```c
/****************************************************************************
 Module
     DM_Display.c
 Description
     Source file for the Dot Matrix LED Hardware Abstraction Layer
     used in ME218
 Notes
     This is the prototype. Students will re-create this functionality
 History
 When           Who     What/Why
 -------------- ---     --------
  10/03/21 12:32 jec    started coding
*****************************************************************************/
/*----------------------------- Include Files -----------------------------*/
#include <xc.h>
#include <stdbool.h>
#include "PIC32_SPI_HAL.h"
#include "DM_Display.h"



/*----------------------------- Module Defines ----------------------------*/
#define NumModules 12 //4 12
#define NUM_ROWS   24//8 24
#define NUM_COLS   32
#define NUM_DISPLAYS 3

#define NumModulesPerDisplay (NumModules / NUM_DISPLAYS)//4
#define NumColumsPerModule (NUM_COLS/NumModulesPerDisplay) //8
#define NumRowsPerDisplay (NUM_ROWS/NUM_DISPLAYS) //8



#define NUM_ROWS_IN_FONT 5
#define DM_START_SHUTDOWN 0x0C00
#define DM_END_SHUTDOWN   0x0C01
#define DM_DISABLE_CODEB  0x0900
#define DM_ENABLE_SCAN    0x0B07
#define DM_SET_BRIGHT     0x0A00

/*------------------------------ Module Types -----------------------------*/
// this union definition assumes that the display is made up of 4 modules
// 4 modules x 8 bits/module = 32 bits total
// this union allows us to easily scroll the whole buffer, using the uint32_t
// while picking out the individual bytes to send them to the controllers
typedef union{
    uint32_t FullRow;
    uint8_t ByBytes[4];
}DM_Row_t;

typedef enum { DM_StepStartShutdown = 0, DM_StepFillBufferZeros,
               DM_StepDisableCodeB, DM_StepEnableScanAll, DM_StepSetBrighness,
               DM_StepCopyBuffer2Display, DM_StepEndShutdown
}InitStep_t;

/*---------------------------- Module Functions ---------------------------*/
static void sendCmd( uint16_t Cmd2Send );
static void sendFrame( uint8_t RowNum );

/*---------------------------- Module Variables ---------------------------*/
```

```
// We make the display buffer from an array of these unions, one for each
// row in the display
static DM_Row_t DM_Display[NUM_ROWS];

// this is the state variable for tracking init steps
static InitStep_t CurrentInitStep =  DM_StepStartShutdown;

// In order to keep up with the display at 10MHz, the bit reverse operation
// must be as fast as possible, hence the look-up table approach is the only
// solution that will work with the SPI at 10MHz
// This bit reversal table comes from
//
https://stackoverflow.com/questions/746171/efficient-algorithm-for-bit-reversal-from-ms
b-lsb-to-lsb-msb-in-c
// usage: BitReversedValue = BitReverseTable256[OriginalByteValue];
//
static const uint8_t BitReverseTable256[] =
{
  0x00, 0x80, 0x40, 0xC0, 0x20, 0xA0, 0x60, 0xE0, 0x10, 0x90, 0x50, 0xD0, 0x30, 0xB0,
0x70, 0xF0,
  0x08, 0x88, 0x48, 0xC8, 0x28, 0xA8, 0x68, 0xE8, 0x18, 0x98, 0x58, 0xD8, 0x38, 0xB8,
0x78, 0xF8,
  0x04, 0x84, 0x44, 0xC4, 0x24, 0xA4, 0x64, 0xE4, 0x14, 0x94, 0x54, 0xD4, 0x34, 0xB4,
0x74, 0xF4,
  0x0C, 0x8C, 0x4C, 0xCC, 0x2C, 0xAC, 0x6C, 0xEC, 0x1C, 0x9C, 0x5C, 0xDC, 0x3C, 0xBC,
0x7C, 0xFC,
  0x02, 0x82, 0x42, 0xC2, 0x22, 0xA2, 0x62, 0xE2, 0x12, 0x92, 0x52, 0xD2, 0x32, 0xB2,
0x72, 0xF2,
  0x0A, 0x8A, 0x4A, 0xCA, 0x2A, 0xAA, 0x6A, 0xEA, 0x1A, 0x9A, 0x5A, 0xDA, 0x3A, 0xBA,
0x7A, 0xFA,
  0x06, 0x86, 0x46, 0xC6, 0x26, 0xA6, 0x66, 0xE6, 0x16, 0x96, 0x56, 0xD6, 0x36, 0xB6,
0x76, 0xF6,
  0x0E, 0x8E, 0x4E, 0xCE, 0x2E, 0xAE, 0x6E, 0xEE, 0x1E, 0x9E, 0x5E, 0xDE, 0x3E, 0xBE,
0x7E, 0xFE,
  0x01, 0x81, 0x41, 0xC1, 0x21, 0xA1, 0x61, 0xE1, 0x11, 0x91, 0x51, 0xD1, 0x31, 0xB1,
0x71, 0xF1,
  0x09, 0x89, 0x49, 0xC9, 0x29, 0xA9, 0x69, 0xE9, 0x19, 0x99, 0x59, 0xD9, 0x39, 0xB9,
0x79, 0xF9,
  0x05, 0x85, 0x45, 0xC5, 0x25, 0xA5, 0x65, 0xE5, 0x15, 0x95, 0x55, 0xD5, 0x35, 0xB5,
0x75, 0xF5,
  0x0D, 0x8D, 0x4D, 0xCD, 0x2D, 0xAD, 0x6D, 0xED, 0x1D, 0x9D, 0x5D, 0xDD, 0x3D, 0xBD,
0x7D, 0xFD,
  0x03, 0x83, 0x43, 0xC3, 0x23, 0xA3, 0x63, 0xE3, 0x13, 0x93, 0x53, 0xD3, 0x33, 0xB3,
0x73, 0xF3,
  0x0B, 0x8B, 0x4B, 0xCB, 0x2B, 0xAB, 0x6B, 0xEB, 0x1B, 0x9B, 0x5B, 0xDB, 0x3B, 0xBB,
0x7B, 0xFB,
  0x07, 0x87, 0x47, 0xC7, 0x27, 0xA7, 0x67, 0xE7, 0x17, 0x97, 0x57, 0xD7, 0x37, 0xB7,
0x77, 0xF7,
  0x0F, 0x8F, 0x4F, 0xCF, 0x2F, 0xAF, 0x6F, 0xEF, 0x1F, 0x9F, 0x5F, 0xDF, 0x3F, 0xBF,
0x7F, 0xFF
};


/*------------------------------- Module Code ------------------------------*/
/****************************************************************************
 Function
  DM_TakeInitDisplayStep

  Description
  Initializes the MAX7219 4-module display performing 1 step for each call:
    First, bring put it in shutdown to disable all displays, return false
    Next fill the display RAM with Zeros to insure blanked, return false
    Then Disable Code B decoding for all digits, return false
    Then, enable scanning for all digits, return false
    The next setup step is to set the brightness to minimum, return false
    Copy our display buffer to the display, return false
    Finally, bring it out of shutdown and return true
****************************************************************************/
bool DM_TakeInitDisplayStep( void )
{
    bool ReturnVal = false;

    switch (CurrentInitStep)
    {
    case DM_StepStartShutdown:
      sendCmd(DM_START_SHUTDOWN); //First, bring put it in shutdown to disable all
displays
      CurrentInitStep++; //move on to next step
      break;

    case DM_StepFillBufferZeros:
      DM_ClearDisplayBuffer();
      CurrentInitStep++;  // move on to next step
      break;

    case DM_StepDisableCodeB:
      sendCmd(DM_DISABLE_CODEB); // Next Disable Code B decoding for all digits
      CurrentInitStep++;  // move on to next step
      break;

    case DM_StepEnableScanAll:
      sendCmd(DM_ENABLE_SCAN); // Then, enable scanning for all digits
      CurrentInitStep++; // move on to next step
      break;

    case DM_StepSetBrighness:
      sendCmd(DM_SET_BRIGHT); // The next setup step is to set the brightness to
minimum
      CurrentInitStep++; // move on to next step
      break;

    case DM_StepCopyBuffer2Display: // copy our display buffer to the display
      if (true == DM_TakeDisplayUpdateStepFrame())
      {
        CurrentInitStep++; // move on to next step
      }
      else
      {}
      break;

    case DM_StepEndShutdown:
        sendCmd(DM_END_SHUTDOWN); // Finally, bring it out of shutdown
        CurrentInitStep = 0; // prepare for a re-init
        ReturnVal = true; //let the caller know that we are done
        break;

    default:
      break;
    }
    return ReturnVal;
}

/****************************************************************************
 Function
  DM_TakeDisplayUpdateStepFrame

 Description
  Copies the contents of the display buffer to the MAX7219 controllers 1 row
  per call.
****************************************************************************/
bool DM_TakeDisplayUpdateStepFrame( void )
{
    bool ReturnVal = false;
    static int8_t WhichRow = 0;

    sendFrame(WhichRow);//send a row of data

    WhichRow++;//increment which row

    if (WhichRow >= 8)//check when we are done sending rows goes here
    {
      ReturnVal = true; // show we are done
      WhichRow = 0; // set up for next update
    }

    return ReturnVal;
}



/****************************************************************************
 Function
  DM_ClearDisplayBuffer

 Description
  Clears the contents of the display buffer by filling it with zeros.
****************************************************************************/
void DM_ClearDisplayBuffer( void )
{


  for (uint8_t rowIndex = 0; rowIndex < NUM_ROWS; rowIndex++)
  {
    DM_Display[rowIndex].FullRow = 0;

  }
}

/****************************************************************************
 Function
  DM_PutDataIntoBufferRow

 Description
  Copies the raw data from the Data2Insert parameter into the specified row
  of the frame buffer
****************************************************************************/
bool DM_PutDataIntoBufferRow( uint32_t Data2Insert, uint8_t WhichRow)
{
  bool ReturnVal = true;
  if (WhichRow < NUM_ROWS){// test for legal row
      DM_Display[WhichRow].FullRow = Data2Insert;// legal row, so stuff the data into
the buffer
  }else{
      ReturnVal = false;
  }
  return ReturnVal;
}

/****************************************************************************
 Function
  DM_QueryRowData

 Description
  copies the contents of the specified row of the frame buffer into the
 location pointed to by pReturnValue
****************************************************************************/
bool DM_QueryRowData( uint8_t RowToQuery, uint32_t * pReturnValue)
{
  bool ReturnVal = true;
  if (RowToQuery < NUM_ROWS){// test for legal row
      *pReturnValue = DM_Display[RowToQuery].FullRow;// legal row, so grab the data
from the buffer
  } else{
      ReturnVal = false;
  }
  return ReturnVal;


}


//*********************************
// private functions
//*********************************

/****************************************************************************
 Function
 sendCmd

 Description
  Send a single command to all 4 modules and waits for the SS to rise to
  indicate completion.
****************************************************************************/
static void sendCmd( uint16_t Cmd2Send )
{
    uint8_t index;
    for (index = 0; index < 2*NumModulesPerDisplay; index++)//Send 8 Commands
    {
        SPIOperate_SPI1_Send16( Cmd2Send );
    }

    //after 8 sends the buffer is full
    for (index = 0; index < (NumModules-8-1); index++) //send 3 commands
    {
        while (SPI1STATbits.SPITBF){;}//wait until buffer has space
        SPIOperate_SPI1_Send16( Cmd2Send );//send command
    }
    while (SPI1STATbits.SPITBF){;}//wait until buffer has space
    SPIOperate_SPI1_Send16Wait(  Cmd2Send ); //send final command and change SS line to 
latch data
}

/****************************************************************************
 Function
 sendFrame

 Description
  Sends a row of data to the NUM_MODULES cluster. Translates from the logical
 row number to the MAX7219 row numbers (mirrors)
****************************************************************************/
static void sendFrame( uint8_t RowNum ) {
    // The rows on the display are mirrored relative to the rows in the memory
    uint8_t newRowNum = NumRowsPerDisplay - RowNum -1;//swaps them top to bottom

    int8_t index = 0;
     // Third display
    for (index = 0; index < (NumModulesPerDisplay); index++) {//send data 4 times
        while (SPI1STATbits.SPITBF){;}//wait until buffer has space
        SPIOperate_SPI1_Send16( (((uint16_t)RowNum+1)<<8) |
                BitReverseTable256[DM_Display[(3*NumRowsPerDisplay) - newRowNum - 
1].ByBytes[index]]);
    }

    //second display
    for (index = NumModulesPerDisplay-1; index > -1; index--) {//send data 4 times
        while (SPI1STATbits.SPITBF){;} //wait until buffer has space
        //second display is reversed and upside down so the send looks slightly 
different
        SPIOperate_SPI1_Send16( (((uint16_t)RowNum+1)<<8) |
               DM_Display[(2*NumRowsPerDisplay)-RowNum-1].ByBytes[index]);
    }

    //first display
     for (index = 0; index < (NumModulesPerDisplay-1); index++) {//send data 3 times
        while (SPI1STATbits.SPITBF){;}//wait until buffer has space
        SPIOperate_SPI1_Send16( (((uint16_t)RowNum+1)<<8) |

BitReverseTable256[DM_Display[(1*NumRowsPerDisplay)-newRowNum-1].ByBytes[index]]);
    }

    while (SPI1STATbits.SPITBF){;}//wait until buffer has space
    //send the final pice of data and use the SS line to latch the data
    SPIOperate_SPI1_Send16Wait( (((uint16_t)RowNum+1)<<8) |

BitReverseTable256[DM_Display[(1*NumRowsPerDisplay)-newRowNum-1].ByBytes[index]]);

}

//adds the dog and bone to the display buffer
bool DM_DrawDogAndBone(int8_t dogX, int8_t dogY, int8_t boneX, int8_t boneY){
    bool returnVal = true;

    DM_ClearDisplayBuffer();//clear the display
    //draws the dog
    DM_Display[dogY].FullRow = (1<<dogX) | (1<< (dogX-2));//legs
    DM_Display[dogY+1].FullRow = (1<<dogX) | (1<< (dogX-1)) | (1<<(dogX-2));//body
    DM_Display[dogY+2].FullRow = (1<< (dogX-2));//head
    DM_Display[dogY+3].FullRow = (1<<(dogX-1)) | (1<< (dogX-3));//ears
    //draws the bone
    DM_Display[boneY].FullRow = DM_Display[boneY].FullRow | (1<<boneX-2);//first bone
row
    DM_Display[boneY+1].FullRow = DM_Display[boneY+1].FullRow | (1<< (boneX-2)) | (1<<
(boneX-3));//second bone row
    DM_Display[boneY+2].FullRow = DM_Display[boneY+2].FullRow | ((1<<boneX) | (1<<
(boneX-1)));//third bone row
    DM_Display[boneY+3].FullRow = DM_Display[boneY+3].FullRow | (1<<
(boneX-1));//fourth bone row

     return returnVal;
}

//adds the cat to the display buffer
bool DM_DrawCat(int8_t catX, int8_t catY){
  bool returnVal = true;
  DM_Display[catY].FullRow = DM_Display[catY].FullRow | ((1<<(catX-1)) | (1<<
(catX-2)));//first cat row
  DM_Display[catY+1].FullRow = DM_Display[catY+1].FullRow | ((1<<catX) | (1<< (catX-1))
| (1<< (catX-2)) | (1<< (catX-3)) );//second cat row
  DM_Display[catY+2].FullRow = DM_Display[catY+2].FullRow | ((1<<catX) | (1<< (catX-1))
| (1<< (catX-2)) | (1<< (catX-3)) );//third cat row
  DM_Display[catY+3].FullRow = DM_Display[catY+3].FullRow | ((1<<catX) | (1<<
(catX-3)));//fourth cat row


  return returnVal;
}

//adds the given frame to the display buffer
bool DM_DrawInstructions(uint32_t frame[24])
{
    bool returnVal = true;
    DM_ClearDisplayBuffer();//clear buffer
    for (uint8_t ii = 0; ii < NUM_ROWS; ii++)//loop through each row of instructions
    {
        DM_Display[ii].FullRow = frame[NUM_ROWS - 1 - ii];//set row in the display
buffer to the instruction row
    }
    return returnVal;
}
```