Handout
SSP.c

```
/*****************************************************************************
 *   $Id:: ssp.c 5804 2010-12-04 00:32:12Z usb00423                          $
 *   Project: NXP LPC17xx SSP example
 *
 *   Description:
 *     This file contains SSP code example which include SSP initialization,
 *     SSP interrupt handler, and APIs for SSP access.
 *
 *****************************************************************************
 * Software that is described herein is for illustrative purposes only
 * which provides customers with programming information regarding the
 * products. This software is supplied "AS IS" without any warranties.
 * NXP Semiconductors assumes no responsibility or liability for the
 * use of the software, conveys no license or title under any patent,
 * copyright, or mask work right to the product. NXP Semiconductors
 * reserves the right to make changes in the software without
 * notification. NXP Semiconductors also make no representation or
 * warranty that such application will be suitable for the specified
 * use without further testing or modification.
*****************************************************************************/
#include "LPC17xx.h"                    /* LPC17xx Peripheral Registers */
#include "ssp.h"

/* statistics of all the interrupts */
volatile uint32_t interrupt0RxStat = 0;
volatile uint32_t interrupt0OverRunStat = 0;
volatile uint32_t interrupt0RxTimeoutStat = 0;
volatile uint32_t interrupt1RxStat = 0;
volatile uint32_t interrupt1OverRunStat = 0;
volatile uint32_t interrupt1RxTimeoutStat = 0;

/*****************************************************************************
** Function name:       SSP_IRQHandler
**
** Descriptions:        SSP port is used for SPI communication.
**                      SSP interrupt handler
**                      The algorithm is, if RXFIFO is at least half full,
**                      start receive until it's empty; if TXFIFO is at least
**                      half empty, start transmit until it's full.
**                      This will maximize the use of both FIFOs and performance.
** parameters:                None
** Returned value:            None
**
*****************************************************************************/
void SSP0_IRQHandler(void)
{
  uint32_t regValue;

  regValue = LPC_SSP0->MIS;
  if ( regValue & SSPMIS_RORMIS )   /* Receive overrun interrupt */
  {
      interrupt0OverRunStat++;
      LPC_SSP0->ICR = SSPICR_RORIC;       /* clear interrupt */
  }
```

```
  if ( regValue & SSPMIS_RTMIS )    /* Receive timeout interrupt */
  {
      interrupt0RxTimeoutStat++;
      LPC_SSP0->ICR = SSPICR_RTIC;        /* clear interrupt */
  }

  /* please be aware that, in main and ISR, CurrentRxIndex and CurrentTxIndex
  are shared as global variables. It may create some race condition that main
  and ISR manipulate these variables at the same time. SSPSR_BSY checking
  (polling)
  in both main and ISR could prevent this kind of race condition */

  if ( regValue & SSPMIS_RXMIS )    /* Rx at least half full */
  {
      interrupt0RxStat++;           /* receive until it's empty */
  }
  return;
}

/*****************************************************************************
** Function name:       SSP_IRQHandler
**
** Descriptions:        SSP port is used for SPI communication.
**                      SSP interrupt handler
**                      The algorithm is, if RXFIFO is at least half full,
**                      start receive until it's empty; if TXFIFO is at least
**                      half empty, start transmit until it's full.
**                      This will maximize the use of both FIFOs and performance.
** parameters:                None
** Returned value:            None
**
*****************************************************************************/
void SSP1_IRQHandler(void)
{
  uint32_t regValue;

  regValue = LPC_SSP1->MIS;
  if ( regValue & SSPMIS_RORMIS )   /* Receive overrun interrupt */
  {
      interrupt1OverRunStat++;
      LPC_SSP1->ICR = SSPICR_RORIC;       /* clear interrupt */
  }
  if ( regValue & SSPMIS_RTMIS )    /* Receive timeout interrupt */
  {
      interrupt1RxTimeoutStat++;
      LPC_SSP1->ICR = SSPICR_RTIC;        /* clear interrupt */
  }

  /* please be aware that, in main and ISR, CurrentRxIndex and CurrentTxIndex
  are shared as global variables. It may create some race condition that main
  and ISR manipulate these variables at the same time. SSPSR_BSY checking
  (polling)
  in both main and ISR could prevent this kind of race condition */
  if ( regValue & SSPMIS_RXMIS )    /* Rx at least half full */
  {
      interrupt1RxStat++;           /* receive until it's empty */
  }
  return;
```

```
}

/*****************************************************************************
** Function name:       SSP0_SSELToggle
**
** Descriptions:        SSP0 CS manual set
**
** parameters:          port num, toggle(1 is high, 0 is low)
** Returned value:      None
**
*****************************************************************************/
void SSP_SSELToggle( uint32_t portnum, uint32_t toggle )
{
  if ( portnum == 0 )
  {
    if ( !toggle )
        LPC_GPIO0->FIOCLR |= (0x1<<16);
    else
        LPC_GPIO0->FIOSET |= (0x1<<16);
  }
  else if ( portnum == 1 )
  {
    if ( !toggle )
        LPC_GPIO0->FIOCLR |= (0x1<<6);
    else
        LPC_GPIO0->FIOSET |= (0x1<<6);
  }
  return;
}

/*****************************************************************************
** Function name:       SSPInit
**
** Descriptions:        SSP port initialization routine
**
** parameters:                None
** Returned value:            None
**
*****************************************************************************/
void SSP0Init( void )
{
  uint8_t i, Dummy=Dummy;

  /* Enable AHB clock to the SSP0. */
  LPC_SC->PCONP |= (0x1<<21);

  /* Further divider is needed on SSP0 clock. Using default divided by 4 */
  LPC_SC->PCLKSEL1 &= ~(0x3<<10);

  /* P0.15~0.18 as SSP0 */
  LPC_PINCON->PINSEL0 &= ~(0x3UL<<30);
  LPC_PINCON->PINSEL0 |= (0x2UL<<30);
  LPC_PINCON->PINSEL1 &= ~((0x3<<0)|(0x3<<2)|(0x3<<4));
  LPC_PINCON->PINSEL1 |= ((0x2<<0)|(0x2<<2)|(0x2<<4));

#if !USE_CS
  LPC_PINCON->PINSEL1 &= ~(0x3<<0);
  LPC_GPIO0->FIODIR |= (0x1<<16);         /* P0.16 defined as GPIO and Outputs */
```

```
#endif

  /* Set DSS data to 8-bit, Frame format SPI, CPOL = 0, CPHA = 0, and SCR is 15 */
  LPC_SSP0->CR0 = 0x0707;

  /* SSPCPSR clock prescale register, master mode, minimum divisor is 0x02 */
  LPC_SSP0->CPSR = 0x2;

  for ( i = 0; i < FIFOSIZE; i++ )
  {
      Dummy = LPC_SSP0->DR;         /* clear the RxFIFO */
  }

  /* Enable the SSP Interrupt */
  NVIC_EnableIRQ(SSP0_IRQn);

  /* Device select as master, SSP Enabled */
#if LOOPBACK_MODE
  LPC_SSP0->CR1 = SSPCR1_LBM | SSPCR1_SSE;
#else
#if SSP_SLAVE
  /* Slave mode */
  if ( LPC_SSP0->CR1 & SSPCR1_SSE )
  {
      /* The slave bit can't be set until SSE bit is zero. */
      LPC_SSP0->CR1 &= ~SSPCR1_SSE;
  }
  LPC_SSP0->CR1 = SSPCR1_MS;        /* Enable slave bit first */
  LPC_SSP0->CR1 |= SSPCR1_SSE;      /* Enable SSP */
#else
  /* Master mode */
  LPC_SSP0->CR1 = SSPCR1_SSE;
#endif
#endif
  /* Set SSPINMS registers to enable interrupts */
  /* enable all error related interrupts */
  LPC_SSP0->IMSC = SSPIMSC_RORIM | SSPIMSC_RTIM;
  return;
}

/*****************************************************************************
** Function name:       SSPInit
**
** Descriptions:        SSP port initialization routine
**
** parameters:                None
** Returned value:            None
**
*****************************************************************************/
void SSP1Init( void )
{
  uint8_t i, Dummy=Dummy;

  /* Enable AHB clock to the SSP1. */
  LPC_SC->PCONP |= (0x1<<10);

  /* Further divider is needed on SSP1 clock. Using default divided by 8  */
  LPC_SC->PCLKSEL0 |= (0x3<<20);
```

```
  /* P0.6~0.9 as SSP1 */
  LPC_PINCON->PINSEL0 &= ~((0x3<<12)|(0x3<<14)|(0x3<<16)|(0x3<<18));
  LPC_PINCON->PINSEL0 |= ((0x2<<12)|(0x2<<14)|(0x2<<16)|(0x2<<18));

//#if !USE_CS
  LPC_PINCON->PINSEL0 &= ~(0x3<<12);
  LPC_GPIO0->FIODIR |= (0x1<<6);          /* P0.6 defined as GPIO and Outputs */
//#endif

  /* Set DSS data to 8-bit, Frame format SPI, CPOL = 0, CPHA = 0, and SCR is 15 */
  LPC_SSP1->CR0 = 0x0707;

  /* SSPCPSR clock prescale register, master mode, minimum divisor is 0x02 */
  LPC_SSP1->CPSR = 0x2;

  for ( i = 0; i < FIFOSIZE; i++ )
  {
      Dummy = LPC_SSP1->DR;         /* clear the RxFIFO */
  }

  /* Enable the SSP Interrupt */
  NVIC_EnableIRQ(SSP1_IRQn);

  /* Device select as master, SSP Enabled */
#if LOOPBACK_MODE
  LPC_SSP1->CR1 = SSPCR1_LBM | SSPCR1_SSE;
#else
#if SSP_SLAVE
  /* Slave mode */
  if ( LPC_SSP1->CR1 & SSPCR1_SSE )
  {
      /* The slave bit can't be set until SSE bit is zero. */
      LPC_SSP1->CR1 &= ~SSPCR1_SSE;
  }
  LPC_SSP1->CR1 = SSPCR1_MS;        /* Enable slave bit first */
  LPC_SSP1->CR1 |= SSPCR1_SSE;      /* Enable SSP */
#else
  /* Master mode */
  LPC_SSP1->CR1 = SSPCR1_SSE;
#endif
#endif
  /* Set SSPINMS registers to enable interrupts */
  /* enable all error related interrupts */
  LPC_SSP1->IMSC = SSPIMSC_RORIM | SSPIMSC_RTIM;
  return;
}

/*****************************************************************************
** Function name:       SSPSend
**
** Descriptions:        Send a block of data to the SSP port, the
**                      first parameter is the buffer pointer, the 2nd
**                      parameter is the block length.
**
** parameters:          buffer pointer, and the block length
** Returned value:      None
**
*****************************************************************************/
```

```
void SSPSend( uint32_t portnum, uint8_t *buf, uint32_t Length )
{
  uint32_t i;
  uint8_t Dummy = Dummy;

  for ( i = 0; i < Length; i++ )
  {
    if ( portnum == 0 )
      {
      /* Move on only if NOT busy and TX FIFO not full. */
        while ( (LPC_SSP0->SR & (SSPSR_TNF|SSPSR_BSY)) != SSPSR_TNF );
        LPC_SSP0->DR = *buf;
        buf++;
#if !LOOPBACK_MODE
        while ( (LPC_SSP0->SR & (SSPSR_BSY|SSPSR_RNE)) != SSPSR_RNE );
        /* Whenever a byte is written, MISO FIFO counter increments, Clear FIFO
        on MISO. Otherwise, when SSP0Receive() is called, previous data byte
        is left in the FIFO. */
        Dummy = LPC_SSP0->DR;
#else
        /* Wait until the Busy bit is cleared. */
        while ( LPC_SSP0->SR & SSPSR_BSY );
#endif
    }
    else if ( portnum == 1 )
      {
        /* Move on only if NOT busy and TX FIFO not full. */
        while ( (LPC_SSP1->SR & (SSPSR_TNF|SSPSR_BSY)) != SSPSR_TNF );
        LPC_SSP1->DR = *buf;
        buf++;
#if !LOOPBACK_MODE
        while ( (LPC_SSP1->SR & (SSPSR_BSY|SSPSR_RNE)) != SSPSR_RNE );
        /* Whenever a byte is written, MISO FIFO counter increments, Clear FIFO
        on MISO. Otherwise, when SSP0Receive() is called, previous data byte
        is left in the FIFO. */
        Dummy = LPC_SSP1->DR;
#else
        /* Wait until the Busy bit is cleared. */
        while ( LPC_SSP1->SR & SSPSR_BSY );
#endif
    }
  }
  return;
}

/*****************************************************************************
** Function name:       SSPSendReceive
** Descriptions:        the module will receive a block of data from
**                      the SSP1 only
** parameters:          send char
** Returned value:      received value
**
*****************************************************************************/
uint8_t SSP1SendReceive(uint8_t out){
      LPC_SSP1->DR =out;
      while(LPC_SSP1->SR & (1<<4));
      return LPC_SSP1->DR;
}
```

```
/*****************************************************************************
** Function name:       SSPReceive
** Descriptions:        the module will receive a block of data from
**                              the SSP, the 2nd parameter is the block
**                              length.
** parameters:                buffer pointer, and block length
** Returned value:            None
**
*****************************************************************************/
void SSPReceive( uint32_t portnum, uint8_t *buf, uint32_t Length )
{
  uint32_t i;

  for ( i = 0; i < Length; i++ )
  {
      /* As long as Receive FIFO is not empty, I can always receive. */
      /* If it's a loopback test, clock is shared for both TX and RX,
      no need to write dummy byte to get clock to get the data */
      /* if it's a peer-to-peer communication, SSPDR needs to be written
      before a read can take place. */
    if ( portnum == 0 )
    {
#if !LOOPBACK_MODE
#if SSP_SLAVE
        while ( !(LPC_SSP0->SR & SSPSR_RNE) );
#else
        LPC_SSP0->DR = 0xFF;
        /* Wait until the Busy bit is cleared */
        while ( (LPC_SSP0->SR & (SSPSR_BSY|SSPSR_RNE)) != SSPSR_RNE );
#endif
#else
        while ( !(LPC_SSP0->SR & SSPSR_RNE) );
#endif
        *buf++ = LPC_SSP0->DR;
    }
      else if ( portnum == 1 )
      {
#if !LOOPBACK_MODE
#if SSP_SLAVE
        while ( !(LPC_SSP1->SR & SSPSR_RNE) );
#else
        LPC_SSP1->DR = 0xFF;
        /* Wait until the Busy bit is cleared */
        while ( (LPC_SSP1->SR & (SSPSR_BSY|SSPSR_RNE)) != SSPSR_RNE );
#endif
#else
        while ( !(LPC_SSP1->SR & SSPSR_RNE) );
#endif
        *buf++ = LPC_SSP1->DR;
      }
  }
  return;
}

/*****************************************************************************
**                            End Of File
*****************************************************************************/
```