```
#ifndef __st
#define __st

#include <eddef.h>
#include <spi.h>

#define FLASH_TYPE           M25P40
#define ASSERT_ST_CS         P5OUT &= ~BIT4     // assert CS
#define DEASSERT_ST_CS       P5OUT |= BIT4    // clear CS
#define PAGE_ADDR(PAGE)      ((PAGE) << 8)
#define SECTOR_ADDR(SECTOR) ((SECTOR) << 15)
#define PAGE(VAL)            ((VAL) >> 8)             // return the page for an
address
#define SECTOR(VAL)          ((VAL) >> 15)            // return the sector for an
address
#define MFG                  0x20         // manufacturer ID
#define TYPE                 0x20         // manufacturer type
#define CAPACITY             0x13         // manufacturer capacity
#define PAGE_BYTES           0x00000100   // size of a page
#define PAGE_MASK            0x00000011   // mask for the offset into a page
#define SECTOR_BYTES         0x00010000   // size of a sector
#define MEMORY_BYTES         0x00080000   // size of memory
#define EVENT_ALLOCATION             16384
#define NUM_SECTOR           (MEMORY_BYTES / SECTOR_BYTES) // number of sectors
#define NUM_PAGE             (MEMORY_BYTES / PAGE_BYTES) // number of pages
#define EVENTS_PER_SECTOR    (SECTOR_BYTES / EVENT_ALLOCATION)
#define NUMBER_OF_EVENTS     (NUM_SECTOR * EVENTS_PER_SECTOR)
#define WIP                  BIT0
#define WREN                 0x06

ERROR_STATE st_init(void); // initialize the flash - verify that the flash
matches
ERROR_STATE WRDI(void); // write disable
ERROR_STATE RDID(UINT8 *man, UINT8 *type, UINT8 *capacity); // get the
manufacturer id
ERROR_STATE RDSR(UINT8 *status); // read the status register (poll for
completion of operation)
ERROR_STATE WRSR(UINT8 status); // write the status register
UINT32      READ(void *dest, UINT32 addr, UINT32 nBytes); // read data out
slower
ERROR_STATE FAST_READ(void *dest, UINT32 addr, UINT32 nBytes); // read data out
faster
UINT32      PP(UINT32 addr, void *src, UINT16 nBytes, BOOL wait); // page
program - need to watch pages on this operation - limit to 256 bytes
UINT32      BP(UINT32 addr, UINT8 value, BOOL wait); // program single byte in
the flash
ERROR_STATE WriteSPIFlash(void *src, UINT32 addr, UINT16 nBytes, BOOL wait); //
Generic write process for the flash
ERROR_STATE SE(UINT32 addr, BOOL wait);
ERROR_STATE BE(void); // bulk erase
ERROR_STATE DP(void); // go into deep power down
ERROR_STATE RES(UINT8 *sig); // release from power down
ERROR_STATE Page_Copy(UINT32 dest, UINT32 src, UINT32 nBytes); // limited copy
function - only copies full memory pages
BOOL FlashBusy(void);

#ifndef NDEBUG
void MemoryDump(UINT32 start, UINT32 nBytes);
```

```
#endif

#endif
```

```
#include <spi.h>

UINT8 *U1RXBuffer, *U1TXBuffer, *MsgEnd, *HeaderEnd;

ERROR_STATE SPI_init(void)
{
//-----------------------------------------------------------------------------
// Initializes the USART1 for SPI operation
//-----------------------------------------------------------------------------
// Configure the ports for CS and SPI
  P4SEL &= 0b00011111;
  P4DIR |= 0b11100000;
  P4OUT |= 0b11100000;
  P5SEL = 0x0E;
  P5DIR |= 0b11111010;
  P5OUT = 0b11110000;
// Configure the SPI
  ME2 |= USPIE1; // enable spi module
  U1CTL |= (CHAR | SYNC | MM | SWRST); // 8 characters, master mode, spi
  U1TCTL |= (SSEL1 | STC); // use smclk as the clock, Mode 0
  U1BR0 = 0x02; // rate = SMCLK /4
  U1BR1 = 0x00;
  U1MCTL = 0x00;
  U1CTL &= ~SWRST; // bring uart out of reset
  U1RXBuffer == NULL;
  return OK;
}

void ReadSPIByteBlock(void *dest, UINT16 nBytes)
{
//-----------------------------------------------------------------------------
// Send block of data across SPI bus - ignore responses
//-----------------------------------------------------------------------------
  UINT8    *p = dest;
  volatile UINT8 *end;
  volatile UINT16 i;

  SPIIRQOFF;
  *p = U1RXBUF;
  end = (UINT8 *)dest + nBytes;
  while (p < end)
  {
    while ((IFG2 & UTXIFG1) == 0) {}
    U1TXBUF = 0x00;
    while ((IFG2 & URXIFG1) == 0) {}
    *p++ = U1RXBUF;
  }
}

void SendSPIByteBlock(void *src, UINT16 nBytes, UINT8 terminate)
{
//-----------------------------------------------------------------------------
// Send block of data across SPI bus - ignore responses
//-----------------------------------------------------------------------------
  UINT8    *p = src;
  volatile UINT8 *end;
  volatile UINT16 i;
```

```
  U1RXBuffer == NULL;
  end = (UINT8 *)src + nBytes;
  while (p < end)
  {
    while ((IFG2 & UTXIFG1) == 0) {}
    U1TXBUF = *p++;
  }
  if (terminate == TERMINATE_ON_TX_EMPTY) while ((U1TCTL & TXEPT) == 0) {}
}

UINT8 SendSPIByte(UINT8 value, UINT8 terminate)
{
  UINT8 response = 0;

  switch (terminate)
  {
    case TERMINATE_ON_RECEIVE:
      U1RXBuffer = &response;
      SPIIRQON;
      U1TXBUF = value;
      while (U1RXBuffer == &response) {}
      SPIIRQOFF;
      break;
    case TERMINATE_ON_TX_READY:
      U1RXBuffer == NULL;
      U1TXBUF = value;
      while ((U1TCTL & TXEPT) == 0) {}
      break;
    case TERMINATE_ON_TX_EMPTY:
      U1RXBuffer == NULL;
      U1TXBUF = value;
      while ((U1TCTL & TXEPT) == 0) {}
      break;
  }
  return response;
}

void USART1_rx (void) __interrupt[USART1RX_VECTOR]
{
  if (U1RXBuffer != NULL) *U1RXBuffer++ = U1RXBUF;
}
```