readPinDataRegister(10) == 1){

GPIO\_writePin(10,0);

qqq=2;

}

}

**else**

{

**if** (GPIO\_readPinDataRegister(10) == 0){

GPIO\_writePin(10,1);

qqq=3;

}

}

}

**void** **updateLimitS**(**void**)//clear

{

test4=100;

**if**(GPIO\_readPin(24)==0){

test6=2;

limitUp = 0;

}

**if**(GPIO\_readPin(24)==1){

test6=1;

limitUp = 1;

}

**if**(GPIO\_readPin(25)==0){

limitDown = 0;

}

**if**(GPIO\_readPin(25)==1){

limitDown = 1;

}

**if**(GPIO\_readPin(26)==0){

limitLeft = 0;

}

**if**(GPIO\_readPin(26)==1){

limitLeft = 1;

}

**if**(GPIO\_readPin(27)==0){

test5=2;

limitRight = 0;

}

**if**(GPIO\_readPin(27)==1){

test5=11;

limitRight = 1;

}

}

**void** **initEPWMGPIO**(**void**)

{

//

// Disable pull up on GPIO0-5 and configure them as PWMs

//

/\*

GPIO\_setPadConfig(0, GPIO\_PIN\_TYPE\_STD);

GPIO\_setPinConfig(GPIO\_0\_EPWM1A);

GPIO\_setPadConfig(1, GPIO\_PIN\_TYPE\_STD);

GPIO\_setPinConfig(GPIO\_1\_EPWM1B);

GPIO\_setPadConfig(2, GPIO\_PIN\_TYPE\_STD);

GPIO\_setPinConfig(GPIO\_2\_EPWM2A);

GPIO\_setPadConfig(3, GPIO\_PIN\_TYPE\_STD);

GPIO\_setPinConfig(GPIO\_3\_EPWM2B);

\*/

**GPIO\_setPadConfig**(4, GPIO\_PIN\_TYPE\_STD);

**GPIO\_setPinConfig**(GPIO\_4\_EPWM3A);

**GPIO\_setPadConfig**(5, GPIO\_PIN\_TYPE\_STD);

**GPIO\_setPinConfig**(GPIO\_5\_EPWM3B);

**GPIO\_setPadConfig**(8, GPIO\_PIN\_TYPE\_STD);

**GPIO\_setPinConfig**(GPIO\_8\_EPWM5A);

**GPIO\_setPadConfig**(9, GPIO\_PIN\_TYPE\_STD);

**GPIO\_setPinConfig**(GPIO\_9\_EPWM5B);

}

**void**

**initCPUTimers**(**void**)

{

//

// Initialize timer period to maximum

//

CPUTimer\_setPeriod(CPUTIMER0\_BASE, 0xFFFFFFFF);

CPUTimer\_setPeriod(CPUTIMER1\_BASE, 0xFFFFFFFF);

CPUTimer\_setPeriod(CPUTIMER2\_BASE, 0xFFFFFFFF);

//

// Initialize pre-scale counter to divide by 1 (SYSCLKOUT)

//

CPUTimer\_setPreScaler(CPUTIMER0\_BASE, 0);

CPUTimer\_setPreScaler(CPUTIMER1\_BASE, 0);

CPUTimer\_setPreScaler(CPUTIMER2\_BASE, 0);

//

// Make sure timer is stopped

//

CPUTimer\_stopTimer(CPUTIMER0\_BASE);

CPUTimer\_stopTimer(CPUTIMER1\_BASE);

CPUTimer\_stopTimer(CPUTIMER2\_BASE);

//

// Reload all counter register with period value

//

CPUTimer\_reloadTimerCounter(CPUTIMER0\_BASE);

CPUTimer\_reloadTimerCounter(CPUTIMER1\_BASE);

CPUTimer\_reloadTimerCounter(CPUTIMER2\_BASE);

//

// Reset interrupt counter

//

cpuTimer0IntCount = 0;

cpuTimer1IntCount = 0;

cpuTimer2IntCount = 0;

}

//

// configCPUTimer - This function initializes the selected timer to the

// period specified by the "freq" and "period" variables. The "freq" is

// CPU frequency in Hz and the period in uSeconds. The timer is held in

// the stopped state after configuration.

//

**void**

**configCPUTimer**(uint32\_t cpuTimer, uint32\_t period)

{

uint32\_t temp, freq = DEVICE\_SYSCLK\_FREQ;

//

// Initialize timer period:

//

temp = ((freq / 1000000) \* period);

CPUTimer\_setPeriod(cpuTimer, temp);

//

// Set pre-scale counter to divide by 1 (SYSCLKOUT):

//

CPUTimer\_setPreScaler(cpuTimer, 0);

//

// Initializes timer control register. The timer is stopped, reloaded,

// free run disabled, and interrupt enabled.

// Additionally, the free and soft bits are set

//

CPUTimer\_stopTimer(cpuTimer);

CPUTimer\_reloadTimerCounter(cpuTimer);

**CPUTimer\_setEmulationMode**(cpuTimer,

*CPUTIMER\_EMULATIONMODE\_STOPAFTERNEXTDECREMENT*);

CPUTimer\_enableInterrupt(cpuTimer);

//

// Resets interrupt counters for the three cpuTimers

//

**if** (cpuTimer == CPUTIMER0\_BASE)

{

cpuTimer0IntCount = 0;

}

**else** **if**(cpuTimer == CPUTIMER1\_BASE)

{

cpuTimer1IntCount = 0;

}

**else** **if**(cpuTimer == CPUTIMER2\_BASE)

{

cpuTimer2IntCount = 0;

}

}

//

// initEPWM - Configure ePWM1

//

**void** **initEPWM**(uint32\_t base)

{

//

// Set-up TBCLK

//

EPWM\_setTimeBasePeriod(base, 3125000/160);

EPWM\_setPhaseShift(base, 0U);

EPWM\_setTimeBaseCounter(base, 0U);

EPWM\_setTimeBaseCounterMode(base, *EPWM\_COUNTER\_MODE\_UP\_DOWN*);

EPWM\_disablePhaseShiftLoad(base);

//

// Set ePWM clock pre-scaler

//

EPWM\_setClockPrescaler(base,

*EPWM\_CLOCK\_DIVIDER\_4*,

*EPWM\_HSCLOCK\_DIVIDER\_4*);

//

// Set up shadowing

//

EPWM\_setCounterCompareShadowLoadMode(base,

*EPWM\_COUNTER\_COMPARE\_A*,

*EPWM\_COMP\_LOAD\_ON\_CNTR\_ZERO*);

//

// Set-up compare

//

EPWM\_setCounterCompareValue(base, *EPWM\_COUNTER\_COMPARE\_A*, EPWM\_TIMER\_TBPRD/4);

EPWM\_setCounterCompareValue(base, *EPWM\_COUNTER\_COMPARE\_B*, 3\*EPWM\_TIMER\_TBPRD/4);

//

// Set actions

//

EPWM\_setActionQualifierAction(base,

*EPWM\_AQ\_OUTPUT\_A*,

*EPWM\_AQ\_OUTPUT\_LOW*,

*EPWM\_AQ\_OUTPUT\_ON\_TIMEBASE\_ZERO*);

EPWM\_setActionQualifierAction(base,

*EPWM\_AQ\_OUTPUT\_A*,

*EPWM\_AQ\_OUTPUT\_HIGH*,

*EPWM\_AQ\_OUTPUT\_ON\_TIMEBASE\_UP\_CMPA*);

EPWM\_setActionQualifierAction(base,

*EPWM\_AQ\_OUTPUT\_A*,

*EPWM\_AQ\_OUTPUT\_NO\_CHANGE*,

*EPWM\_AQ\_OUTPUT\_ON\_TIMEBASE\_PERIOD*);

EPWM\_setActionQualifierAction(base,

*EPWM\_AQ\_OUTPUT\_A*,

*EPWM\_AQ\_OUTPUT\_LOW*,

*EPWM\_AQ\_OUTPUT\_ON\_TIMEBASE\_DOWN\_CMPA*);

EPWM\_setActionQualifierAction(base,

*EPWM\_AQ\_OUTPUT\_B*,

*EPWM\_AQ\_OUTPUT\_LOW*,

*EPWM\_AQ\_OUTPUT\_ON\_TIMEBASE\_ZERO*);

EPWM\_setActionQualifierAction(base,

*EPWM\_AQ\_OUTPUT\_B*,

*EPWM\_AQ\_OUTPUT\_HIGH*,

*EPWM\_AQ\_OUTPUT\_ON\_TIMEBASE\_UP\_CMPB*);

EPWM\_setActionQualifierAction(base,

*EPWM\_AQ\_OUTPUT\_B*,

*EPWM\_AQ\_OUTPUT\_NO\_CHANGE*,

*EPWM\_AQ\_OUTPUT\_ON\_TIMEBASE\_PERIOD*);

EPWM\_setActionQualifierAction(base,

*EPWM\_AQ\_OUTPUT\_B*,

*EPWM\_AQ\_OUTPUT\_LOW*,

*EPWM\_AQ\_OUTPUT\_ON\_TIMEBASE\_DOWN\_CMPB*);

}

//adc

////

// configureADC - Write ADC configurations and power up the ADC for the

// selected ADC

//

**void** **configureADC**(uint32\_t adcBase)

{

//

// Set ADCCLK divider to /4

//

ADC\_setPrescaler(adcBase, *ADC\_CLK\_DIV\_4\_0*);

//

// Set resolution and signal mode (see #defines above) and load

// corresponding trims.

//

**#if**(EX\_ADC\_RESOLUTION == 12)

**ADC\_setMode**(adcBase, *ADC\_RESOLUTION\_12BIT*, *ADC\_MODE\_SINGLE\_ENDED*);

**#elif**(EX\_ADC\_RESOLUTION == 16)

**#if**(EX\_ADC\_SIGNALMODE == "SINGLE-ENDED")

ADC\_setMode(adcBase, ADC\_RESOLUTION\_16BIT, ADC\_MODE\_SINGLE\_ENDED);

**#elif**(EX\_ADC\_SIGNALMODE == "DIFFERENTIAL")

ADC\_setMode(adcBase, ADC\_RESOLUTION\_16BIT, ADC\_MODE\_DIFFERENTIAL);

**#endif**

**#endif**

//

// Set pulse positions to late

//

ADC\_setInterruptPulseMode(adcBase, *ADC\_PULSE\_END\_OF\_CONV*);

//

// Power up the ADCs and then delay for 1 ms

//

ADC\_enableConverter(adcBase);

//

// Delay for 1ms to allow ADC time to power up

//

DEVICE\_DELAY\_US(1000);

}

//

// Function to configure ePWM1 to generate the SOC.

//

**void** **initEPWM7**(**void**)

{

//

// Disable SOCA

//

EPWM\_disableADCTrigger(EPWM7\_BASE, *EPWM\_SOC\_A*);

//

// Configure the SOC to occur on the first up-count event

//

EPWM\_setADCTriggerSource(EPWM7\_BASE, *EPWM\_SOC\_A*, *EPWM\_SOC\_TBCTR\_U\_CMPA*);

EPWM\_setADCTriggerEventPrescale(EPWM7\_BASE, *EPWM\_SOC\_A*, 1);

//

// Set the compare A value to 1000 and the period to 1999

// Assuming ePWM clock is 100MHz, this would give 50kHz sampling

// 50MHz ePWM clock would give 25kHz sampling, etc.

// The sample rate can also be modulated by changing the ePWM period

// directly (ensure that the compare A value is less than the period).

//

EPWM\_setCounterCompareValue(EPWM7\_BASE, *EPWM\_COUNTER\_COMPARE\_A*, 1000);

EPWM\_setTimeBasePeriod(EPWM7\_BASE, 1999);

//

// Set the local ePWM module clock divider to /1

//

EPWM\_setClockPrescaler(EPWM7\_BASE,

*EPWM\_CLOCK\_DIVIDER\_1*,

*EPWM\_HSCLOCK\_DIVIDER\_1*);

//

// Freeze the counter

//

EPWM\_setTimeBaseCounterMode(EPWM7\_BASE, *EPWM\_COUNTER\_MODE\_STOP\_FREEZE*);

}

//

// Function to configure SOCs on ADCA and ADCC to be triggered by ePWM1.

//

**void** **initADCSOC**(**void**)

{

uint16\_t acqps;

//

// Determine minimum acquisition window (in SYSCLKS) based on resolution

//

**if**(EX\_ADC\_RESOLUTION == 12)

{

acqps = 14; // 75ns

}

**else** //resolution is 16-bit

{

acqps = 63; // 320ns

}

//

// - NOTE: A longer sampling window will be required if the ADC driving

// source is less than ideal (an ideal source would be a high bandwidth

// op-amp with a small series resistance). See TI application report

// SPRACT6 for guidance on ADC driver design.

// - NOTE: SOCs need not use the same S+H window duration, but SOCs

// occurring in parallel (in this example, SOC0 on both ADCs occur in

// parallel, as do the SOC1s on both ADCs, etc.) should usually

// use the same value to ensure simultaneous samples and synchronous

// operation.

//

// Select the channels to convert and the configure the ePWM trigger

//

ADC\_setupSOC(ADCA\_BASE, *ADC\_SOC\_NUMBER0*, *ADC\_TRIGGER\_EPWM7\_SOCA*,

*ADC\_CH\_ADCIN0*, acqps);

ADC\_setupSOC(ADCA\_BASE, *ADC\_SOC\_NUMBER1*, *ADC\_TRIGGER\_EPWM7\_SOCA*,

*ADC\_CH\_ADCIN1*, acqps);

ADC\_setupSOC(ADCA\_BASE, *ADC\_SOC\_NUMBER2*, *ADC\_TRIGGER\_EPWM7\_SOCA*,

*ADC\_CH\_ADCIN2*, acqps);

ADC\_setupSOC(ADCC\_BASE, *ADC\_SOC\_NUMBER0*, *ADC\_TRIGGER\_EPWM7\_SOCA*,

*ADC\_CH\_ADCIN2*, acqps);

ADC\_setupSOC(ADCC\_BASE, *ADC\_SOC\_NUMBER1*, *ADC\_TRIGGER\_EPWM7\_SOCA*,

*ADC\_CH\_ADCIN3*, acqps);

ADC\_setupSOC(ADCC\_BASE, *ADC\_SOC\_NUMBER2*, *ADC\_TRIGGER\_EPWM7\_SOCA*,

*ADC\_CH\_ADCIN4*, acqps);

//

// Selec SOC2 on ADCA as the interrupt source. SOC2 on ADCC will end at

// the same time, so either SOC2 would be an acceptable interrupt triggger.

//

ADC\_setInterruptSource(ADCA\_BASE, *ADC\_INT\_NUMBER1*, *ADC\_SOC\_NUMBER2*);

ADC\_enableInterrupt(ADCA\_BASE, *ADC\_INT\_NUMBER1*);

ADC\_clearInterruptStatus(ADCA\_BASE, *ADC\_INT\_NUMBER1*);

}

//

// ADC A Interrupt 1 ISR

//

**\_\_interrupt** **void** **adcA1ISR**(**void**)

{

//

// Store results

//

adcAResult0 = ADC\_readResult(ADCARESULT\_BASE, *ADC\_SOC\_NUMBER0*);

adcAResult1 = ADC\_readResult(ADCARESULT\_BASE, *ADC\_SOC\_NUMBER1*);

adcAResult2 = ADC\_readResult(ADCARESULT\_BASE, *ADC\_SOC\_NUMBER2*);

//adcCResult0 = ADC\_readResult(ADCCRESULT\_BASE, ADC\_SOC\_NUMBER0);

//adcCResult1 = ADC\_readResult(ADCCRESULT\_BASE, ADC\_SOC\_NUMBER1);

//adcCResult2 = ADC\_readResult(ADCCRESULT\_BASE, ADC\_SOC\_NUMBER2);

/\*

//lpf

adcFinalRe = 0.9\*adcAResult0+0.1\*adcRe[1];

adcRe[0] = adcAResult0;

adcRe[1] = adcFinalRe;

//lpf\_test\_const

adcFinalReT = k\*adcAResult0+(1-k)\*adcReT[1];

adcReT[0] = adcAResult0;

adcReT[1] = adcFinalReT;

//lpf\_1od

adcFinalReT1 = (adcAResult0 + k1 \* adcReT1[1])/(k1+1);

adcReT1[0] = adcAResult0;

adcReT1[1] = adcFinalReT1;

//lpf\_2od

dt2=adcFinalReT2;

adcFinalReT2 = (adcAResult0 +(k2\*adCR\*adCC+2\*k2\*adCL\*adCC)\*adcReT2[1]-k2\*adCL\*adCC\*(adcReT2[2]))/(k2\*adCL\*adCC+1+k2\*adCR\*adCC);

adcReT2[0] = adcAResult0;

adcReT2[1] = adcFinalReT2;

adcReT2[2] = dt2;

//lpf\_end

\*/

//iiR

iiR[1] = iiR[0];

wx0=adcAResult0;

wy0=(B0\*wx0+B1\*wx1+B2\*wx2)\*gain-wy1\*A1-wy2\*A2;

adc\_iiR=wy0/A0;

wx2=wx1;wx1=wx0;

wy2=wy1;wy1=wy0;

iiR[0]=adc\_iiR;

dif = iiR[0]-iiR[1];

//iiR\_end

//

// Clear the interrupt flag

//

ADC\_clearInterruptStatus(ADCA\_BASE, *ADC\_INT\_NUMBER1*);

//

// Check if overflow has occurred

//

**if**(**true** == ADC\_getInterruptOverflowStatus(ADCA\_BASE, *ADC\_INT\_NUMBER1*))

{

ADC\_clearInterruptOverflowStatus(ADCA\_BASE, *ADC\_INT\_NUMBER1*);

ADC\_clearInterruptStatus(ADCA\_BASE, *ADC\_INT\_NUMBER1*);

}

//

// Acknowledge the interrupt

//

Interrupt\_clearACKGroup(INTERRUPT\_ACK\_GROUP1);

}