Implementing an LCD Controller

May 2008, ver. 1.0 Application Note 527

Introduction

Graphical LCD modules are increasingly prevalent in embedded systems, where they are used to control, configure, and interact with applications. Inexpensive LCD modules available today provide high resolution and color display capabilities, and many also integrate a touch panel interface. Implementing an LCD controller in an FPGA provides the flexibility to incorporate additional LCD module features quickly and easily as they become available.

The Altera® Nios® Embedded Evaluation Kit (NEEK) includes an LCD module made by TPO, formerly Toppoly Optoelectronics Corp. The 4.3" Toppoly TD043MTEA1 LCD module incorporates an 800×480-pixel, active-matrix color display and a touch-screen interface. The NEEK includes pregenerated examples that demonstrate the capabilities of the LCD display. The kit also provides the necessary hardware designs, source code, IP peripherals, and driver source code for these examples, from which you can develop your own graphical applications.

This application note teaches you to create your own LCD module for your embedded design, by providing the following information:

- Background information on the LCD controller subsystem as implemented in the NEEK. The application note describes the peripherals and device drivers required to control the LCD module, and their configuration.
- Porting guidelines to help you implement the LCD controller subsystem for your own LCD module.

Prerequisites

This document assumes you are familiar with the following Altera software design tools and intellectual property (IP):

- Quartus® II software
- SOPC Builder
- Avalon® Memory-Mapped (Avalon-MM) and Avalon Streaming (Avalon-ST) interfaces
- Nios II processor

NEEK LCD Controller Hardware Components

Figure 1 provides a high-level, hierarchical view of the peripherals and interfaces that implement the NEEK LCD controller design. The main components of the LCD controller are the LCD module, the MAX® II device, and the video pipeline in the FPGA.

Figure 1. LCD Controller Subsystem

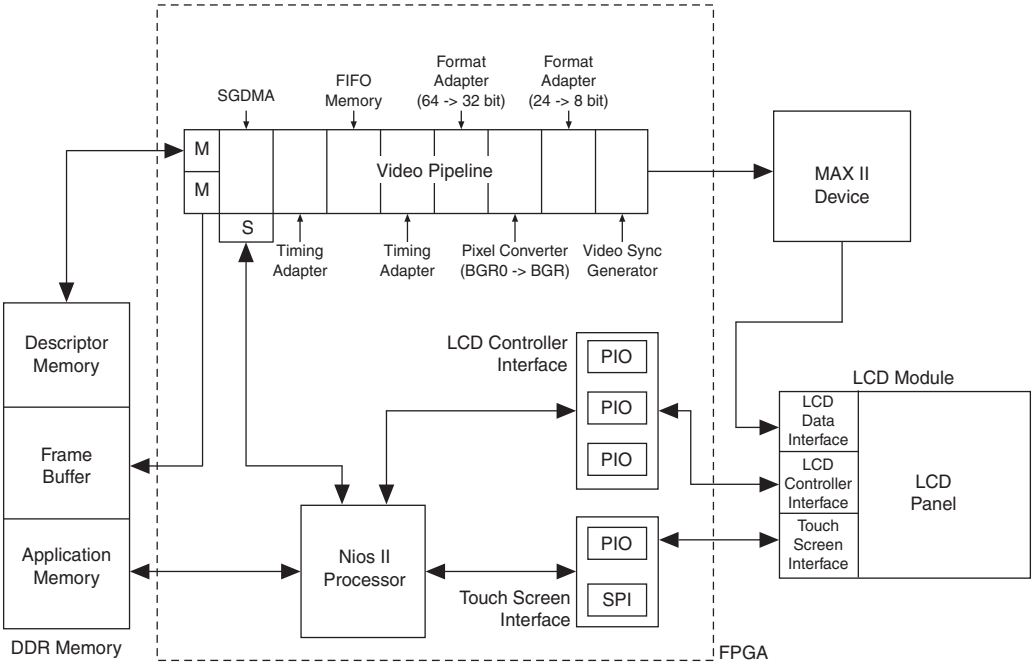

Figure 1 focuses on the connection of the peripherals for the LCD module, including the video pipeline. For a system-level view of the NEEK, refer to the *Nios II Standard Hardware for the Embedded Evaluation Board* chapter of the *Nios II Embedded Evaluation Kit, Cyclone® III Edition User Guide*.

The following sections describe the main components of the LCD controller:

- "Touch Screen LCD Module"
- "MAX II Device" on page 4
- "Video Pipeline" on page 5

Touch Screen LCD Module

The touch screen LCD module is a single hardware device with the following three distinct subsystems:

- LCD graphical data interface
- Touch screen interface
- LCD controller interface

These interfaces use different buses and protocols.

LCD Data Interface

The LCD graphical data interface carries video data to the LCD module. This interface includes a 24-bit Red-Green-Blue (RGB) data bus and some control signals.

To display a frame of video data, the control and data line behavior and sequencing must conform to the Toppoly LCD module specification. Conceptually, the video sync generator peripheral—the final stage in the video pipeline—implements the display operation. However, because of pin restrictions on the FPGA, the MAX II device accepts a time-division multiplexed (TDM) stream of pixel data from the FPGA device, performs demultiplexing to convert it to a 24-bit stream, and sends the 24-bit stream to the LCD data interface.

The TDM stream in the NEEK implementation is in Blue-Green-Red (BGR) format: the blue color component is transmitted first, followed by the green color component, and then the red color component.

For more information about the Toppoly LCD module specification, refer to the user guide at www.terasic.com under TRDB_LTM.

Touch Screen Interface

A Serial Peripheral Interface (SPI) and a Parallel I/O (PIO) peripheral implement the touch screen interface. The SPI peripheral communicates with the Analog Devices AD7843 touch screen digitizer chip to signal `pen_move` events. A single PIO line captures pen interrupt events—transitions on the `pen_down` line from the AD7843 chip—to indicate `pen_down` and `pen_up` events. The Nios II processor in the system runs software that drives the SPI and PIO peripherals.

Table 1 lists the touch screen interface peripherals in the NEEK hardware example design.

Table 1. Touch Screen Interface Peripherals

Name	SOPC Peripheral Type	Role
touch_panel_spi	SPI (3-Wire Serial)	implements SPI interface
touch_panel_pen_irq_n	Parallel I/O	implements pen interrupt interface

LCD Controller Interface

The LCD display module contains a controller chip that configures the module for operation. The controller chip communicates through a simple, proprietary three-wire interface. The simple communication protocol for sending and receiving data is implemented with three lines from a general purpose PIO peripheral on the FPGA.

The PIO peripheral is controlled by a general purpose Hardware Abstraction Layer (HAL) software driver in the system. Therefore, most of the communication over the three-wire interface is driven by the Nios II processor toggling individual PIO peripheral ports.

Table 2 lists the LCD controller interface peripherals in the NEEK hardware example design.

Table 2. LCD Controller Interface Peripherals

Name	SOPC Peripheral Type	Role
lcd_i2c_scl	Parallel I/O	implements clock signal
lcd_i2c_en	Parallel I/O	implements device enable signal
lcd_i2c_dat	Parallel I/O	implements data signal

MAX II Device

The MAX II device provides voltage translation and color demultiplexing between the FPGA and the LCD module. The MAX II device also serves as a voltage translator between the Cyclone III device 2.5V inputs and the 3.3 V outputs of many of the external peripherals to which it connects.

To conserve pins on the FPGA, the video pipeline generates an 8-bit TDM stream, in which each clocked value corresponds to one color component of the pixel's value (red or green or blue). The MAX II device accepts this 8-bit stream and converts it back to the 24-bit, parallel RGB format that the LCD module expects.

For more information about the MAX II design, and how it performs time-domain based multiplexing and demultiplexing, refer to the MAX II design file that is included with the NEEK evaluation kit. The path to the MAX II design file is <*Installation Path*>**/board_design_files/assembly/lcd_multimedia_daughtercard/maxII**.

Video Pipeline

The video pipeline is the core of the LCD example design. This component is responsible for driving video data signals on the LCD module data bus and for reading frame buffer data generated by the Nios II processor. The video pipeline is a series of specialized Avalon-ST peripherals that move, process, and operate on pixel data.

For more information about the Avalon-ST interface specification, refer to the *Avalon Streaming Interfaces* chapter in *Avalon Interface Specifications*.

The following sections describe the peripherals in the video pipeline:

- "Video Sync Generator Peripheral"—the output stage of the pipeline
- "Avalon-ST Data Format Adapter Peripheral (24 Bits to 8 Bits)"
- "Avalon-ST Pixel Converter Peripheral (BGR0 --> BGR)"
- "Avalon-ST Data Format Adapter Peripheral (64 Bits to 32 Bits)"
- "Avalon-ST Timing Adapter Peripheral (FIFO-to-Data Format Adapter)"
- "FIFO Memory Peripheral"
- "Avalon-ST Timing Adapter Peripheral (SGDMA to FIFO Memory)"
- "Scatter-Gather DMA (SGDMA) Controller Peripheral"—the initial stage of the pipeline

Video Sync Generator Peripheral

The video sync generator peripheral transmits pixel data to the LCD module. The generator sequences the control and data signals for the LCD module's data bus.

The video sync generator accepts a stream of pixel data at its input, encoded with particular **Data Stream Width** and **Beats per Pixel** values. The generator outputs the video data in the same format in which it was received, but includes additional sequencing information to drive the display.

SOPC Builder requires that you determine values for the following video sync generator parameters:

- **Data Stream Width**
- **Beats per Pixel**
- **Number of Columns**
- **Number of Rows**
- **Horizontal Blank Lines**
- **Horizontal Front Porch Pixels**
- **Vertical Blank Lines**
- **Vertical Front Porch Lines**
- **Total Horizontal Scan Pixels**
- **Total Vertical Scan Pixels**

The **Data Stream Width** and **Beats per Pixel** parameters control the interfaces to the previous peripheral (stage) in the video pipeline. The other parameters control correct sequencing of data and control for the LCD data bus; their correct values are determined from the Toppoly LCD module datasheet or user guide.

In the NEEK design, **Data Stream Width** has value 8 and **Beats per Pixel** has value 3. These parameter values cause the pixel data to enter the MAX II device 8 bits at a time, in sequences of three 8-bit vectors. Each sequence represents the color of a single pixel. The MAX II device performs demultiplexing to convert these sequences for the display into 24-bit vectors that each contain complete color information for a pixel. The previous pipeline stage, which feeds the video sync generator component, must output video pixel data in the 8-bit vector format.

For more information about the video sync generator peripheral, refer to "LCD Panel Interface" on page 21.

For more information about the Toppoly LCD module specification, refer to the user guide at www.terasic.com under TRDB_LTM.

Table 3 describes the video sync generator peripheral in the NEEK hardware example design.

Table 3. Video Sync Generator Peripheral

Name	SOPC Peripheral Type	Role
lcd_sync_generator	Video Sync Generator	drives LCD module data bus

For more information about the video sync generator peripheral, refer to the *Video Sync Generator and Pixel Converter Cores* chapter in volume 5 of the *Quartus II Handbook*.

Avalon-ST Data Format Adapter Peripheral (24 Bits to 8 Bits)

The Avalon-ST data format adapter peripheral enables you to convert data units between buses with different widths. The widths are parameter values you set in SOPC Builder.

In the NEEK design, this peripheral's role in the video pipeline is to convert a 24-bit pixel stream—an RGB pixel value—to an 8-bit pixel stream in which each RGB color component is transmitted separately.

This Avalon-ST data format adapter peripheral accepts one 24-bit pixel value per clock cycle at its input (3 data symbols, each 8 bits wide), and generates three 8-bit, clocked values at its output (one data symbol at a time, each 8 bits wide). This conversion matches the downstream video sync generator's requirement for an 8-bit data stream, and the upstream pixel converter's 24-bit output.

Table 4 describes this Avalon-ST data format adapter peripheral in the NEEK hardware example design.

Table 4. Avalon-ST Data Format Adapter Peripheral (24 to 8 Bits)

Name	SOPC Peripheral Type	Role
lcd_32_to_8_bits_dfa	Avalon-ST Data Format Adapter	converts data values from 24 to 8 bits

For more information about the Avalon-ST data format adapter, refer to the *System Interconnect Fabric for Streaming Interfaces* chapter in volume 4 of the *Quartus II Handbook*.

Avalon-ST Pixel Converter Peripheral (BGR0 --> BGR)

The Avalon-ST pixel converter peripheral converts 32-bit pixel values, in a 32-bit format in which the final byte is unused (BGR0), to the 24-bit format the LCD module expects.

You set the **Source symbols per beat** parameter of the Avalon-ST pixel converter peripheral in SOPC Builder, but you control the function of this peripheral by modifying its Verilog HDL source file. All HDL files provided on the NEEK installation CD are in Verilog HDL. Many of these files can be recreated in VHDL using SOPC Builder. However, the pixel converter peripheral is a custom peripheral for which a VHDL implementation is not provided.

Table 5 describes the Avalon-ST pixel converter peripheral in the NEEK hardware example design.

Table 5. Avalon-ST Pixel Converter Peripheral

Name	SOPC Peripheral Type	Role
`lcd_pixel_converter`	Pixel Converter (BGR0 --> BGR)	removes unused final byte of incoming 32-bit data

For more information about modifying the Avalon-ST pixel converter peripheral, refer to"Video Pipeline Peripherals" on page 27.

For more information about the pixel converter peripheral, refer to the *Video Sync Generator and Pixel Converter Cores* chapter in volume 5 of the *Quartus II Handbook*.

Avalon-ST Data Format Adapter Peripheral (64 Bits to 32 Bits)

This Avalon-ST data format adapter converts 64-bit data values (8 data symbols, each 8 bits wide) to 32-bit data values (4 data symbols, each 8 bits wide). Each 64-bit data value contains two RGB pixel values. Each pixel value is encoded in 32 bits.

The downstream Avalon-ST pixel converter peripheral requires a 32-bit data value for its input, but the upstream data value coming from the timing adapter peripheral is 64 bits.

Table 6 describes this Avalon-ST data format adapter peripheral in the NEEK hardware example design.

Table 6. Avalon-ST Data Format Adapter Peripheral (64 to 32 Bits)

Name	SOPC Peripheral Type	Role
lcd_64_to_32_bits	Avalon-ST Data Format Adapter	converts data values from 64 to 32 bits

Avalon-ST Timing Adapter Peripheral (FIFO-to-Data Format Adapter)

The Avalon-ST timing adapter peripheral is a bridge between two Avalon-ST peripherals that have different latency requirements. You set the values of the timing adapter peripheral parameters in SOPC Builder.

This timing adapter peripheral reconciles the data latency mismatch between the downstream format adapter peripheral, which has a latency of 0, and the upstream FIFO memory peripheral, which has a latency of 1.

Table 7 describes the FIFO-to-data format adapter Avalon-ST timing adapter peripheral in the NEEK hardware example design.

Table 7. Avalon-ST Timing Adapter Peripheral (FIFO to Data Format Adapter)

Name	SOPC Peripheral Type	Role
lcd_ta_fifo_to_dfa	Avalon-ST Timing Adapter	bridges latency timing between the FIFO and the data format adapter peripheral

For more information about the Avalon-ST timing adapter, refer to the *Avalon Streaming Interconnect Components* chapter in volume 4 of the *Quartus II Handbook.*

FIFO Memory Peripheral

The FIFO memory pipeline stage is implemented by an on-chip FIFO memory peripheral. The FIFO memory provides temporary data buffering in the video pipeline. Pixel data loaded in the video pipeline from frame buffer memory by the SGDMA peripheral may not arrive in a timely manner. Delays can occur due to latency or bus contention issues experienced by the SGDMA peripheral while accessing the frame buffer memory.

In the NEEK design, the FIFO memory peripheral is configured to accept as many as 128, 64-bit data values (eight 8-bit symbols, each 8 bits wide) from the upstream timing adapter peripheral. The FIFO memory peripheral passes any valid data units it has to the downstream timing adapter peripheral, one 64-bit data value per clock. The FIFO memory peripheral also includes Avalon-ST backpressure support to prevent the upstream timing adapter peripheral from loading new data in the FIFO memory when the FIFO memory is full.

This FIFO memory smooths out the video pipeline data flow successfully because, on average, data arrives from the SGDMA more quickly than it can be processed by the rest of the pipeline. Buffering is necessary because the SGDMA cannot guarantee precisely when this data arrives.

Table 8 describes the FIFO memory peripheral in the NEEK hardware example design.

Table 8. FIFO Memory Peripheral

Name	SOPC Peripheral Type	Role
lcd_pixel_fifo	On-Chip FIFO Memory	pixel data buffer

For more information about configuring and using the FIFO memory peripheral, refer to "Video Pipeline Peripherals" on page 27.

For more information about the Avalon-ST on-chip FIFO memory peripheral, refer to the *On-Chip FIFO Memory Core* chapter in volume 5 of the *Quartus II Handbook*.

Avalon-ST Timing Adapter Peripheral (SGDMA to FIFO Memory)

This timing adapter peripheral reconciles the data latency mismatch between the downstream FIFO memory peripheral, which has a latency of 1, and the upstream SGDMA peripheral, which has a latency of 0.

Table 9 describes the SGDMA-to-FIFO memory Avalon-ST timing adapter peripheral in the NEEK hardware example design.

Table 9. Avalon-ST Timing Adapter Peripheral (SGDMA to FIFO Memory)

Name	SOPC Peripheral Type	Role
lcd_ta_sgdma_to_fifo	Avalon-ST Timing Adapter	bridges latency timing between the SGDMA and the FIFO memory peripheral

Scatter-Gather DMA (SGDMA) Controller Peripheral

After the descriptor table of the SGDMA controller is programmed, the SGDMA peripheral moves pixel data from the frame buffer memory to the video pipeline autonomously, without intervention from the Nios II processor.

In the NEEK design, the SGDMA peripheral is configured to read pixel data from the frame buffer memory—located in the DDR SDRAM component—and to pass the read data to the rest of the video pipeline for processing. For optimal performance, the SGDMA reads units of pixel data 64 bits at a time, corresponding to the interface width of the DDR SDRAM memory.

The SGDMA performs memory transfer operations by descriptor. Each descriptor can specify up to 64,000 bytes per transfer. To display an entire video frame buffer, the Nios II processor creates a list of these descriptors for the SGDMA to process. The SGDMA processes the entire descriptor chain, continuously and repeatedly, to drive the video pipeline.

Conceptually, the Nios II processor manipulates the frame buffer memory to change the pixel values. The Nios II processor can either operate on the frame buffer being utilized by the SGDMA, or on a new frame buffer which is then be passed to the SGDMA for processing. For efficiency, the Nios II processor manipulates RGB pixel values using 32-bit read and write operations. The pixel values are stored in the frame buffer memory in a 64-bit format that represents the values for two pixels. A software application programming interface (API) controls the process for manipulating and changing the frame buffer memory.

Table 10 describes the SGDMA peripheral in the NEEK hardware example design.

Table 10. SGDMA Peripheral

Name	SOPC Peripheral Type	Role
`lcd_sgdma`	Scatter-Gather DMA Controller	reads pixel data from frame buffer memory

For more information about configuring the SGDMA peripheral for operation, refer to "Video Buffer Memory and SGDMA" on page 25.

For more information about the SGDMA peripheral, refer to the *Scatter-Gather DMA Controller Core* chapter in volume 5 of the *Quartus II Handbook*.

Software Driver and Graphics Routines

This section discusses the Altera-supplied API for the LCD controller subsystem components.

The component APIs available to developers often use additional function calls to accomplish a particular task. For more information about how an API function accomplishes a particular task, you must examine the API function source file. For a list of source files and their locations in the software example design directories, refer to "Appendix A. NEEK Design Examples and Sources" on page 32.

Touch Screen Software API

The touch screen software API provides the application with an abstract pen interface, consisting of X and Y coordinates, and pen state (*up* or *down*).

The touch-screen source code is located in the following two files:

- **alt_touchscreen.h**
- **alt_touchscreen.c**

API operations are available to perform the following four distinct actions on the touch screen:

- Initialize
- Calibrate
- Operate
- Stop

Initializing

The function `alt_touchscreen_init()` configures the touch-screen interface with the following information:

- `screen`: Instance variable corresponding to your screen
- `spi_controller_base`: Base address of SPI peripheral used to communicate with the LCD touch screen interface
- `spi_controller_irq_number`: IRQ number of SPI peripheral used to communicate with the LCD touch screen interface
- `pen_detect_pio_base`: Base address of PIO peripheral connected to pen detect line of the LCD touch screen
- `samples_per_second`: Sampling rate in Hz
- `swap_xy`: A Boolean value that controls whether or not to swap the X and Y coordinate axes

Your application must call this function before calling any other functions in the touch-screen API.

Calibrating the Screen Coordinates

The `alt_touchscreen_calibrate_upper_right()` and `alt_touchscreen_calibrate_lower_left()` functions calibrate the touch screen. The `alt_touchscreen_calibrate_upper_right()` function calibrates the upper right corner of the LCD display, and the `alt_touchscreen_calibrate_lower_left()` function calibrates its lower left corner.

These functions accept two X and Y coordinate value pairs, one corresponding to the touch-screen analog-to-digital (ADC) value and the other corresponding to the respective screen pixel location. The ADC value for the pixel location is determined empirically, while the pixel screen value is determined from the geometry of the LCD screen.

To ensure that the touch-screen drivers behave properly, Altera recommends that you call both of these functions before operating the touch screen.

The NEEK applications are designed for a specific screen and use pre-computed value pairs for these functions. No calibration API functions are run. However, for accuracy, Altera recommends that you call both of the calibration functions before operating the touch screen.

Operating

The `alt_touchscreen_get_pen()` and `alt_touchscreen_event_loop_update()` functions provide a simple operating interface for the touch screen. The `alt_touchscreen_get_pen()` function reports the touch screen status, and the `alt_touchscreen_event_loop_update()` function updates the internal state of the touch screen software.

Querying

The `alt_touchscreen_get_pen()` function returns the following parameters:

- `pen_down`: A Boolean value that indicates whether the pen is *up* or *down*
- `x`, `y`: The X and Y coordinates of the pen location

The X and Y coordinate values correspond to the most recent recorded pen location, which is usually meaningful only when the pen status is *down*. This function reports touch-screen status information asynchronously with hardware operations. Actual reads of the

touch-screen hardware occur at lower levels of the API in an interrupt-driven process; the timer interrupt service routine (ISR) queries the touch-screen hardware at the sampling rate (specified with the `alt_touchscreen_init()` function), and data is read back and reported by the ISR for the SPI described in "Touch Screen Interface" on page 3.

Callback Methodology

The touch screen API supports a mixed callback methodology in which registered callback events are detected in an ISR-driven process, but the callback function is run from the application.

To register a callback event, run the `alt_touchscreen_register_callback_func()` function, specifying the following parameters:

- `screen`: Instance variable corresponding to your screen
- `callback_reason`: Event that triggers the callback function. Supported events include `pen_up`, `pen_down`, and `pen_move`
- `callback_func`: The function to handle the specified event, specified using the `alt_touchscreen_event_callback()` function
- `context`: Pointer to a user-defined data structure or value that represents the callback context

To disable a callback event, specify a `null` value for the callback function.

The application must call the `alt_touchscreen_event_loop_update()` function periodically. This function checks all registered callback events to see if they occurred, and if so, calls the associated callback function. The `alt_touchscreen_event_loop_update()` function can be called from an operating system thread or periodically in an event loop.

The callback function accepts the following parameters:

- `pen_down`: indicates whether pen is down (`pen_down == 1`)
- `x`: value corresponding to the pen's location on the X axis
- `y`: value corresponding to the pen's location on the Y axis
- `context`: pointer to a user-defined data structure or value

Stopping

The `alt_touchscreen_stop()` function stops the touch screen and puts it in a safe state. When called, this function disables interrupts for the timer and SPI peripherals, stops the video hardware, and clears any registered callback events.

LCD Module Software API

The LCD software API provides a high-level initialization function and a set of low-level functions for communicating with the LCD module registers. During normal operation, you should only need to communicate with the LCD module during system configuration. This communication occurs through the software API.

The LCD module source code is located in the following two files:

- **alt_tpo_lcd.h**
- **alt_tpo_lcd.c**

Initializing

The initialization function enables you to quickly configure the LCD module for operation. This function configures the LCD module with a precomputed set of default parameters for the gamma curve and the positive polarity voltage.

The initialization function has the following prototype:

```
int alt_tpo_lcd_init (alt_tpo_lcd *lcd, alt_u32 width, alt_u32 height);
```

The function has the following parameters:

- `lcd`: Instance variable corresponding to your screen
- `width`: Width you would like displayed
- `height`: Height you would like displayed

Before calling this function you must specify the pins on the PIO peripheral connected to the LCD controller interface's three-wire interface, described in "LCD Controller Interface" on page 4.

The PIO pins are directly assigned in the C structure that holds the information about the current state of the display (the `lcd` struct). Example 1 illustrates an assignment for the three pins:

Example 1. A Sample Assignment for the PIO Pins

```
lcd.scen_pio = LCD_I2C_EN_BASE;
lcd.scl_pio = LCD_I2C_SCL_BASE;
lcd.sda_pio = LCD_I2C_SDAT_BASE;
```

Accessing the Configuration Registers

The LCD module's registers are physically accessed through the three-wire interface described in "LCD Controller Interface" on page 4. This interface connects the LCD module to the FPGA. You access the LCD configuration registers with the following functions:

- `alt_u8 alt_tpo_lcd_read_config_register( alt_tpo_lcd *lcd, alt_u8 addr)`
- `void alt_tpo_lcd_write_config_register( alt_tpo_lcd *lcd, alt_u8 addr, alt_u8 data)`

The `alt_tpo_lcd_read_config_register()` function reads a register and returns the data it reads. This function accepts the following parameters:

- `lcd`: Instance variable corresponding to your screen
- `addr`: LCD module configuration register to access

The `alt_tpo_lcd_write_config_register()` function writes a register. It has all the parameters that the read function has, and the following additional input parameter:

- `data`: The data value to be written.

Before calling either of these functions, you must assign the corresponding pins to the C `lcd` structure, as shown in Example 1.

For more information about the LCD configuration registers, refer to the user guide at www.terasic.com under TRDB_LTM.

Video Pipeline Subsystem API

The video pipeline software API provides a self-contained method to control the video pipeline and manage the graphical frame buffers.

The video pipeline source code is located in the following two files:

- **alt_video_display.h**
- **alt_video_display.c**

API operations are available to perform the following three distinct actions on the video pipeline:

- Initialize
- Stop
- Manage and manipulate frame buffers

Initializing

The initialization function for the video pipeline is a high-level wrapper function that provides a managed interface for system frame buffers. The initialization function has the following prototype:

```
alt_video_display* alt_video_display_init(
       char* sgdma_name,
       int width,
       int height,
       int color_depth,
       int buffer_location,
       int descriptor_location,
       int num_buffers);
```

The `alt_video_display_init()` function initializes the video pipeline for operation. This function returns a pointer to an `alt_video_display` structure if successful, or `null` if it fails. The C source `alt_video_display` struct keeps track of all of the frame buffer information, including the start addresses for all of the frame buffers. This function accepts the following parameters:

- `sgdma_name`: SGDMA instance connected up to the video pipeline
- `width`: Width of display
- `height`: Height of display
- `color_depth`: Bits required to represent a pixel
- `buffer_location`: Location of frame buffer
- `descriptor_location`: Location of SGDMA descriptor memory
- `num_buffers`: Total number of frame buffers to use

After this function completes successfully, all frame buffers in the system are allocated and the video pipeline hardware is running. All allocated frame buffers are filled with the pixel data corresponding to the color black. You can overwrite these default parameters by modifying the macros defined in the header file **alt_video_display.h**.

Buffer and SGDMA Descriptor Locations

The `buffer_location` and `descriptor_location` parameters can specify either of the following two modes of operation:

- Absolute address operation
- Heap-based operation

In absolute address operation, the integer value you provide for both parameters corresponds to the first address in a range of addresses. The `buffer_location` parameter value is the initial address of the video display buffer, and the `descriptor_location` parameter value is the initial address of the descriptor memory.

In heap-based operation you do not specify an absolute address, but instead rely on the C runtime library's heap-memory manager to provide you with the memory required. The advantage of the heap-based approach is that all frame buffer video memory is managed for you. The limitation of this approach is the potential access contention when both the SGDMA—the initial stage in the video pipeline—and the Nios II processor require access to the heap memory, which could result in some performance degradation for both peripherals.

To implement heap-based operation for either of the two parameters—the `buffer_location` or the `descriptor_location` parameter—you must pass the macro value `ALT_VIDEO_DISPLAY_USE_HEAP` to the `alt_video_display_init()` function as the value for that parameter.

Stopping

Video pipeline operation is stopped through the use of the `alt_video_display_close()` function, which has the following prototype:

```
void alt_video_display_close( alt_video_display* display,
                      int buffer_location,
                      int descriptor_location );
```

The function accepts the following parameters:

- `display`: Pointer to the `alt_video_display` structure
- `buffer_location`: Pointer to the frame buffer
- `descriptor_location`: Pointer to the SGDMA descriptors

Call the `alt_video_display_close()` function only after you have completed using the video pipeline subsystem and do not intend to display any more graphical information on the LCD display.

Managing and Manipulating Frame Buffers

The video pipeline frame buffers are managed with the functions `alt_video_display_buffer_is_available()` and `alt_video_display_register_written_buffer()`. In addition, the `alt_video_display_clear_screen()` function is provided to clear the screen with a single function call.

Managing Frame Buffers
The frame buffers are managed through the `alt_video_display_buffer_is_available()` and `alt_video_display_register_written_buffer()` functions. The former function acquires a free frame buffer to which to write, and the latter function displays the frame buffer.

The `alt_video_display_buffer_is_available()` function has the following prototype:

```
int alt_video_display_buffer_is_available(alt_video_display* display);
```

This function returns the next free frame buffer available for display. The function returns a condition code of `0` if an empty buffer is found; any other return value indicates failure. The function accepts the following parameter:

- `display`: This variable is a pointer to a data structure that keeps track of all of the frame buffer information, including the start addresses for all of the frame buffers.

The `alt_video_display_register_written_buffer()` function has the following prototype:

```
int alt_video_display_register_written_buffer(alt_video_display* display);
```

You call this function to register your written frame buffer for display. This function returns a condition code of 0 if the buffer is registered for display successfully. Any other return value indicates that the buffer registered for display is already being displayed by the video pipeline component. This function accepts the following parameter:

- `display`: This variable is a pointer to a data structure that keeps track of all of the frame buffer information, including the start addresses for all of the frame buffers. The function uses this information to determine the actual frame buffer to display.

An important behavioral aspect of video-pipeline software is the persistence of the displayed buffer contents. A buffer registered for display with the

alt_video_display_register_written_buffer() function continues to be displayed on the LCD screen until this function is called again with a new buffer. Calling the alt_video_display_register_written_buffer() function repeatedly causes the registered buffers to display in FIFO ordering, and the contents of the final buffer to remain on display.

Manipulating Frame Buffer Contents

The video pipeline API is intentionally limited to the single helper function alt_video_display_clear_screen(), which clears the screen. Your application is responsible for filling the frame buffer memory with the pixel data you would like to display to the screen.

The alt_video_display_clear_screen() function has the following prototype:

```
inline void alt_video_display_clear_screen(
                    alt_video_display* frame_buffer, char color);
```

The function accepts the following parameters:

- frame_buffer: Pointer to the frame buffer to be cleared
- color: The 8-bit value specifying the color representation for each of red, green, and blue in the pixel color with which to fill the display

To display your own screen contents, you must manipulate the pixel data in the frame buffer manually. In the case of the Altera-supplied reference design, the pixels are encoded in BGR0 format. The pixels are arranged in the frame buffer such that the first buffer location contains the upper-leftmost pixel for the display. The LCD module rasterizes the pixels on the LCD screen from left to right, line by line.

Graphics Libraries and Software

In your design, you may wish to use a graphics library or display routines to simplify your system design. The NEEK includes several graphics libraries that you can use. Altera provides a simple graphics library that includes support for rendering shapes and displaying text on the screen. Demonstrations of several third-party graphics libraries are also provided.

For a list of all of the Altera-supplied graphics libraries and examples for which source code is available, refer to "Appendix A. NEEK Design Examples and Sources" on page 32.

Porting Guidelines

This section discusses porting the LCD design to your own hardware. To plan how to port the LCD design, first consider the following aspects of your design:

1. LCD touch-screen interface: Determine the peripherals required to support the LCD module's touch screen—if present—and to communicate with the controller hardware. Determine the configuration of the video sync generator peripheral that is required to support the LCD module's graphical data interface.

2. External device: Determine whether you need additional external devices to support the video interface (such as an Altera MAX II CPLD device).

3. Video frame buffer and SGMDA peripheral: Determine how many frame buffers you wish your system to have, the required size of these frame buffers, and their location in memory. You must configure the SGDMA peripheral to support the memory type you choose.

4. Video pipeline peripherals: Determine the peripherals you require in the video pipeline to support the transport, sequencing, and conversion of the pixel data from the video frame buffer(s) to the video sync generator peripheral.

This section discusses these steps in detail.

In addition, you must plan to port your software routines, implement LCD panel drivers, and use the video pipeline subsystem API to implement your application.

This section of the document contains the following sections:

LCD Panel Interface

Most LCD panels have at least two different classes of interfaces, a data-plane interface for receiving graphical data, and a control-plane interface for communicating with control registers or the touch panel. You must consider these two classes of interfaces separately when

connecting them to the system. The Toppoly LCD module has three interfaces, an LCD data interface (a data-plane interface), and the touch screen and LCD controller interfaces (control-plane interfaces).

Touch Screen and LCD Controller Interfaces

In the case of the LCD display for the NEEK board, both the LCD panel's control register and touch screen controller interfaces use serial communication channels. The LCD panel's control register uses a proprietary three-wire interface, and the touch screen controller interface uses an SPI interface (provided through the AD7843 digitizer chip).

Serial interfaces are very popular for low-speed, control-plane communication between integrated circuits. Your LCD module probably has a serial configuration interface. SOPC Builder supports different classes of serial interfaces easily, as shown in Table 11.

Table 11. Supporting Different Classes of Serial Interfaces

Type	Description	Solution(s)
SPI	Serial Peripheral Interface	Altera Avalon SPI core
I^2C	Inter-Integrated Circuit	Altera Avalon PIO core and software implementation
		Third-party IP
Proprietary	Any non-standard serial interface	Altera Avalon PIO core and software implementation
		Custom or third-party IP

As shown in Table 11, SOPC Builder supports several different standard and non-standard serial interfaces. When performance is not a concern, almost any serial bus interface can be created using the Altera PIO peripheral driven by Nios II processor software. The LCD controller interface is implemented this way.

For more information about available serial IP core offerings, refer to the Altera IP Megastore.

LCD Data Interface

The LCD module's data interface is composed of the following two parts:

- Physical interface—data pins that drive the display
- Logical interface—the encoding of the pixel data—how red, green, and blue color components are represented

To implement your LCD controller straightforwardly in SOPC Builder, you must ensure that your LCD module's physical and logical interfaces are compatible with that of the video sync generator component. Physical interface compatibility requires that both the voltages and the pin mappings of the two components are compatible. Altera recommends that you design for compatibility between the LCD module's data interface and the FPGA in the following order:

1. "Voltage Compatibility"—part of the physical interface
2. "Physical Interface"—pin mapping
3. "Logical Interface"

Voltage Compatibility

Ensure that the LCD module's data voltage signaling requirements match those of the FPGA. If an incompatibility exists, you must bridge the difference with a voltage translation device such as a buffer IC or Max II device.

The NEEK development kit uses the MAX3378E IC from Maxim Integrated Products (www.maxim-ic.com) for direct voltage translation, and the MAX II CPLD for indirect voltage translation. Voltage translators are required because the Cyclone III device uses 2.5-V input and output pins while many of the external peripherals to which it connects use 3.3-V pins.

Physical Interface

Ensure that the LCD module's data physical interface is compatible with the video sync generator peripheral's interface presented on the FPGA. The video sync generator peripheral exposes a very specific set of pins to support one class of LCD display. The only configurable aspect of the video sync generator's physical interface is the multiplexing of the red, green, and blue color channels to a single set of data lines.

If the video sync generator cannot interface to your LCD display's data interface directly, you must use additional logic, either inside the FPGA or externally through another device, to correct this problem.

In the NEEK board design, the video sync generator's physical interface does not match that of the LCD module. The video sync generator is configured to multiplex color channels on a single 8-bit data bus, to conserve FPGA pin resources. The MAX II device accepts the video sync generator's 8-bit data and control signals and performs demultiplexing to match the LCD module's data interface.

For more information about the pins available on the video sync generator peripheral, refer to the *Video Sync Generator and Pixel Converter Cores* chapter in volume 5 of the *Quartus II Handbook*.

Logical Interface

The logical interface, in the context of the LCD controller's data interface, is how the physical interface signals of the video sync generator component are sequenced to transmit a video frame. In the NEEK board design, the video sync generator peripheral control signals (`HSYNC`, `VSYNC`, and others) output the pixel data (`RED`, `GREEN`, and `BLUE` bus signals) in some defined sequence to transmit a video frame.

The video sync generator component provides parameterization options to control the height, width, and control signals timings for the LCD module.

The logical interface between the video sync generator and the LCD module's data interface may or may not be compatible. In some cases, the interface mismatch can be handled with glue logic implemented in the FPGA or through an external device such as a MAX II CPLD. For basic information on how to create logical interface glue logic, refer to "Using the MAX II Device".

For more information about the configuration options provided by the video sync generator peripheral, refer to the *Video Sync Generator and Pixel Converter Cores* chapter in volume 5 of the *Quartus II Handbook*.

Using the MAX II Device

If you use a pin-limited FPGA device for your application, you can use an additional programmable logic device to perform multiplexing and demultiplexing operations on the signals.

The NEEK board design uses this strategy to bridge both the physical and logical interfaces of the LCD module and the video sync generator component.

For more information about the multiplexing and demultiplexing operations on the NEEK board, refer to the *Nios II Standard Hardware for the Embedded Evaluation Board* chapter in the *Nios II Embedded Evaluation Kit, Cyclone III Edition User Guide*.

The design files for the MAX II design are available on the NEEK installation CD. For a list of the design examples and software source files included in this CD, refer to "Appendix A. NEEK Design Examples and Sources" on page 32.

Video Buffer Memory and SGDMA

The next step in porting the LCD controller is to determine the location of your video buffer in physical memory, and the best way to access this memory with the SGDMA peripheral. Based on your LCD module's display parameters, you must determine the following parameters for your video buffer:

- Size required for each frame buffer
- Total number of frame buffers to use
- Physical memory location of the frame buffers
- How the SGDMA should access the frame buffers

Size

The size of your video buffer depends on the color depth of the pixels, the number of pixels per frame, and the number of frame buffers you wish to use.

The color depth of the pixels is the number of bits used to encode a single pixel. In the case of the 4.3" Toppoly TD043MTEA1 display, the color depth is 24 bits per pixel, with 8 bits assigned to each of the colors red, green, and blue. Aligning data units on word boundaries is most efficient for the processor and the SGDMA. Therefore, consider using 32 bits per pixel, leaving the extra 8 bits unused.

The frame buffer size required to display a single frame of LCD data is determined by the following formula, in which frame buffer size is expressed in bytes, and frame height and frame width are expressed in numbers of pixels:

```
Frame buffer size = (bits per pixel/8) × frame height × frame width
```

The total video buffer memory size also depends on the number of frame buffers you wish to have in your system. As described in "Video Pipeline Subsystem API" on page 16, you can use multiple frame buffers to sequence data to the display. The total video buffer memory size is determined by the following formula, in which the video buffer memory size and frame buffer size are expressed in bytes:

```
Video buffer memory size = frame buffer size × number of frame buffers
```

For display data integrity, ensure that the memory peripheral in which your video buffer is located is at least as large as the video buffer memory size derived from the equations.

Location

After you determine the total video buffer size, you must select the video buffer location in physical memory. Select a memory in your system that can support the bandwidth required by your LCD module's interface. In SOPC Builder, you must connect the SGDMA peripheral `M_READ` and the Nios II processor's `DATA_MASTER` ports to the memory component, as shown in Figure 1 on page 2.

If you expect several peripherals in the system to share the video buffer memory, you may wish to tune the arbitration share mechanism for the memory peripheral that holds the video buffer. The arbitration shares parameter is configured in SOPC Builder. This parameter specifies how many transactions a master can transact with a slave before relinquishing control to another master. To ensure that the video pipeline subsystem can display pixel data without interruption, you must assign an adequate number of arbitration shares to the SGDMA peripheral `M_READ` master port that is connected to the video buffer memory peripheral.

All of the NEEK reference designs locate the video buffer in the same memory peripheral as the Nios II processor's execution memory. This sharing requires a memory component with adequate bandwidth, and careful tuning of the Nios II processor and SGDMA arbitration shares to the memory.

For more information about arbitration shares, refer to the *System Interconnect Fabric for Memory-Mapped Interfaces* chapter in volume 4 of the *Quartus II Handbook* and the *Avalon Memory-Mapped Design Optimizations* chapter in the *Embedded Design Handbook*.

SGDMA Access Mode and Parameters

The SGDMA peripheral moves pixel data from the video frame buffer to the rest of the video pipeline. You must determine the SGDMA peripheral's data width and memory location to store the SGDMA descriptors.

For more information about the SGDMA peripheral and its role in the NEEK LCD controller, refer to "NEEK LCD Controller Hardware Components" on page 2.

Data Width

For efficiency, the SGDMA peripheral should have a data width equal to the frame buffer memory peripheral's data width. This match ensures that the SGDMA peripheral reads the maximum possible amount of data

in each bus transaction. In the NEEK example designs, both the DDR SDRAM component that holds the frame buffer, and the SGDMA peripheral, have a width of 64 bits.

Descriptor Location

The SGDMA peripheral performs data transfers by accessing a list of descriptors from memory. The Nios II processor creates the descriptors, and the SGDMA peripheral's `descriptor_read` and `descriptor_write` master ports operate on them.

You must decide the location of the descriptors in memory. The memory you select must accommodate the total number of descriptors for all frame buffers in the system. The maximum required size is determined by the following formulas, in which frame buffer size is expressed in bytes:

```
Max bytes per descriptor = 65280
Descriptor size = 32 bytes
Descriptors per frame = frame buffer size / max bytes per descriptor,
                        rounded up to the nearest whole number
Total descriptor memory size =
    descriptors per frame × number of frame buffers × descriptor size
```

In the NEEK reference designs, the SGDMA peripheral's descriptors are located in the same memory peripheral as the Nios II processor's execution memory. This peripheral sharing does not impact system performance, because the descriptors do not require continuous access by the SGDMA peripheral for operation.

For more information about the SGDMA peripheral, refer to the *Scatter-Gather DMA Controller Core* chapter in volume 5 of the *Quartus II Handbook*.

Video Pipeline Peripherals

After you configure the video sync generator peripheral for your LCD module, and configure the SGDMA peripheral and frame buffer, you have completed the input and output portions of the LCD controller subsystem. The next step in the configuration process is to manage—buffer and convert—the flow of the pixel data from the frame buffer to the video sync generator peripheral.

The NEEK LCD controller reference design, shown in Figure 1 on page 2, includes a series of Avalon-ST hardware blocks to manage the pixel data flow. Each hardware block performs a specific task on a unit of data before passing it to the following hardware block.

The SGDMA and video sync generator peripherals are the initial and final stages, respectively, of the NEEK LCD controller's video pipeline subsystem. For more information about the SGDMA in the NEEK video pipeline subsystem, refer to "Scatter-Gather DMA (SGDMA) Controller Peripheral" on page 11. For more information about porting the SGDMA for your own design, refer to "Video Buffer Memory and SGDMA" on page 25. For more information about the video sync generator peripheral in the video pipeline subsystem, refer to "Video Sync Generator Peripheral" on page 5. For more information about porting the video sync generator peripheral for your own design, refer to "LCD Data Interface" on page 22.

Your LCD subsystem may have different requirements than the NEEK LCD reference design. However, your system probably requires that you perform one or more management functions on the data. The following sections describe the Avalon-ST hardware blocks that you can use to manage your pixel data flow.

Avalon-ST Timing Adapter

The Avalon-ST timing adapter peripheral is used as an adapter between two pipelined peripherals that have different ready latency values. When inserted between the upstream peripheral's output interface and the downstream peripheral's input interface, the timing adapter ensures a smooth flow of data between these peripherals.

SOPC Builder has a built-in facility to check whether any two peripherals in an Avalon-ST pipeline require timing adapters. If so, the tool warns you of this error condition. If you add a timing adapter between two peripherals, ensure that the timing latency values for the upstream and downstream peripherals are correct, as determined from the peripheral datasheet. Also, verify that the timing adapter's **Bits per Symbol** and **Symbols per Beat** values for the peripherals you are connecting are consistent with their protocols and functions.

For information about how the Avalon-ST timing adapter is used in the NEEK LCD controller's video pipeline subsystem, refer to "Avalon-ST Timing Adapter Peripheral (FIFO-to-Data Format Adapter)" on page 9 and "Avalon-ST Timing Adapter Peripheral (SGDMA to FIFO Memory)" on page 10.

For more information about the Avalon-ST timing adapter, refer to the *Avalon Streaming Interconnect Components* chapter in volume 4 of the *Quartus II Handbook*.

On-Chip FIFO Memory

The on-chip FIFO memory peripheral buffers pixel data between the SGDMA peripheral and the rest of the video pipeline. In the NEEK LCD controller's video pipeline subsystem, the on-chip FIFO memory peripheral is loaded with data by the SGDMA peripheral.

You should incorporate an on-chip FIFO memory peripheral in your video pipeline because it can prevent data underflow. Data underflow can occur if the SGDMA controller is temporarily prevented from accessing the video frame buffer (by another master accessing the frame buffer memory, latency due to bank switching in SDRAM, or other access contention issues). The FIFO memory helps prevent a data underflow condition by acting as a buffer, supplying video data to the rest of the video pipeline when the SGDMA cannot access new video data from the frame buffer memory.

When adding the on-chip FIFO memory peripheral in your system, you must ensure that the Avalon-ST port settings for **Bits per Symbol** and **Symbols per Beat** are set correctly, as follows:

```
SGDMA width = Bits per Symbol × Symbols per Beat
```

For most systems, the default value for **Bits per Symbol** is 8. Most memory peripherals have data widths that are multiples of 8 bits, and the SGDMA peripheral is designed to operate with this constraint. Therefore, any data passed to the on-chip FIFO memory has a multiple of 8 bits.

For information about how the on-chip FIFO memory peripheral is used in the NEEK LCD controller's video pipeline subsystem, refer to "FIFO Memory Peripheral" on page 9.

For more information about the Avalon-ST on-chip FIFO memory peripheral, refer to the *On-Chip FIFO Memory Core* chapter in volume 5 of the *Quartus II Handbook*.

Avalon-ST Data Format Adapter

The Avalon-ST data format adapter peripheral is typically used to modify the width of data units between different Avalon-ST peripherals. In the NEEK LCD controller reference design, two Avalon-ST data format adapters are included in the video pipeline subsystem. One Avalon-ST data format adapter peripheral converts a single 64- bit unit of data (8 data units × 8 bits) into two sequential 32- bit units of data (4 data units × 8 bits). The other instance converts a single 24-bit unit of data (3 data units × 8 bits) into three sequential 8-bit units of data (1 data unit × 8 bits).

Plan to incorporate an Avalon-ST data format adapter in your video pipeline subsystem if the data output width of an Avalon-ST peripheral does not match the input width on the following Avalon-ST peripheral in the pipeline.

For information about how the Avalon-ST data format peripheral is used in the NEEK LCD controller's video pipeline subsystem, refer to "Avalon-ST Data Format Adapter Peripheral (64 Bits to 32 Bits)" on page 8 and "Avalon-ST Data Format Adapter Peripheral (24 Bits to 8 Bits)" on page 7.

For more information about the Avalon-ST data format adapter peripheral, refer to the *Avalon Streaming Interconnect Components* chapter in volume 4 of the *Quartus II Handbook*.

Avalon-ST Pixel Converter

In the NEEK LCD controller video pipeline, the Avalon-ST pixel converter peripheral converts the pixel data in the pipeline from the 32-bit format to the 24-bit format that the NEEK LCD data interface requires.

You may need to include an Avalon-ST pixel converter in your design if your LCD module's pixel data format does not match the pixel data format in your video frame buffer.

Because the pixel data format required by LCD display modules varies widely, Altera provides only a minimal configuration GUI for the Avalon-ST pixel converter. To configure the Avalon-ST pixel converter component fully, you must edit an SOPC Builder-generated Verilog HDL file called **altera_avalon_pixel_converter.v** to specify the peripheral's pixel conversion behavior in your system.

For more information about how the Avalon-ST pixel converter peripheral is used in the NEEK LCD controller's video pipeline subsystem, refer to "Avalon-ST Pixel Converter Peripheral (BGR0 --> BGR)" on page 8.

For more information about creating a customized version of the Avalon-ST pixel converter peripheral, refer to the *Video Pipeline Data Flow* appendix in the *Nios II Embedded Evaluation Kit, Cyclone III Edition User Guide*.

Software Routines

Most of the software drivers that control the LCD subsystem are portable to diverse designs. This section provides porting notes for the software drivers. However, the best source of detailed information is the source files themselves. These files contain extensive comments describing the implementation, use, and porting of the functions.

LCD Module Drivers

The LCD controller and touch screen driver source code may not be directly portable to your LCD module, because many manufacturers use their own proprietary interfaces to the LCD controller and touch screen components on their LCD modules.

The NEEK LCD module's controller uses a proprietary three-wire interface for communication, and the touch-screen controller uses an SPI interface. The following sections briefly describe the device driver source code files.

LCD Panel Controller Sources

The LCD panel controller device driver implementation requires three lines from a PIO peripheral. The lines are used to implement a three-wire serial interface. Because this type of interface occurs frequently, the NEEK LCD controller source code may be functional in your design with only minor modifications.

For a list of the source files that implement the NEEK LCD controller device driver, refer to Table 13 on page 33.

Touch Screen Sources

The touch screen device driver implementation requires a HAL system clock timer alarm, a PIO line connected to the `pen_down` signal, and an IRQ-enabled SPI peripheral connected to the AD7843 digitizer chip. For a list of the source files that implement the NEEK LCD touch screen interface device driver, refer to Table 13 on page 33.

Video Pipeline Subsystem API

The video pipeline subsystem API should be portable to your system without modifications. For a list of the source files that implement the NEEK video pipeline subsystem API, refer to Table 13 on page 33.

When porting this API, you should be aware of the following settings in the **alt_video_display.h** file:

- `ALT_VIDEO_DISPLAY_MAX_BUFFERS`: This value must be greater than or equal to the number of buffers you intend to use.
- `ALT_VIDEO_DISPLAY_BLACK_8`: This macro is the 8-bit repeating color value to set the initial frame buffers to the color black.The current code assumes you repeat this value for each of the red, green, and blue components of the pixel value, to display the color black. In the NEEK LCD display, a pixel is black when you write the value `0` to each byte of the RGB color value.

  You may need to change this value for your particular LCD module.

Conclusion

LCD graphics modules are a popular user interface device in embedded systems. As the capabilities of LCD modules increase—additional pixels, larger color depth, and so on—designers need to develop more advanced LCD controllers. By implementing the LCD controller on FPGAs, you can quickly and easily accommodate the needs of new LCD modules.

Appendix A. NEEK Design Examples and Sources

The NEEK example designs include all of the hardware and source code files for implementing the LCD controller. The design examples and corresponding source code files are listed in Table 12 and Table 13.

In Table 12, a checkmark indicates the code base listed in the current row is used in the example design listed in the current column. A dash indicates the code is not used in the example design.

Table 12. NEEK Example Designs and Software Source Locations (Part 1 of 2)

Role (Directory Name)	Application Selector Utility	Mandelbrot C2H Application	Picture Viewer Application	Web Server Application
LCD Touch Panel (**alt_touchscreen**)	✓	✓	✓	—
LCD Controller (**alt_tpo_lcd**)	✓	✓	✓	—
Video Pipeline (**alt_video_display**)	✓	✓	✓	✓
Graphics Libraries (**graphics_lib**)	✓	—	✓	✓
Fonts (**fonts**)	✓	—	—	✓
Gimp support (**gimp_bmp**)	✓	—	—	—

Table 12. NEEK Example Designs and Software Source Locations (Part 2 of 2)

Role (Directory Name)	Application Selector Utility	Mandelbrot C2H Application	Picture Viewer Application	Web Server Application
Bitmap Library (**bmp**)	—	—	✓	—
JPEG Library (**jpeg**)	—	—	✓	—

The design examples are available on the NEEK installation CD. By default, they are installed in the directory <*Installation Path*>**/kits/cycloneIII_3c25_niosII_eval/examples**. Each design example in Table 12 has its own <*example*> directory under this directory. The software source files for each design example are located in the directory <*example*>**/software_examples/app**.

Table 13 lists the source code files for each of the peripherals in the NEEK LCD controller subsystem.

Table 13. NEEK Component Software Source Files (Part 1 of 2)

Role (directory name) /Source File	Description
LCD Touch Panel (**alt_touchscreen**)	Support for LCD touch panel
alt_touchscreen.h	Header file to support touch screen interface for LCD panel
alt_touchscreen.c	Touch screen interface for LCD panel
LCD Controller (**alt_tpo_lcd**)	Support for communicating with LCD controller
alt_tpo_lcd.h	Header file to support LCD controller interface for LCD panel
alt_tpo_lcd.c	LCD controller interface for LCD panel
alt_tpo_lcd_console.c	Console debug tool for LCD panel drivers
Video Pipeline (**alt_video_display**)	Frame buffer management for video pipeline
alt_video_display.h	Header file to support video pipeline components
alt_video_display.c	Video pipeline driver
Graphics Libraries (**graphics_lib**)	Altera simple graphics library
simple_graphics.h	Header file to support Altera simple graphics routines
simple_graphics.c	General purpose graphics routines
simple_text.c	Support for displaying text

Table 13. NEEK Component Software Source Files (Part 2 of 2)	
Role (directory name) /Source File	**Description**
Fonts (**fonts**)	
fonts.h	Header file to support font display
tahomabold_20.c	Source for Tahoma bold 20pt font
tahomabold_32.c	Source Tahoma bold 20pt font
Gimp support (**gimp_bmp**)	Used to support GIMP generated bitmaps
gimp_bmp.h	Header file to support Altera simple graphics library
gimp_bmp.c	Source for supporting rendering of GIMP generated bitmaps

Referenced Documents

This application note references the following documents:

- *Avalon Interface Specifications*
- *Avalon Memory-Mapped Design Optimizations* chapter in the *Embedded Design Handbook*
- *Avalon Streaming Interconnect Components* chapter in volume 4 of the *Quartus II Handbook*
- *Nios II Embedded Evaluation Kit, Cyclone III Edition User Guide*
- *On-Chip FIFO Memory Core* chapter in volume 5 of the *Quartus II Handbook*
- *Scatter-Gather DMA Controller Core* chapter in volume 5 of the *Quartus II Handbook*
- *System Interconnect Fabric for Memory-Mapped Interfaces* chapter in volume 4 of the *Quartus II Handbook*
- *System Interconnect Fabric for Streaming Interfaces* chapter in volume 4 of the *Quartus II Handbook*
- *Video Sync Generator and Pixel Converter Cores* chapter in volume 5 of the *Quartus II Handbook*

Document Revision History

Table 14 shows the revision history for this application note.

Table 14. Document Revision History		
Date and Document Version	**Changes Made**	**Summary of Changes**
May 2008 v1.0	Initial release.	—

101 Innovation Drive
San Jose, CA 95134
www.altera.com
Technical Support:
www.altera.com/support/