

```

//*****
// rit128x96x4.c - Driver for the RIT 128x96x4 graphical OLED display.
//
// Copyright (c) 2007 Luminary Micro, Inc. All rights reserved.
//
// Software License Agreement
//
// Luminary Micro, Inc. (LMI) is supplying this software for use solely and
// exclusively on LMI's microcontroller products.
//
// The software is owned by LMI and/or its suppliers, and is protected under
// applicable copyright laws. All rights are reserved. Any use in violation
// of the foregoing restrictions may subject the user to criminal sanctions
// under applicable laws, as well as to civil liability for the breach of the
// terms and conditions of this license.
//
// THIS SOFTWARE IS PROVIDED "AS IS". NO WARRANTIES, WHETHER EXPRESS, IMPLIED
// OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF
// MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE APPLY TO THIS SOFTWARE.
// LMI SHALL NOT, IN ANY CIRCUMSTANCES, BE LIABLE FOR SPECIAL, INCIDENTAL, OR
// CONSEQUENTIAL DAMAGES, FOR ANY REASON WHATSOEVER.
//
// This is part of revision 1582 of the Stellaris Peripheral Driver Library.
//
//*****
//!
//! \addtogroup ek_lm3s8962_api
//! @{
//!
//*****
#include "hw_ssi.h"
#include "hw_memmap.h"
#include "hw_sysctl.h"
#include "hw_types.h"
#include "debug.h"
#include "gpio.h"
#include "ssi.h"
#include "sysctl.h"
#include "rit128x96x4.h"

//*****
// Macros that define the peripheral, port, and pin used for the OLEDDC
// panel control signal.
//
//*****
#define SYSCTL_PERIPH_GPIO_OLEDDC    SYSCTL_PERIPH_GPIOA
#define GPIO_OLEDDC_BASE              GPIO_PORTA_BASE
#define GPIO_OLEDDC_PIN               GPIO_PIN_6
#define GPIO_OLEDEN_PIN               GPIO_PIN_7

//*****
// Flag to indicate if SSI port is enabled for display usage.
//
//*****
static volatile tBoolean g_bSSIEnabled = false;

```

```

// Buffer for storing sequences of command and data for the display.
//
//*****
static unsigned char g_pucBuffer[8];

//*****
//
// Define the SSD1329 128x96x4 Remap Setting(s). This will be used in
// several places in the code to switch between vertical and horizontal
// address incrementing. Note that the controller support 128 rows while
// the RIT display only uses 96.
//
// The Remap Command (0xA0) takes one 8-bit parameter. The parameter is
// defined as follows.
//
// Bit 7: Reserved
// Bit 6: Disable(0)/Enable(1) COM Split Odd Even
//         When enabled, the COM signals are split Odd on one side, even on
//         the other. Otherwise, they are split 0-63 on one side, 64-127 on
//         the other.
// Bit 5: Reserved
// Bit 4: Disable(0)/Enable(1) COM Remap
//         When Enabled, ROW 0-127 map to COM 127-0 (i.e. reverse row order)
// Bit 3: Reserved
// Bit 2: Horizontal(0)/Vertical(1) Address Increment
//         When set, data RAM address will increment along the column rather
//         than along the row.
// Bit 1: Disable(0)/Enable(1) Nibble Remap
//         When enabled, the upper and lower nibbles in the DATA bus for access
//         to the data RAM are swapped.
// Bit 0: Disable(0)/Enable(1) Column Address Remap
//         When enabled, DATA RAM columns 0-63 are remapped to Segment Columns
//         127-0.
//
//*****
#define RIT_INIT_REMAP      0x52 // app note says 0x51
#define RIT_INIT_OFFSET      0x00
static const unsigned char g_pucRIT128x96x4VerticalInc[] = { 0xA0, 0x56 };
static const unsigned char g_pucRIT128x96x4HorizontalInc[] = { 0xA0, 0x52 };

//*****
//
// A 5x7 font (in a 6x8 cell, where the sixth column is omitted from this
// table) for displaying text on the OLED display. The data is organized as
// bytes from the left column to the right column, with each byte containing
// the top row in the LSB and the bottom row in the MSB.
//
// Note: This is the same font data that is used in the EK-LM3S811
// osram96x16x1 driver. The single bit-per-pixel is expaned in the StringDraw
// function to the appropriate four bit-per-pixel gray scale format.
//
//*****
static const unsigned char g_pucFont[96][5] =
{
    { 0x00, 0x00, 0x00, 0x00, 0x00 }, // "
    { 0x00, 0x00, 0x4f, 0x00, 0x00 }, // !
    { 0x00, 0x07, 0x00, 0x07, 0x00 }, // "
    { 0x14, 0x7f, 0x14, 0x7f, 0x14 }, // #
    { 0x24, 0x2a, 0x7f, 0x2a, 0x12 }, // $
    { 0x23, 0x13, 0x08, 0x64, 0x62 }, // %
    { 0x36, 0x49, 0x55, 0x22, 0x50 }, // &
    { 0x00, 0x05, 0x03, 0x00, 0x00 }, // '
    { 0x00, 0x1c, 0x22, 0x41, 0x00 }, // (
    { 0x00, 0x41, 0x22, 0x1c, 0x00 }, // )
}

```

```

{ 0x14, 0x08, 0x3e, 0x08, 0x14 }, // *
{ 0x08, 0x08, 0x3e, 0x08, 0x08 }, // +
{ 0x00, 0x50, 0x30, 0x00, 0x00 }, // ,
{ 0x08, 0x08, 0x08, 0x08, 0x08 }, // -
{ 0x00, 0x60, 0x60, 0x00, 0x00 }, // .
{ 0x20, 0x10, 0x08, 0x04, 0x02 }, // /
{ 0x3e, 0x51, 0x49, 0x45, 0x3e }, // 0
{ 0x00, 0x42, 0x7f, 0x40, 0x00 }, // 1
{ 0x42, 0x61, 0x51, 0x49, 0x46 }, // 2
{ 0x21, 0x41, 0x45, 0x4b, 0x31 }, // 3
{ 0x18, 0x14, 0x12, 0x7f, 0x10 }, // 4
{ 0x27, 0x45, 0x45, 0x45, 0x39 }, // 5
{ 0x3c, 0x4a, 0x49, 0x49, 0x30 }, // 6
{ 0x01, 0x71, 0x09, 0x05, 0x03 }, // 7
{ 0x36, 0x49, 0x49, 0x49, 0x36 }, // 8
{ 0x06, 0x49, 0x49, 0x29, 0x1e }, // 9
{ 0x00, 0x36, 0x36, 0x00, 0x00 }, // :
{ 0x00, 0x56, 0x36, 0x00, 0x00 }, // ;
{ 0x08, 0x14, 0x22, 0x41, 0x00 }, // <
{ 0x14, 0x14, 0x14, 0x14, 0x14 }, // =
{ 0x00, 0x41, 0x22, 0x14, 0x08 }, // >
{ 0x02, 0x01, 0x51, 0x09, 0x06 }, // ?
{ 0x32, 0x49, 0x79, 0x41, 0x3e }, // @
{ 0x7e, 0x11, 0x11, 0x11, 0x7e }, // A
{ 0x7f, 0x49, 0x49, 0x49, 0x36 }, // B
{ 0x3e, 0x41, 0x41, 0x41, 0x22 }, // C
{ 0x7f, 0x41, 0x41, 0x22, 0x1c }, // D
{ 0x7f, 0x49, 0x49, 0x49, 0x41 }, // E
{ 0x7f, 0x09, 0x09, 0x09, 0x01 }, // F
{ 0x3e, 0x41, 0x49, 0x49, 0x7a }, // G
{ 0x7f, 0x08, 0x08, 0x08, 0x7f }, // H
{ 0x00, 0x41, 0x7f, 0x41, 0x00 }, // I
{ 0x20, 0x40, 0x41, 0x3f, 0x01 }, // J
{ 0x7f, 0x08, 0x14, 0x22, 0x41 }, // K
{ 0x7f, 0x40, 0x40, 0x40, 0x40 }, // L
{ 0x7f, 0x02, 0x0c, 0x02, 0x7f }, // M
{ 0x7f, 0x04, 0x08, 0x10, 0x7f }, // N
{ 0x3e, 0x41, 0x41, 0x41, 0x3e }, // O
{ 0x7f, 0x09, 0x09, 0x09, 0x06 }, // P
{ 0x3e, 0x41, 0x51, 0x21, 0x5e }, // Q
{ 0x7f, 0x09, 0x19, 0x29, 0x46 }, // R
{ 0x46, 0x49, 0x49, 0x49, 0x31 }, // S
{ 0x01, 0x01, 0x7f, 0x01, 0x01 }, // T
{ 0x3f, 0x40, 0x40, 0x40, 0x3f }, // U
{ 0x1f, 0x20, 0x40, 0x20, 0x1f }, // V
{ 0x3f, 0x40, 0x38, 0x40, 0x3f }, // W
{ 0x63, 0x14, 0x08, 0x14, 0x63 }, // X
{ 0x07, 0x08, 0x70, 0x08, 0x07 }, // Y
{ 0x61, 0x51, 0x49, 0x45, 0x43 }, // Z
{ 0x00, 0x7f, 0x41, 0x41, 0x00 }, // [
{ 0x02, 0x04, 0x08, 0x10, 0x20 }, // "\"
{ 0x00, 0x41, 0x41, 0x7f, 0x00 }, // ]
{ 0x04, 0x02, 0x01, 0x02, 0x04 }, // ^
{ 0x40, 0x40, 0x40, 0x40, 0x40 }, // =
{ 0x00, 0x01, 0x02, 0x04, 0x00 }, // \
{ 0x20, 0x54, 0x54, 0x54, 0x78 }, // a
{ 0x7f, 0x48, 0x44, 0x44, 0x38 }, // b
{ 0x38, 0x44, 0x44, 0x44, 0x20 }, // c
{ 0x38, 0x44, 0x44, 0x48, 0x7f }, // d
{ 0x38, 0x54, 0x54, 0x54, 0x18 }, // e
{ 0x08, 0x7e, 0x09, 0x01, 0x02 }, // f
{ 0x0c, 0x52, 0x52, 0x52, 0x3e }, // g
{ 0x7f, 0x08, 0x04, 0x04, 0x78 }, // h
{ 0x00, 0x44, 0x7d, 0x40, 0x00 }, // i

```

```

{ 0x20, 0x40, 0x44, 0x3d, 0x00 }, // j
{ 0x7f, 0x10, 0x28, 0x44, 0x00 }, // k
{ 0x00, 0x41, 0x7f, 0x40, 0x00 }, // l
{ 0x7c, 0x04, 0x18, 0x04, 0x78 }, // m
{ 0x7c, 0x08, 0x04, 0x04, 0x78 }, // n
{ 0x38, 0x44, 0x44, 0x44, 0x38 }, // o
{ 0x7c, 0x14, 0x14, 0x14, 0x08 }, // p
{ 0x08, 0x14, 0x14, 0x18, 0x7c }, // q
{ 0x7c, 0x08, 0x04, 0x04, 0x08 }, // r
{ 0x48, 0x54, 0x54, 0x54, 0x20 }, // s
{ 0x04, 0x3f, 0x44, 0x40, 0x20 }, // t
{ 0x3c, 0x40, 0x40, 0x20, 0x7c }, // u
{ 0x1c, 0x20, 0x40, 0x20, 0x1c }, // v
{ 0x3c, 0x40, 0x30, 0x40, 0x3c }, // w
{ 0x44, 0x28, 0x10, 0x28, 0x44 }, // x
{ 0x0c, 0x50, 0x50, 0x50, 0x3c }, // y
{ 0x44, 0x64, 0x54, 0x4c, 0x44 }, // z
{ 0x00, 0x08, 0x36, 0x41, 0x00 }, // {
{ 0x00, 0x00, 0x7f, 0x00, 0x00 }, // |
{ 0x00, 0x41, 0x36, 0x08, 0x00 }, // }
{ 0x02, 0x01, 0x02, 0x04, 0x02 }, // ~
{ 0x02, 0x01, 0x02, 0x04, 0x02 }, // ~
};

//*****
// The sequence of commands used to initialize the SSD1329 controller. Each
// command is described as follows: there is a byte specifying the number of
// bytes in the command sequence, followed by that many bytes of command data.
// Note: This initialization sequence is derived from RIT App Note for
// the P14201. Values used are from the RIT app note, except where noted.
//
//*****
static const unsigned char g_pucRIT128x96x4Init[] =
{
    //
    // Unlock commands
    //
    3, 0xFD, 0x12, 0xe3,

    //
    // Display off
    //
    2, 0xAE, 0xe3,

    //
    // Icon off
    //
    3, 0x94, 0, 0xe3,

    //
    // Multiplex ratio
    //
    3, 0xA8, 95, 0xe3,

    //
    // Contrast
    //
    3, 0x81, 0xb7, 0xe3,

    //
    // Pre-charge current
    //
    3, 0x82, 0x3f, 0xe3,
}

```

```
//  
// Display Re-map  
//  
3, 0xA0, RIT_INIT_REMAP, 0xe3,  
  
//  
// Display Start Line  
//  
3, 0xA1, 0, 0xe3,  
  
//  
// Display Offset  
//  
3, 0xA2, RIT_INIT_OFFSET, 0xe3,  
  
//  
// Display Mode Normal  
//  
2, 0xA4, 0xe3,  
  
//  
// Phase Length  
//  
3, 0xB1, 0x11, 0xe3,  
  
//  
// Frame frequency  
//  
3, 0xB2, 0x23, 0xe3,  
  
//  
// Front Clock Divider  
//  
3, 0xB3, 0xe2, 0xe3,  
  
//  
// Set gray scale table. App note uses default command:  
// 2, 0xB7, 0xe3  
// This gray scale attempts some gamma correction to reduce the  
// the brightness of the low levels.  
//  
17, 0xB8, 1, 2, 3, 4, 5, 6, 8, 10, 12, 14, 16, 19, 22, 26, 30, 0xe3,  
  
//  
// Second pre-charge period. App note uses value 0x04.  
//  
3, 0xBB, 0x01, 0xe3,  
  
//  
// Pre-charge voltage  
//  
3, 0xBC, 0x3f, 0xe3,  
  
//  
// Display ON  
//  
2, 0xAF, 0xe3,  
};  
//*********************************************************************  
//  
// ! internal  
//!
```

```

///! Write a sequence of command bytes to the SSD1329 controller.
///
///! The data is written in a polled fashion; this function will not return
///! until the entire byte sequence has been written to the controller.
///
///! \return None.
///
//*****
static void
RITWriteCommand(const unsigned char *pucBuffer, unsigned long ulCount)
{
    unsigned long ulTemp;

    //
    // Return if SSI port is not enabled for RIT display.
    //
    if(!g_bSSIEnabled)
    {
        return;
    }

    //
    // Clear the command/control bit to enable command mode.
    //
    GPIOPinWrite(GPIO_OLEDDC_BASE, GPIO_OLEDDC_PIN, 0);

    //
    // Loop while there are more bytes left to be transferred.
    //
    while(ulCount != 0)
    {
        //
        // Write the next byte to the controller.
        //
        SSIDataPut(SSIO_BASE, *pucBuffer++);

        //
        // Dummy read to drain the fifo and time the GPIO signal.
        //
        SSIDataGet(SSIO_BASE, &ulTemp);

        //
        // Decrement the BYTE counter.
        //
        ulCount--;
    }
}

//*****
//!
///! \internal
///!
///! Write a sequence of data bytes to the SSD1329 controller.
///!
///! The data is written in a polled fashion; this function will not return
///! until the entire byte sequence has been written to the controller.
///
///! \return None.
///
//*****
static void
RITWriteData(const unsigned char *pucBuffer, unsigned long ulCount)
{
    unsigned long ulTemp;

```

```
//  
// Return if SSI port is not enabled for RIT display.  
//  
if(!_g_bSSIEnabled)  
{  
    return;  
}  
  
//  
// Set the command/control bit to enable data mode.  
//  
GPIOPinWrite(GPIO_OLEDDC_BASE, GPIO_OLEDDC_PIN, GPIO_OLEDDC_PIN);  
  
//  
// Loop while there are more bytes left to be transferred.  
//  
while(ulCount != 0)  
{  
    //  
    // Write the next byte to the controller.  
    //  
    SSIDataPut(SSI0_BASE, *pucBuffer++);  
  
    //  
    // Dummy read to drain the fifo and time the GPIO signal.  
    //  
    SSIDataGet(SSI0_BASE, &ulTemp);  
  
    //  
    // Decrement the BYTE counter.  
    //  
    ulCount--;  
}  
}  
  
*****  
//  
// Clears the OLED display.  
//  
// This function will clear the display RAM. All pixels in the display will  
// be turned off.  
//  
// \return None.  
//  
*****  
void  
RIT128x96x4Clear(void)  
{  
    static const unsigned char pucCommand1[] = { 0x15, 0, 63 };  
    static const unsigned char pucCommand2[] = { 0x75, 0, 127 };  
    unsigned long ulRow, ulColumn;  
  
    //  
    // Clear out the buffer used for sending bytes to the display.  
    //  
    *(unsigned long *)&g_pucBuffer[0] = 0;  
    *(unsigned long *)&g_pucBuffer[4] = 0;  
  
    //  
    // Set the window to fill the entire display.  
    //  
    RITWriteCommand(pucCommand1, sizeof(pucCommand1));  
    RITWriteCommand(pucCommand2, sizeof(pucCommand2));
```

```

RITWriteCommand(g_pucRIT128x96x4HorizontalInc,
                sizeof(g_pucRIT128x96x4HorizontalInc));

//  

// Loop through the rows  

//  

for(ulRow = 0; ulRow < 96; ulRow++)  

{  

    //  

    // Loop through the columns. Each byte is two pixels,  

    // and the buffer hold 8 bytes, so 16 pixels are cleared  

    // at a time.  

    //  

    for(ulColumn = 0; ulColumn < 128; ulColumn += 8 * 2)  

    {  

        //  

        // Write 8 clearing bytes to the display, which will  

        // clear 16 pixels across.  

        //  

        RITWriteData(g_pucBuffer, sizeof(g_pucBuffer));  

    }  

}  

}

//*****  

//  

/// Displays a string on the OLED display.  

///  

/// \param pcStr is a pointer to the string to display.  

/// \param ulX is the horizontal position to display the string, specified in  

/// columns from the left edge of the display.  

/// \param uly is the vertical position to display the string, specified in  

/// rows from the top edge of the display.  

/// \param ucLevel is the 4-bit grey scale value to be used for displayed text.  

///  

/// This function will draw a string on the display. Only the ASCII characters  

/// between 32 (space) and 126 (tilde) are supported; other characters will  

/// result in random data being drawn on the display (based on whatever appears  

/// before/after the font in memory). The font is mono-spaced, so characters  

/// such as "i" and "l" have more white space around them than characters such  

/// as "m" or "w".  

///  

/// If the drawing of the string reaches the right edge of the display, no more  

/// characters will be drawn. Therefore, special care is not required to avoid  

/// supplying a string that is "too long" to display.  

///  

/// \note Because the OLED display packs 2 pixels of data in a single byte, the  

/// parameter \e ulX must be an even column number (e.g. 0, 2, 4, etc).  

///  

/// \return None.  

//  

//*****  

void  

RIT128x96x4StringDraw(const char *pcStr, unsigned long ulX,  

                      unsigned long uly, unsigned char ucLevel)  

{  

    unsigned long ulIdx1, ulIdx2;  

    unsigned char ucTemp;  

  

    //  

    // Check the arguments.  

    //  

    ASSERT(ulX < 128);  

    ASSERT((ulX & 1) == 0);

```

```

ASSERT(ulY < 96);
ASSERT(ucLevel < 16);

//
// Setup a window starting at the specified column and row, ending
// at the right edge of the display and 8 rows down (single character row).
//
g_pucBuffer[0] = 0x15;
g_pucBuffer[1] = ulX / 2;
g_pucBuffer[2] = 63;
RITWriteCommand(g_pucBuffer, 3);
g_pucBuffer[0] = 0x75;
g_pucBuffer[1] = ulY;
g_pucBuffer[2] = ulY + 7;
RITWriteCommand(g_pucBuffer, 3);
RITWriteCommand(g_pucRIT128x96x4VerticalInc,
                sizeof(g_pucRIT128x96x4VerticalInc));

//
// Loop while there are more characters in the string.
//
while(*pcStr != 0)
{
    //
    // Get a working copy of the current character and convert to an
    // index into the character bit-map array.
    //
    ucTemp = *pcStr;
    ucTemp &= 0x7F;
    if(ucTemp < ' ')
    {
        ucTemp = ' ';
    }
    else
    {
        ucTemp -= ' ';
    }

    //
    // Build and display the character buffer.
    //
    for(ulIdx1 = 0; ulIdx1 < 6; ulIdx1 += 2)
    {
        //
        // Convert two columns of 1-bit font data into a single data
        // byte column of 4-bit font data.
        //
        for(ulIdx2 = 0; ulIdx2 < 8; ulIdx2++)
        {
            g_pucBuffer[ulIdx2] = 0;
            if(g_pucFont[ucTemp][ulIdx1] & (1 << ulIdx2))
            {
                g_pucBuffer[ulIdx2] = (ucLevel << 4) & 0xf0;
            }
            if((ulIdx1 < 4) &&
               (g_pucFont[ucTemp][ulIdx1 + 1] & (1 << ulIdx2)))
            {
                g_pucBuffer[ulIdx2] |= (ucLevel << 0) & 0x0f;
            }
        }
        //
        // Send this byte column to the display.
        //
    }
}

```

```

        RITWriteData(g_pucBuffer, 8);
        ulX += 2;

        //
        // Return if the right side of the display has been reached.
        //
        if(ulX == 128)
        {
            return;
        }

        //
        // Advance to the next character.
        //
        pcStr++;
    }

//*****
//!
/// Displays an image on the OLED display.
//!
/// \param pucImage is a pointer to the image data.
/// \param ulX is the horizontal position to display this image, specified in
/// columns from the left edge of the display.
/// \param uly is the vertical position to display this image, specified in
/// rows from the top of the display.
/// \param ulWidth is the width of the image, specified in columns.
/// \param ulHeight is the height of the image, specified in rows.
//!
/// This function will display a bitmap graphic on the display. Because of the
/// format of the display RAM, the starting column (\e ulX) and the number of
/// columns (\e ulWidth) must be an integer multiple of two.
//!
/// The image data is organized with the first row of image data appearing left
/// to right, followed immediately by the second row of image data. Each byte
/// contains the data for two columns in the current row, with the leftmost
/// column being contained in bits 7:4 and the rightmost column being contained
/// in bits 3:0.
//!
/// For example, an image six columns wide and seven scan lines tall would
/// be arranged as follows (showing how the twenty one bytes of the image would
/// appear on the display):
//!
/// \verbatim
//!
+-----+-----+-----+
//! | Byte 0 | Byte 1 | Byte 2 |
//!
+-----+-----+-----+
//! | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
//!
+-----+-----+-----+-----+
//! | Byte 3 | Byte 4 | Byte 5 |
//!
+-----+-----+-----+
//! | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
//!
+-----+-----+-----+-----+
//! | Byte 6 | Byte 7 | Byte 8 |
//!
+-----+-----+-----+
//! | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
//!
+-----+-----+-----+-----+
//! | Byte 9 | Byte 10 | Byte 11 |
//!
+-----+-----+-----+
//! | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
//!
+-----+-----+-----+-----+
//! | Byte 12 | Byte 13 | Byte 14 |

```

```

//!
//+-----+-----+-----+-----+-----+
//! | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
//! +-----+-----+-----+-----+-----+
//! |      Byte 15      |      Byte 16      |      Byte 17      |
//! +-----+-----+-----+-----+-----+
//! | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
//! +-----+-----+-----+-----+-----+
//! |      Byte 18      |      Byte 19      |      Byte 20      |
//! +-----+-----+-----+-----+-----+
//! | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
//! +-----+-----+-----+-----+-----+
//! \endverbatim
//!
//! \return None.
//*
//***** ****
void RIT128x96x4ImageDraw(const unsigned char *pucImage, unsigned long ulX,
                           unsigned long ulY, unsigned long ulWidth,
                           unsigned long ulHeight)
{
    //
    // Check the arguments.
    //
    ASSERT(ulX < 128);
    ASSERT((ulX & 1) == 0);
    ASSERT(ulY < 96);
    ASSERT((ulX + ulWidth) <= 128);
    ASSERT((ulY + ulHeight) <= 96);
    ASSERT((ulWidth & 1) == 0);

    //
    // Setup a window starting at the specified column and row, and ending
    // at the column + width and row+height.
    //
    g_pucBuffer[0] = 0x15;
    g_pucBuffer[1] = ulX / 2;
    g_pucBuffer[2] = (ulX + ulWidth - 2) / 2;
    RITWriteCommand(g_pucBuffer, 3);
    g_pucBuffer[0] = 0x75;
    g_pucBuffer[1] = ulY;
    g_pucBuffer[2] = ulY + ulHeight - 1;
    RITWriteCommand(g_pucBuffer, 3);
    RITWriteCommand(g_pucRIT128x96x4HorizontalInc,
                    sizeof(g_pucRIT128x96x4HorizontalInc));

    //
    // Loop while there are more rows to display.
    //
    while(ulHeight--)
    {
        //
        // Write this row of image data.
        //
        RITWriteData(pucImage, (ulWidth / 2));

        //
        // Advance to the next row of the image.
        //
        pucImage += (ulWidth / 2);
    }
}
//*****

```

```

//  

///! Enable the SSI component of the OLED display driver.  

//!  

///! \param ulFrequency specifies the SSI Clock Frequency to be used.  

//!  

///! This function initializes the SSI interface to the OLED display.  

//!  

///! \return None.  

//  

//*****  

void RIT128x96x4Enable(unsigned long ulFrequency)  

{  

    unsigned long ulTemp;  

  

    //  

    // Disable the SSI port.  

    //  

    SSIDisable(SSI0_BASE);  

  

    //  

    // Configure the SSI0 port for master mode.  

    //  

    SSICfgSetExpClk(SSI0_BASE, SysCtlClockGet(), SSI_FRF_MOTO_MODE_2,  

                    SSI_MODE_MASTER, ulFrequency, 8);  

  

    //  

    // (Re)Enable SSI control of the FSS pin.  

    //  

    GPIOPinTypeSSI(GPIO_PORTA_BASE, GPIO_PIN_3);  

    GPIOPadConfigSet(GPIO_PORTA_BASE, GPIO_PIN_3, GPIO_STRENGTH_8MA,  

                     GPIO_PIN_TYPE_STD_WPU);  

  

    //  

    // Enable the SSI port.  

    //  

    SSIEnable(SSI0_BASE);  

  

    //  

    // Drain the receive fifo.  

    //  

    while(SSIDataGetNonBlocking(SSI0_BASE, &ulTemp) != 0)  

    {  

    }  

  

    //  

    // Indicate that the RIT driver can use the SSI Port.  

    //  

    g_bSSIEnabled = true;  

}  

  

//*****  

//  

///! Enable the SSI component of the OLED display driver.  

//!  

///! This function initializes the SSI interface to the OLED display.  

//!  

///! \return None.  

//  

//*****  

void RIT128x96x4Disable(void)  

{  

    unsigned long ulTemp;

```

```

//  

// Indicate that the RIT driver can no longer use the SSI Port.  

//  

g_bSSIEnabled = false;  

//  

// Drain the receive fifo.  

//  

while(SSIDataGetNonBlocking(SSI0_BASE, &ulTemp) != 0)  

{  

}  

//  

// Disable the SSI port.  

//  

SSIDisable(SSI0_BASE);  

//  

// Disable SSI control of the FSS pin.  

//  

GPIOPinTypeGPIOOutput(GPIO_PORTA_BASE, GPIO_PIN_3);  

GPIOPadConfigSet(GPIO_PORTA_BASE, GPIO_PIN_3, GPIO_STRENGTH_8MA,  

                 GPIO_PIN_TYPE_STD_WPU);  

GPIOPinWrite(GPIO_PORTA_BASE, GPIO_PIN_3, GPIO_PIN_3);  

}  

*****  

//  

// Initialize the OLED display.  

//!  

//! \param ulFrequency specifies the SSI Clock Frequency to be used.  

//!  

//! This function initializes the SSI interface to the OLED display and  

//! configures the SSD1329 controller on the panel.  

//!  

//! \return None.  

//  

*****  

void  

RIT128x96x4Init(unsigned long ulFrequency)  

{  

    unsigned long ulIdx;  

//  

// Enable the SSI0 and GPIO port blocks as they are needed by this driver.  

//  

SysCtlPeripheralEnable(SYSCTL_PERIPH_SSI0);  

SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);  

SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIO_OLEDDC);  

//  

// Configure the SSI0CLK and SSIOTX pins for SSI operation.  

//  

GPIOPinTypeSSI(GPIO_PORTA_BASE, GPIO_PIN_2 | GPIO_PIN_3 | GPIO_PIN_5);  

GPIOPadConfigSet(GPIO_PORTA_BASE, GPIO_PIN_2 | GPIO_PIN_3 | GPIO_PIN_5,  

                 GPIO_STRENGTH_8MA, GPIO_PIN_TYPE_STD_WPU);  

//  

// Configure the GPIO port pin used as a D/Cn signal for OLED device,  

// and the port pin used to enable power to the OLED panel.  

//  

GPIOPinTypeGPIOOutput(GPIO_OLEDDC_BASE, GPIO_OLEDDC_PIN | GPIO_OLEDEN_PIN);  

GPIOPadConfigSet(GPIO_OLEDDC_BASE, GPIO_OLEDDC_PIN | GPIO_OLEDEN_PIN,

```

```

        GPIO_STRENGTH_8MA, GPIO_PIN_TYPE_STD);
GPIOPinWrite(GPIO_OLEDDC_BASE, GPIO_OLEDDC_PIN | GPIO_OLEDEN_PIN,
              GPIO_OLEDDC_PIN | GPIO_OLEDEN_PIN);

// Configure and enable the SSIO port for master mode.
// RIT128x96x4Enable(ulFrequency);

// Clear the frame buffer.
// RIT128x96x4Clear();

// Initialize the SSD1329 controller. Loop through the initialization
// sequence array, sending each command "string" to the controller.
//
for(ulIdx = 0; ulIdx < sizeof(g_pucRIT128x96x4Init);
    ulIdx += g_pucRIT128x96x4Init[ulIdx] + 1)
{
    //
    // Send this command.
    //
    RITWriteCommand(g_pucRIT128x96x4Init + ulIdx + 1,
                    g_pucRIT128x96x4Init[ulIdx] - 1);
}

//*****
//!
//! Turns on the OLED display.
//!
//! This function will turn on the OLED display, causing it to display the
//! contents of its internal frame buffer.
//!
//! \return None.
//!
//*****
void
RIT128x96x4DisplayOn(void)
{
    unsigned long ulIdx;

    //
    // Initialize the SSD1329 controller. Loop through the initialization
    // sequence array, sending each command "string" to the controller.
    //
    for(ulIdx = 0; ulIdx < sizeof(g_pucRIT128x96x4Init);
        ulIdx += g_pucRIT128x96x4Init[ulIdx] + 1)
    {
        //
        // Send this command.
        //
        RITWriteCommand(g_pucRIT128x96x4Init + ulIdx + 1,
                        g_pucRIT128x96x4Init[ulIdx] - 1);
    }

//*****
//!
//! Turns off the OLED display.
//!
//! This function will turn off the OLED display. This will stop the scanning

```

```
///! of the panel and turn off the on-chip DC-DC converter, preventing damage to
///! the panel due to burn-in (it has similar characters to a CRT in this
///! respect).
//!
///! \return None.
//!
//*****void
RIT128x96x4DisplayOff(void)
{
    static const unsigned char pucCommand1[] =
    {
        0xAE, 0xe3
    };

    //
    // Put the display to sleep.
    //
    RITWriteCommand(pucCommand1, sizeof(pucCommand1));
}

//*****
//!
// Close the Doxygen group.
//! @}
//!
//*****
```