AN2641
Application note

Using the color LCD controller (CLCD) in the SPEAr embedded MPU family

Introduction

The SPEAr embedded MPU family is family of configurable MPUs, based on the ARM926 CPU core. Several members the SPEAr MPU family have an embedded ARM PL-110 Color-LCD controller (CLCD). This Application Note describes how to configure and operate any type of color LCD with the CLCD controller and provides application developers with troubleshooting information on common integration issues.

Color display devices play an important role in any embedded system, since usually displays are the most visible output devices.

In SPEAr, the CLCD is connected internally to the AHB bus and acts as a bus master-slave module. The CLCD performs translation of pixel-coded data into the required formats and timings to drive a variety of single/dual mono and color STN as well as color TFT LCD displays. The CLCD fetches the data for display from a frame buffer and uses its own dedicated DMA controller to transfer the display data to the LCD panel.

Contents

1 Overview of the CLCD controller 6

2 CLCD panel programmable parameters 7

2.1 Panel dimensions 7

2.1.1 X and Y resolution 7

2.1.2 Synchronization parameters 7

2.1.3 Dimension related registers 8

2.2 Panel clock source and frequency 9

2.2.1 Clock related registers 10

2.3 Panel display type 11

2.3.1 Display related registers 12

3 CLCD controller dataflow 13

3.1 Data related registers 15

4 Enabling and disabling the CLCD 16

5 CLCD interrupts 17

5.1 Interrupt causes 17

5.2 Interrupt related registers 17

6 Linux frame buffer 19

6.1 Frame buffer 19

6.2 Kernel level modifications for CLCD 20

6.3 User level perspective of CLCD 23

6.3.1 Data structures for user level 23

6.3.2 User level application sample 25

6.4 More details on the frame buffer in Linux 27

6.4.1 Data structures 27

6.4.2 A brief look at the source files 29

7 Touchscreen 30

7.1 Theory of operation 30

7.2 Linux input layer 31

7.3 Calibration process 31

8 Troubleshooting 33

8.1 Video flickering 33

8.2 Video tearing 33

8.3 White screen 33

9 Summary 34

Definitions 35

10 Revision history 36

List of tables

Table 1. Different display types and their properties 11
Table 2. Palette data storage 14
Table 3. Source code information on frame buffer 29
Table 4. Acronyms used in this document 35
Table 5. Document revision history 36

List of figures

Figure 1. A basic CLCD panel . . . 7
Figure 2. CLCD panel layout . . . 8
Figure 3. PLL2 connection for CLCD clock . . . 10
Figure 4. Data flow in CLCD controller . . . 13
Figure 5. CLCD OFF/ON sequence . . . 16
Figure 6. Advantage of using the frame buffer architecture . . . 19
Figure 7. CLCD panel structure . . . 21
Figure 8. Return CLCD panel structure . . . 21
Figure 9. Entry of CLCD info for menu config . . . 22
Figure 10. CLCD selection from menu config - 1 . . . 22
Figure 11. CLCD selection from menu config - 2 . . . 23
Figure 12. Structure fb_var_screeninfo . . . 24
Figure 13. Structure fb_fix_screeninfo . . . 24
Figure 14. Structure fb_cmap . . . 24
Figure 15. A sample user application . . . 26
Figure 16. Structure fb_info . . . 27
Figure 17. Structure fb_ops . . . 28
Figure 18. Touchscreen state machine . . . 30
Figure 19. Sequence of events from touchscreen driver . . . 31
Figure 20. Appearance of calibration tool for touchscreen . . . 32

1 Overview of the CLCD controller

Main features:

- Dual 16-deep programmable 32-bit wide FIFOs for buffering incoming display data
- Supports single and dual panel mono Super Twisted Nematic (STN) displays with 4 or 8-bit interfaces
- Supports single and dual-panel color and monochrome STN displays
- Supports Thin Film Transistor (TFT) color displays
- Resolution programmable up to 1024 x 768
- 15 gray-level mono, 3375 color STN, and 32K color TFT support
- 1, 2, or 4 bits-per-pixel (bpp) palettized displays for mono STN
- 1, 2, 4 or 8 bpp palettized color displays for color STN and TFT
- 16 bits-per-pixel (bpp) true-color non-palettized, for color STN and TFT
- 24 bpp true-color non-palettized, for color TFT
- Programmable timing for different display panels
- 256 entry, 16-bit palette RAM, physically arranged as a 128 x 32-bit RAM
- Frame, line and pixel clock signals
- AC bias signal for STN and data enable signal for TFT panels
- Patented gray scale algorithm
- Supports little-endian, big-endian or WinCE data formats
- LCD can support standard panel resolutions such as:
  - 320 x 200, 320 x 240
  - 640 x 200, 640 x 240, 640 x 480
  - 800 x 600
  - 1024 x 768

Types of LCD panel supported are:

- Active matrix TFT panels with up to 24-bit bus interface
- Single-panel monochrome STN panels (4-bit and 8-bit bus interface)
- Dual-panel monochrome STN panels (4-bit and 8-bit bus interface per panel)
- Single-panel color STN panels, 8-bit bus interface
- Dual-panel color STN panels, 8-bit bus interface per panel
- Compliance to the AMBA Specification (Rev 2.0) onwards for easy integration into the SPEAr MPU device

2 CLCD panel programmable parameters

To allow the CLCD controller to interface with the CLCD panel you intend to use in your application, you have to program a set of parameters, which varies according to the type of CLCD. For example, some parameters depend upon CLCD dimensions and synchronization, other parameters depend on the CLCD clock source and frequency, or the display panel type.

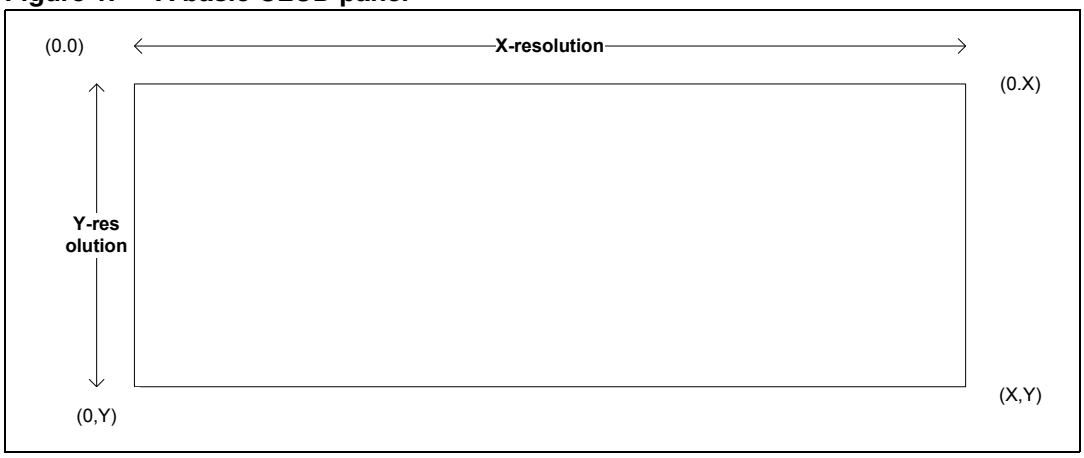
2.1 Panel dimensions

This section describes the parameters and the corresponding registers that depend on the LCD dimensions: X and Y resolution, vertical and horizontal synchronization.

2.1.1 X and Y resolution

The entire screen of CLCD is composed of pixels, each pixel a single dot. The number of pixels along the horizontal axis forms the X-resolution and the number of pixels along the vertical axis make up the Y-resolution. X-resolution and Y-resolution are also known respectively as the number of pixels per line and the number of lines per panel.

Figure 1. A basic CLCD panel



One common example is a 480 x 272 CLCD panel, which has 480 pixels horizontally and 272 pixels vertically.

2.1.2 Synchronization parameters

To draw any image on a CLCD panel, the screen is first scanned left to right is scanned and then the screen is scanned top to bottom. After each scan-line, the electron beam has to move back to the left side of the screen and to the next line, this is called the horizontal retrace. After the whole screen (frame) is painted, the beam moves back to the upper left corner, this is called the vertical retrace.

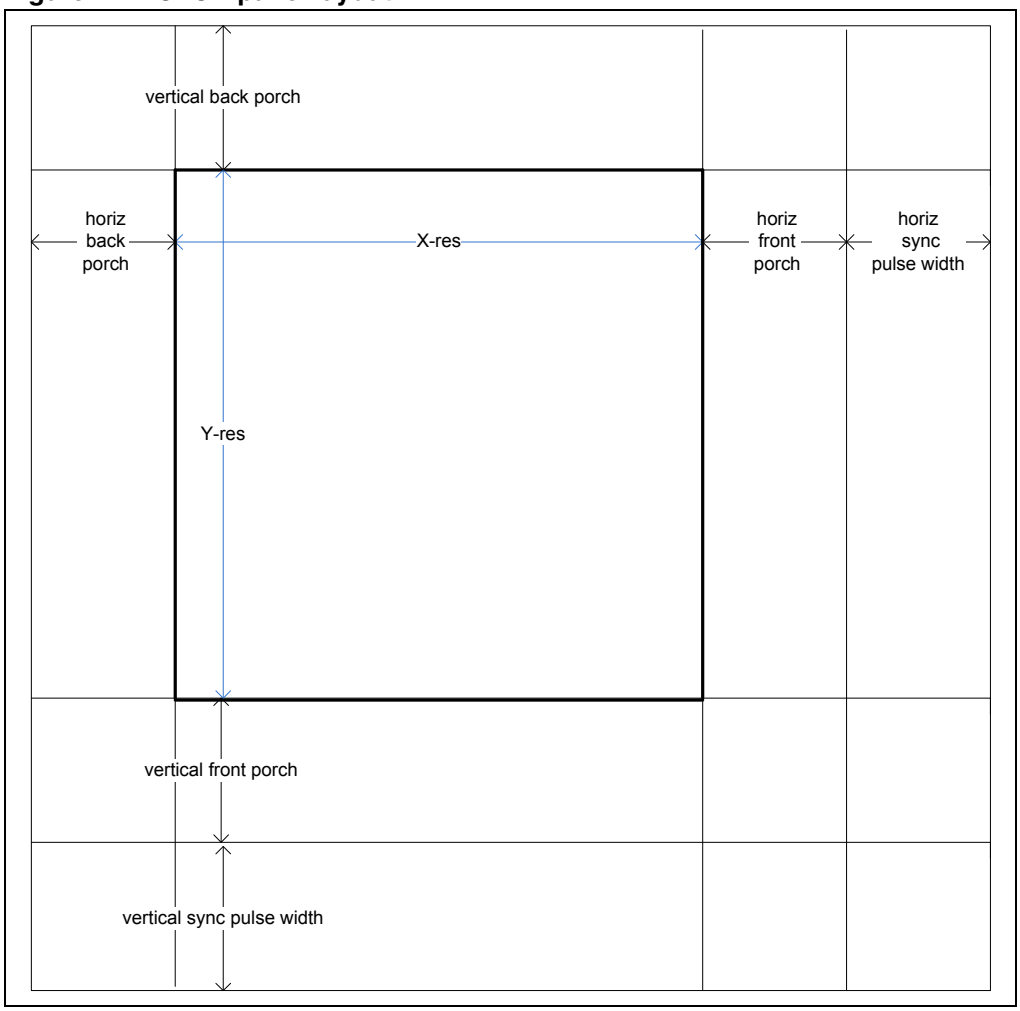
Since the CLCD panel does not know when a new scan-line starts, the CLCD controller generates a synchronization pulse (Horizontal Sync Width) for each scan-line. Similarly, it supplies a synchronization pulse (Vertical Sync Width) for each new frame. The position of the image on the screen is controlled by the timing of the synchronization pulses.

Figure 2 summarizes all the synchronization timings. The horizontal retrace time is the sum of the horizontal back porch, the horizontal front porch and the horizontal sync width, while the vertical retrace time is the sum of the vertical back porch, the vertical front porch and the vertical sync width.

The horizontal front porch is the time from picture to sync and horizontal back porch is the time from sync to picture.

The vertical front porch is the time from picture to sync and the vertical back porch is the time from sync to picture.

Figure 2. CLCD panel layout



2.1.3 Dimension related registers

1. **HBP, 8-bit [31:24] field in the LCD-Timing1 register**: specifies the number of pixel clock periods inserted at the beginning of each line or row of pixels. After the line clock for the previous line has been de-asserted, the value in HBP counts the number of pixel clocks to wait before starting the next display line. HBP can generate a delay of 1-256 pixel clock cycles.
2. **HFP, 8-bit [23:16] field in the LCD-Timing1 register**: sets the number of pixel clock intervals at the end of each line or row of pixels, before the LCD line clock is pulsed.

When a complete line of pixels is transmitted to the LCD driver, the value in HFP counts the number of pixel clocks to wait before asserting the line clock. HFP can generate a period of 1-256 pixel clock cycles.

3. **HSW, 8-bit [15:8] field in the LCD-Timing1 register**: specifies the pulse width of the line clock in passive mode, or the horizontal synchronization pulse in active mode.
4. **PPL, 6-bit [7:2] field in the LCD-Timing1 register**: is a value that represents between 16 and 1024 PPL. PPL controls how much data is read from the DMA input buffers through to the gray-scaler as follows:

   Actual pixels-per-line = 16 * (PPL + 1)

   The Actual pixels-per-line specifies the number of pixels in each line (or row) of the screen.
5. **VBP, 8-bit [31:24] field, in the LCD-Timing1 register:** specifies the number of line clocks inserted at the beginning of each frame. The VBP count starts just after the vertical synchronization signal for the previous frame has been negated for active mode, or the extra line clocks have been inserted as specified by the VSW bit field in passive mode. After this has occurred, the count value in VBP sets the number of line clock periods inserted before the next frame. VBP generates from 0-255 extra line clock cycles.
6. **VFP, 8-bit [23:16] field, in the LCD-Timing1 register**: specifies the number of line clocks to insert at the end of each frame. When a complete frame of pixels is transmitted to the LCD display, the value in VFP is used to count the number of line clock periods to wait.

   After the count has elapsed, the CLFP vertical synchronization signal, is asserted in active mode, or extra line clocks are inserted as specified by the VSW bit-field in passive mode. VFP generates from G0-255 line clock cycles.
7. **VSW, 6-bit [15:10] field, in the LCD-Timing1 register**: specifies the pulse width of the vertical synchronization pulse. The register is programmed with the number of line clocks in VSync minus one.
8. **LPP, 10-bit [9:0] field, in the LCD-Timing1 register**: specifies the total number of lines or rows on the LCD panel being controlled. LPP has an allowed range 1-1024 lines. The register is programmed with the number of lines per LCD panel minus 1. For dual panel displays this register is programmed with the number of lines on each of the upper and lower panels.

2.2 Panel clock source and frequency

CLCD_CLK is a free running reference clock (currently 48 MHz), which drives the CLCD controller.

CLCP, the output of the panel clock generator, is the CLCD panel clock frequency which goes from the CLCD controller to the CLCD panel. CLCP can be programmed in the range from CLCD_CLK/2 to CLCD_CLK/33 to match the BPP data rate of the LCD panel.

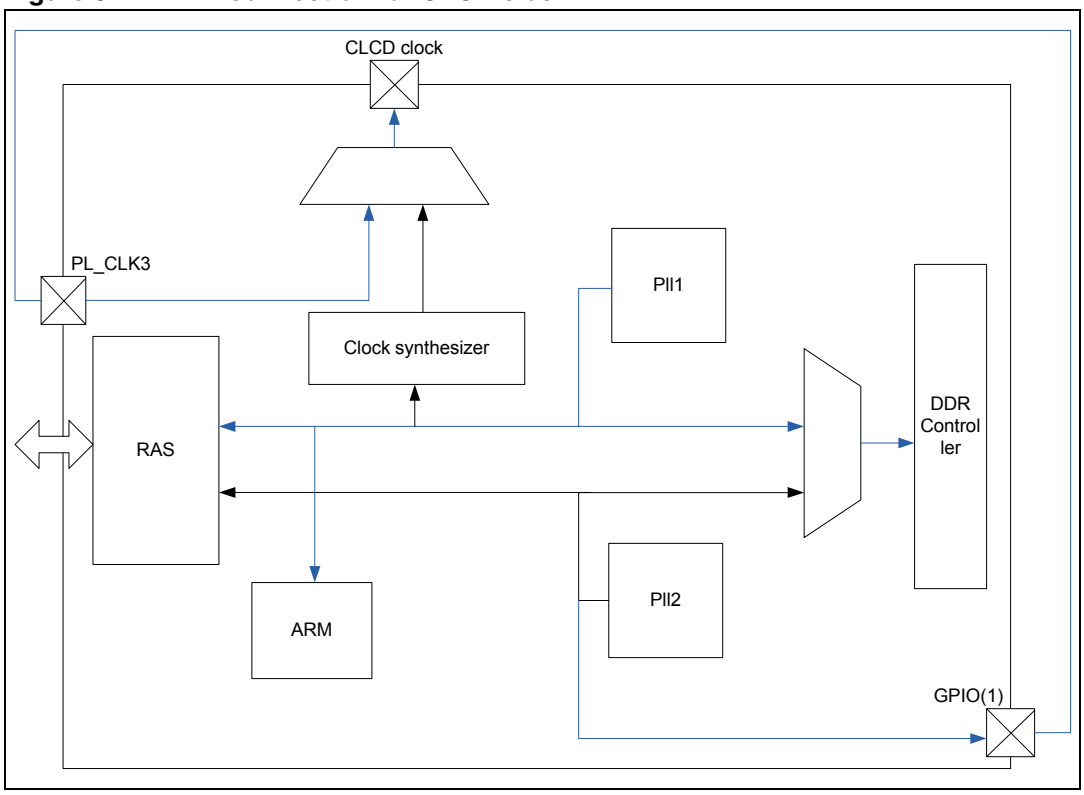
If the CLCD panel requires a frequency outside this range (CLCD_CLK/2 to CLCD_CLK/33), then an alternative route is to use another clock source (for example PLL2 or AHB) instead of CLCD_CLK and use a prescaler to obtain the desired frequency.

- One choice is to select the AHB as input clock. This can be done just using software modifications. You can select the AHB clock source using the CLKSEL bit and can divide the AHB frequency using the PCD bits. All these registers are described in the

next section. Using this technique it is possible to generate a frequency of AHB/2, AHB/3, and AHB/4...etc. for CLCP.

- Another choice is to use PLL2. The connection is shown in *Figure 3*.
  - In SPEAr600 the PLL2 output frequency can be taken directly from the clock synthesizer.
  - In SPEAr300, an additional connection at board level is required: GPIO (1) must be shorted to PL_CLK3.

Figure 3. PLL2 connection for CLCD clock



2.2.1 Clock related registers

1. **CLKSEL, bit [5] in the LCD-timing-2 register,** drives CLCDCLKSEL which is the select signal for the external LCD clock multiplexer.
2. **Clocks per Line (CPL) field [25:16] in the LCD-Timing2 register**, specifies the number of actual CLCP clocks for each line of the LCD panel. This is the number of PPL divided by 1 for TFT displays.
3. **Panel Clock Divisor (PCD), bits [31:27] & [4:0] in the LCD-Timing2 register**, is used to derive the LCD panel clock frequency CLCP from the CLCDCLK frequency as follows:

   $$CLCP = CLCDCLK / (PCD+2)$$

   For TFT, the PCD can be bypassed by setting the BCD bit.
4. **Bypass pixel clock divider (BCD), bit [26] in the LCD-Timing2 register.** Setting this to 1 bypasses the PCD logic. This is mainly used for TFT displays.
5. **Invert Output Enable (IOE) bit [14] in the LCD-Timing2 register**, used to select the active polarity of the output enable signal in TFT mode. In this mode, the CLAC pin is

used as an enable that indicates to the LCD panel when valid display data is available. In active display mode, data is driven onto the LCD data lines at the programmed edge of CLCP when CLAC is in its active state.

0 = CLAC output pin is active HIGH in TFT mode

1 = CLAC output pin is active LOW in TFT mode

6. **Invert panel clock (IPC) bit [13] in the LCD-Timing2 register**, used to select the edge of the panel clock on which pixel data is driven out onto the LCD data lines.

   0 = Data is driven on the LCD data lines on the rising edge of CLCP

   1 = Data is driven on the LCD data lines on the falling edge of CLCP

7. **Invert Horizontal Synchronization (IHS) bit [12] in the LCD-Timing2 register,** used to invert the polarity of the CLLP signal.

   0 = CLLP pin is active HIGH and inactive LOW

   1 = CLLP pin is active LOW and inactive HIGH

8. **Invert Vertical Synchronization (IVS) bit [11] in the LCD-Timing2 register**, is used to invert the polarity of the CLFP signal.

   0 = CLFP pin is active HIGH and inactive LOW

   1 = CLFP pin is active LOW and inactive HIGH.

9. **AC bias pin frequency (ACB) bits [10:6] in the LCD-Timing2 register,** applies only to STN displays, which require the pixel voltage polarity to be periodically reversed to prevent damage due to DC charge accumulation. Program this field with the required value minus 1 to apply the number of line clocks between each toggle of the AC bias pin, CLAC. This field has no effect if the CLCD controller is operating in TFT mode when the CLAC pin is used as a data enable signal.

2.3 Panel display type

The CLCD controller supports the following types of LCD panels:

- **TFT (Thin Film Transistor) display:** Active matrix display panels which require a digital color value of each pixel to be applied to the display data inputs.
- **STN (Super Twisted Nematic) display:** Passive matrix display panels that require algorithmic pixel pattern generation to provide pseudo gray scaling on mono or color creation for display (for which a dedicated hardware element, grey scalar, is provided).

Table 1. Different display types and their properties

LCD type	Single/dual panel	Palettized/non-palettized	Mono/color	BPP
TFT	Single	Non-Palettized	Color	24
TFT	Single	Non-Palettized	Color	16
TFT	Single	Palettized	Color	1, 2, 4 and 8
STN	Single and Dual	Non-Palettized	Color	16
STN	Single and Dual	Palettized	Color	1, 2, 4 and 8
STN	Single and Dual	Palettized	Mono	1,2 and 4

2.3.1 Display related registers

1. **LCD TFT, bit [5] in the LCD-Control register**, selects the display type. For TFT, set to 1. For STN, set to 0.
2. **LCD Dual, bit [7] in the LCD-Control register**, selects dual or single STN panel. For dual panel STN, set to 1, for single panel STN, set to 0. If TFT panel is used then set to 0.
3. **LCD Mono bit [6] in the LCD-Control register,** controls whether monochrome STN LCD uses a 4 or 8-bit parallel interface. Set to 0 if mono LCD uses 4-bit interface and set to 1 if mono LCD uses 8-bit interface. This bit has no meaning in other modes and must be programmed to zero.
4. **LCD BW bit [4] in the LCD-Control register,** selects monochrome or color STN panel. If STN LCD is monochrome (black and white), set to 1. If STN LCD is color, set to 0. This bit has no meaning in TFT mode.
5. The color palette register is described in next section.

3 CLCD controller dataflow

Packets of pixel-coded data are first stored in memory (frame buffer). Then, they are fed, via the AHB interface, to dual 16-deep 32 bit-wide DMA FIFO, both of which act as input data flow buffers. The buffered pixel-coded data is then unpacked via a pixel serializer.

Depending on the LCD type and mode, the unpacked data may represent:

- an actual true display gray or color value or
- an address in a 256 x 16 bit wide palette RAM containing a gray or color value

In the case of STN displays, either a palette value or the true value is passed to the gray scaling generators. The hardware coded gray scale algorithm logic sequences the addressed pixels activity over a programmed number of frames to provide the effective display appearance.

For TFT display, either an addressed palette value or true color value is passed directly to the output display drivers.

Figure 4. Data flow in CLCD controller

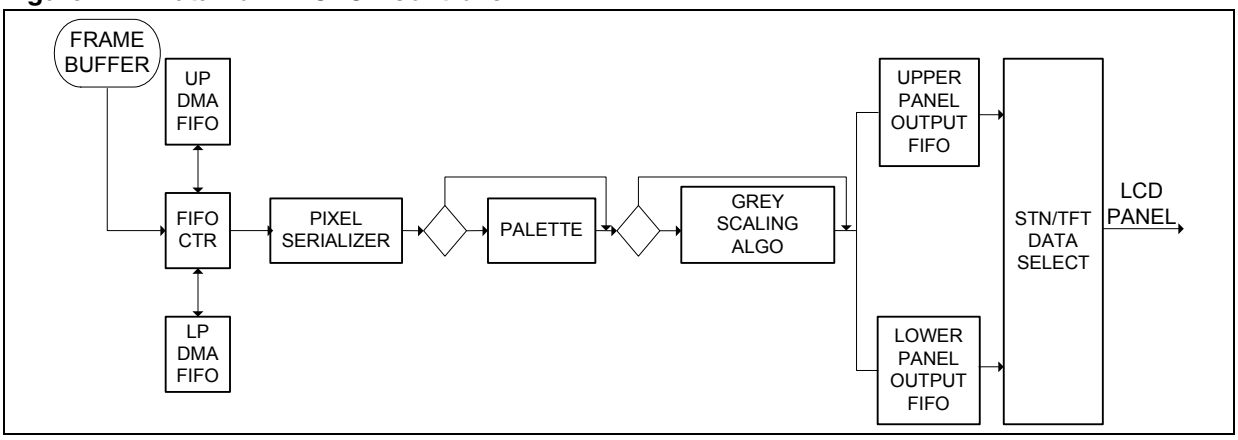


Figure 4 shows the Data Flow in the respective hardware blocks. There are 2 decision boxes.

First decision box - using the palette RAM:

- Dependent upon the LCD type and code, the unpacked data may represent:
  - An actual true display gray or color value (16 BPP or higher) or
  - An address to a 256 x 16 bit wide palette RAM gray or color value (1, 2, 4 or 8 BPP)
- Palette use is linked to display type and mode. A palette is used by:
  - Mono STN displays with 1, 2 or 4 BPP
  - Color STN or TFT displays with 1, 2, 4, 8 BPP

Otherwise, non-palettized true-color is used.

Second decision box - using the grey scaling algorithm:

- In the case of STN displays, either a palette value or the true value is passed to the gray scaling generators. The hardware coded grey scale algorithm logic sequences the

activity of the addressed pixels over a programmed number of frames to provide the effective display appearance.

- For TFT displays, either an addressed palette value or true color value is passed directly to the output display drivers, always bypassing the grey scale algorithm logic.

Three major components of the data flow are:

1. **Pixel Serializer,** this block reads the 32-bit wide LCD data from output port of the DMA FIFO and extracts 24, 16, 8, 4, 2 or 1 BPP data, depending on the current mode of operation. The CLCD controller supports big endian, little endian and WinCE data formats.

   Dependent upon the operation mode, the extracted data may be used to point to a color/gray scale value in the palette RAM or may actually be a true color value that can be directly applied to an LCD panel.

2. **Palette RAM** is a 256 x 16 bit dual port RAM (physically structured as 128 x 32 bits) with independent controls and addresses for each port. Port1 is used as a read/write port and is connected to the AMBA AHB slave interface. The palette entries can be written and verified through this port. Port2 is used as a read-only port and is connected to the gray scalar.

   Table 2 shows the bit representation of each word in the palette.

Table 2. Palette data storage

Bit	Name	Description
[31]	I	Intensity/unused
[30:26]	B [4:0]	Blue palette data
[25:21]	G [4:0]	Green palette data
[20:16]	R [4:0]	Red palette data
[15]	I	Intensity/unused
[14:10]	B [4:0]	Blue palette data
[9:5]	G [4:0]	Green palette data
[4:0]	R [4:0]	Red palette data

3. **A gray scale algorithm** drives mono and color STN panels. This provides 15 gray scales for mono displays. In the case of STN color displays, the three color components (RGB) are gray scaled simultaneously which results in 3375 (15 x 15 x 15) available colors. The gray scalar transforms each 4-bit gray value into a sequence of activity-per-pixel over several frames to give the representation of gray scales and colors.

3.1 Data related registers

1. **Panel frame base address register,** is the CLCD DMA frame address read/write register, which is used to program the base address of the frame buffer.

   For example, if frame data is at a given location X then you just have to write X in this register. This location X must be word-aligned.

   Bits [1:0] are reserved but you do not need to explicitly shift the address before writing to this register. You simply write the base address and the shift is handled automatically by the controller hardware.
2. **Panel current address value register** is a read-only register that contains an approximate value of the upper and lower panel data DMA addresses when read. The registers can change at any time and therefore can be used for testing or for a coarse delay mechanism.
3. **BEBO (big-endian byte order), bit [9] in the LCD-control register**, selects the byte order. For little-endian byte order, set to 0. For big-endian byte order, set to 1.
4. **BEPO (big-endian pixel ordering within a byte), bit [10] in the LCD-control register**. Setting 0 will do little-endian pixel ordering within a byte and setting 1 will do big-endian pixel ordering within a byte. The BEPO bit selects between little and big-endian pixel packing for 1, 2 and 4 BPP display modes. It has no effect on 8 or 16 BPP pixel formats.
5. **BGR, bit [8] in the LCD-control register** does RGB or BGR selection. Setting to 0 will have RGB as normal output and 1 will be BGR (red and blue swapped)
6. **BPP (bits-per-pixel) bits [3:1] in the LCD-control register,** select the BPP. The controller can support 1, 2, 4, 8, 16, and 24 BPP.

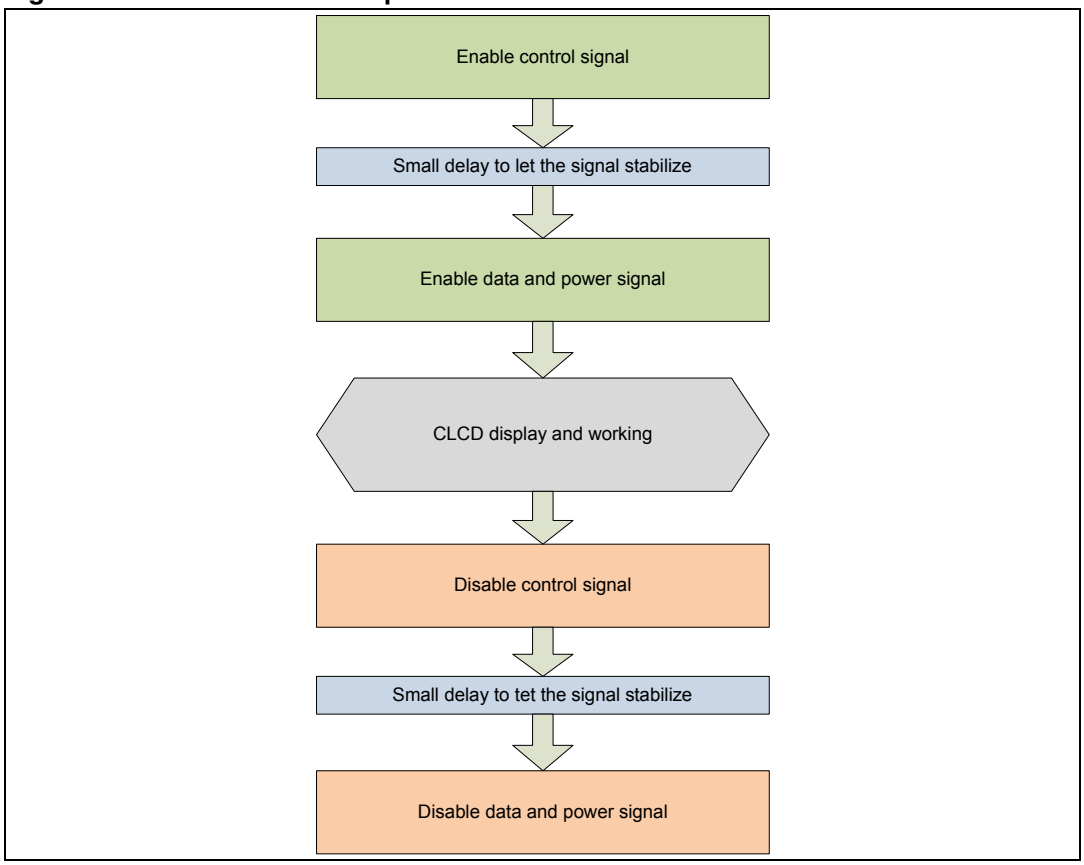
   24 BPP (means each pixel can be represented by 24 bits) is for TFT only. However, we actually use 32 bits, so 24 bits are for RGB (8 bits each) and the remaining 8 bits are reserved or can be used for transparency.

4 Enabling and disabling the CLCD

The procedure for enabling and disabling the CLCD is shown in *Figure 5*. The delay for signal stabilization depends on the display type.

Caution: If the delay is too short or non existent, the panel may not work properly and sometimes hangs with a white screen.

Figure 5. CLCD OFF/ON sequence



- Enable the control signals by setting the LCD controller enable bit [0] in the LCD-control register to 1. To disable the control signals, set this bit to 0.
- Connect the power to the CLCD panel by setting LCD power enable bit [11] in the LCD-control register to 1. To disconnect the power from the panel, set this bit to 0.

5 CLCD interrupts

The CLCD interrupt is connected the vectored interrupt controller (VIC) as follows:

- SPEAR600: IRQ45 (CLCD_INTR)
- SPEAr300: IRQ30 (generic Interrupt #3 from RAS)

5.1 Interrupt causes

There are 4 kinds of event that can cause an CLCD interrupt:

1. DMA FIFO underflow:

   The FIFO underflow interrupt is asserted when data is requested from an empty DMA FIFO. Note that the size of the DMA FIFO is 32 words.

   Setting Watermark Level bit as 1 will cause this interrupt when DMA FIFO has eight or more empty locations. Setting Watermark Level bit as 0 will cause this interrupt when DMA FIFO has four or more empty locations.

   The interrupt requests the display data from memory, to fill the FIFO to above the programmed watermark.

2. Base address update:

   The LCD base address update interrupt is asserted when the 'LCD panel frame buffer base' values are transferred to the 'Current LCD panel frame buffer base' incrementors. This signals to the system that it is safe to update the 'LCD panel frame buffer base' registers with new frame base addresses if required.

   This interrupt is mainly used to reprogram the base address when generating double-buffered video.

3. Vertical compare:

   The vertical compare interrupt is asserted when one of four vertical display regions, is reached. By setting the following values in the LCD V-comp bit in the Control Register the interrupt can be generated at the:

   00 = Start of vertical synchronization

   01 = Start of back porch

   10 = Start of active video

   11 = Start of front porch

4. Bus Error:

   The master bus error interrupt is asserted when an ERROR response is received by the master interface during a transaction with a slave. When this error is encountered, the master interface enters an error state and remains in this state until clearance of the error has been signaled to it (possibly by the interrupt service routine).

- A single combined interrupt is asserted if any of the above four interrupt lines is asserted and each interrupt event is handled respectively by the interrupt service routine.

5.2 Interrupt related registers

There are four registers which are used for handling Interrupts:

1. **Interrupt clear register** is a write-only register. Writing logic 1 to the relevant bit clears the corresponding interrupt. This register is mainly used by the interrupt service routine to clear the interrupt, once it is handled.
2. **Raw interrupt status register** is a read-only register. On a read it returns five bits that can generate interrupts when set. This register can be used to check whether an interrupt occurred.
3. **Interrupt mask set/clear register** is used to enable the corresponding raw interrupt in the 'Raw Interrupt Status register' (bit values) to be passed to the 'Masked Interrupt Status register'. This read/write register can disable or enable an interrupt.
4. **Masked interrupt status register** is a read-only register. It is a bit-by-bit logical AND of the 'Interrupt Mask Set/Clear register' and the 'Raw Interrupt Status register'. Interrupt lines correspond to each interrupt. This register can be used to check whether an interrupt is masked.

6 Linux frame buffer

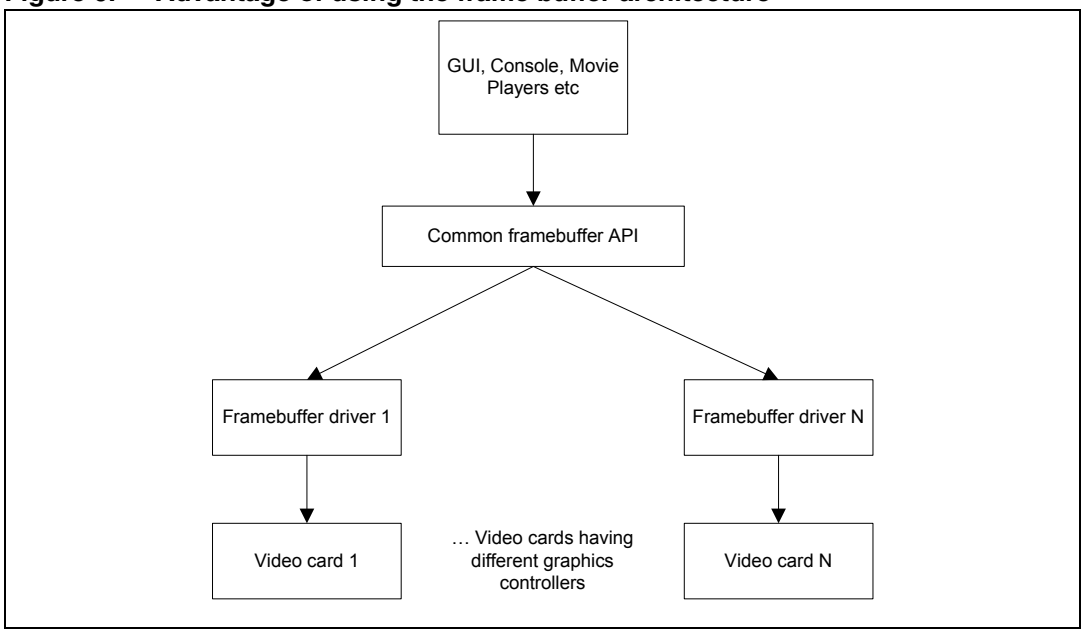
This section describes how to use the CLCD in a Linux environment from user space, what modifications are needed in the kernel when adding a new CLCD in Linux, and explains the concept of the frame buffer.

6.1 Frame buffer

The frame buffer is a part of the Linux Kernel and should not be confused with the Panel Data Frame buffer register of the CLCD controller (as discussed in*Section 3.1*).

The frame buffer device provides a general abstraction for the graphics hardware and hence allows application software to access the graphics hardware through a well-defined programming interface, so the application software doesn't need to know anything about the low-level (hardware register) stuff. *Figure 6* show the advantage of frame buffer which allows applications to be independent of the vagaries of the underlying graphics hardware. Applications run unchanged over diverse types of video hardware if they and the display drivers conform to the frame buffer interface.

Figure 6. Advantage of using the frame buffer architecture



If GUIs (Graphic User Interface) such as MiniGUI, MicroWindows are used, LCD device drivers must be implemented as Linux frame buffer device drivers in addition to those low level operations which only deal with commands provided by LCD controllers.

- The device is accessed through special device nodes, usually located in the /dev directory, and have names such as /dev/fb0 (or /dev/fb/0), /dev/fb1 etc. The frame buffer devices are also `normal' memory devices, this means one can read and write their contents. You can, for example, even make a screen snapshot using:

  cat /dev/fb0 > myfile

- Application software that uses the frame buffer device (for example the X server) can use /dev/fb0 by default. One can specify an alternative frame buffer device by setting

the environment variable $FRAMEBUFFER to the path name of a frame buffer device, e.g. (for sh/bash users):

export FRAMEBUFFER=/dev/fb1

or (for csh users):

setenv FRAMEBUFFER /dev/fb1

After this the X server will use the second frame buffer.

6.2 Kernel level modifications for CLCD

The kernel can be changed to accommodate information about a new CLCD panel. The files which need to changed are:

For SPEAr300:

'arch/arm/mach-spear300/spear300.c' (or 'arch/arm/mach-spearbasic/spearbasic.c') and 'drivers/video/Kconfig'.

For SPEAr600:

'arch/arm/mach-spear600/spear600.c' (or 'arch/arm/mach-spearbasic/spearplus.c').

Step-by-step procedure:

1. The file 'arch/arm/mach-spear300/spear300.c' (or 'arch/arm/mach-spearbasic/spearbasic.c') fills the clcd_panel structure with all the information about CLCD.

Figure 7. CLCD panel structure

```
#ifdef CONFIG_FB_ARMCLCD_SHARP_LQ043T1DG01
static struct clcd_panel sharp_LQ043T1DG01_in = {
      .mode          = {
      .name          = "Sharp LQ043T1DG01",
       .refresh      = 0,
       .xres         = 480,
       .yres         = 272,
       .pixclock     = 48000,
       .left_margin  = 2,
       .right_margin = 2,
       .upper_margin = 2,
       .lower_margin = 2,
       .hsync_len    = 41,
       .vsync_len    = 11,
       .sync         = 0, //FB_SYNC_HOR_HIGH_ACT |
       .vmode        = FB_VMODE_NONINTERLACED,
   },
   .width        = -1,
   .height       = -1,
   .tim2         = TIM2_IOE | TIM2_CLKSEL | 3, //TIM2_CLKSEL| 4,
   .cntl         = CNTL_LCDTFT | CNTL_BGR,
   .bpp          = 32,
};
#endif
```

The 'clcd_panel' structure has 4 important members: tim2 (corresponding to LCD-Timing register 2), tim3 (corresponding to LCD-Timing register 3), cntl (corresponding to LCD-Control register) and bpp (bits per pixel).

The 'mode' field (structure 'fb_videomode') of structure 'clcd_panel' has the kind of information (panel resolution and margins) which needs to be filled in LCD-timing register 0 and LCD-Timing register 1.

2. Return the appropriate CLCD panel structure by filling in the following function:

Figure 8. Return CLCD panel structure

```
{
  Struct clcd_panel *panel;
  #ifdef CONFIG_FB_ARMCLCD_SHARP_LQ043T1DG01
  Panel = &sharp_LQ043T1DG01_in;
  #endif
  Return panel;
}
```

3. Note the usage of macros like CONFIG_FB_ARMCLCD_SHARP_LQ043T1DG01. They are defined in 'drivers/video/Kconfig', the file which defines the choices in the menuconfig.

Figure 9. Entry of CLCD info for menu config

```
choice
   depends on FB_ARMCLCD
   prompt "LCD Panel"
   default FB_ARMCLCD_SHARP_LQ043T1DG01

config FB_ARMCLCD_SHARP_LQ043T1DG01
   bool "SHARP LQ043T1DG01 CLCD 4.2\" TFT(480x272)"
   help
     This is an implementation of the Sharp LQ043T1DG01, a 4.2"
     color High Performance TFT panel.
     The native resolution is 480x272.
```

4. To select CLCD from menu config, just go to:

   Device Drivers → Graphics support → Support for frame buffer devices →

   Figure 10 and *Figure 11* show how you can select CLCD panel from the menu. After saving, the menu can be exited.

   Just to verify whether CLCD is properly selected, verify that the .config file has:

   – CONFIG_FB_ARMCLCD=y

   – CONFIG_FB_ARMCLCD_SHARP_LQ043T1DG01=y

Now do, 'make uImage' and to generate a new kernel image containing the information on the new CLCD panel.

Figure 10. CLCD selection from menu config - 1

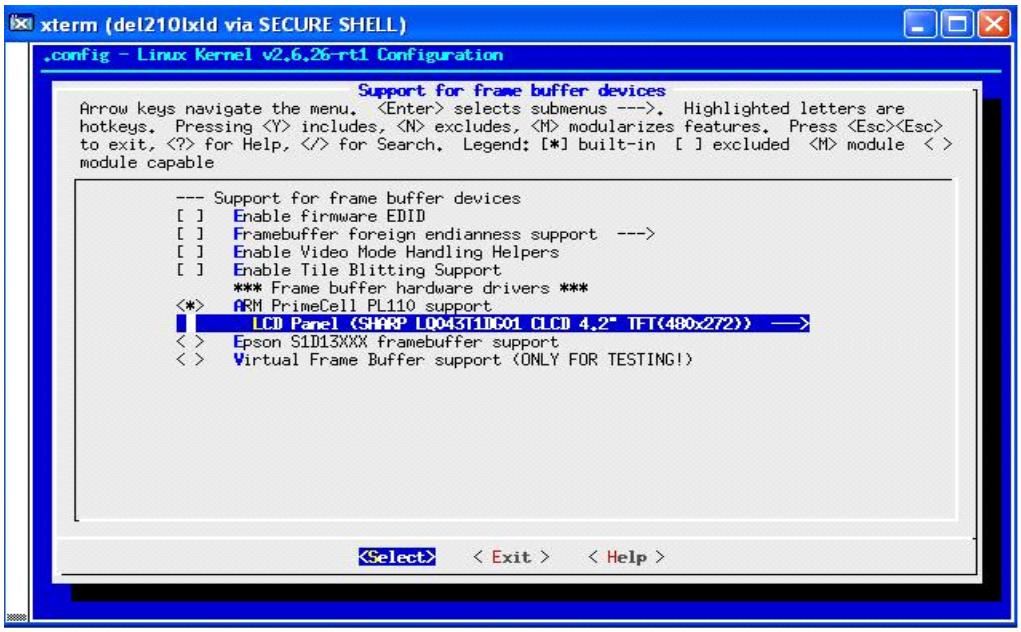
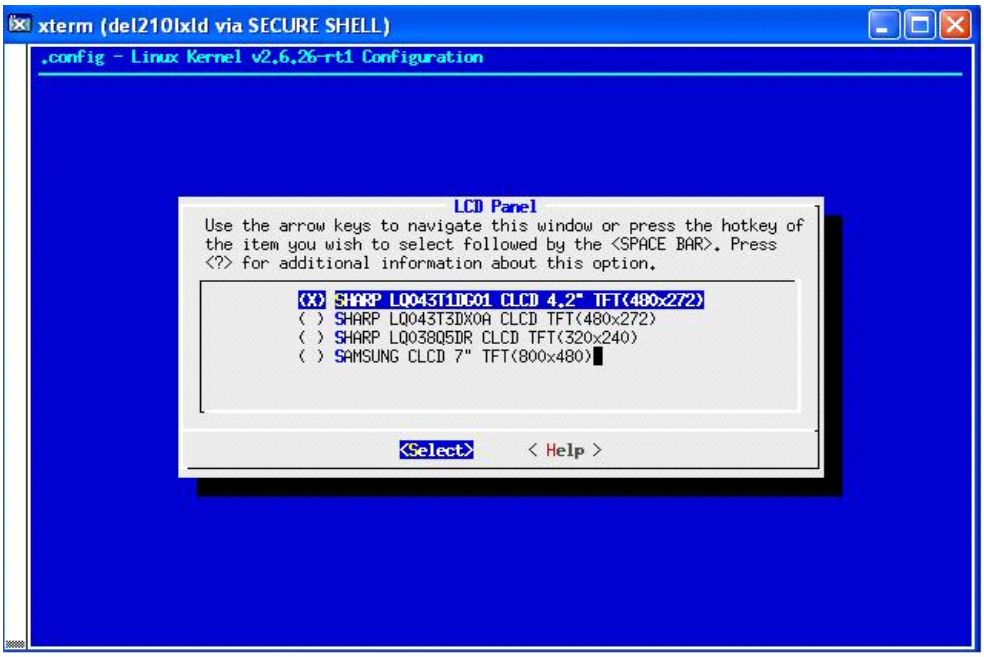


Figure 11. CLCD selection from menu config - 2



6.3 User level perspective of CLCD

As we already know, a frame buffer device is a memory device like /dev/mem and it has the same features. You can read it, write it, seek to some location in it and mmap() it (the main usage). The difference is just that the memory that appears in the special file is not the whole memory, but the frame buffer of the video hardware.

6.3.1 Data structures for user level

It is essential to understand the following three structures, they are used by all frame buffer user applications.

1. Variable information pertaining to the video card is held in struct fb_var_screeninfo. This structure contains fields such as the X-resolution, Y-resolution, bits required to

hold a pixel, pixclock, HSYNC duration, VSYNC duration, and margin lengths. These values are programmable by the user:

Figure 12. Structure fb_var_screeninfo

```
struct fb_var_screeninfo {
    __u32 xres; /* Visible resolution in the X axis */
    __u32 yres; /* Visible resolution in the Y axis */
    /* ... */
    __u32 bits_per_pixel; /* Number of bits required to hold a pixel */
    /* ... */
    __u32 pixclock; /* Pixel clock in picoseconds */
    __u32 left_margin; /* Time from sync to picture */
    __u32 right_margin; /* Time from picture to sync */
    /* ... */
    __u32 hsync_len; /* Length of horizontal sync */
    __u32 vsync_len; /* Length of vertical sync */
    /* ... */
};
```

2. Fixed information about the video hardware, such as the start address and size of frame buffer memory, is held in struct fb_fix_screeninfo. These values cannot be altered by the user:

Figure 13. Structure fb_fix_screeninfo

```
struct fb_fix_screeninfo {
    char id[16]; /* Identification string */
    unsigned long smem_start; /* Start of frame buffer memory */
    __u32 smem_len; /* Length of frame buffer memory */
    /* ... */
};
```

3. The fb_cmap structure specifies the color map, which is used to convey the user's definition of colors to the underlying video hardware. You can use this structure to define the RGB (Red, Green, and Blue) ratio that you desire for different colors: Device independent color-map information. You can get and set the color-map using the FBIOGETCMAP and FBIOPUTCMAP ioctls:

Figure 14. Structure fb_cmap

```
struct fb_cmap {
    __u32 start; /* First entry */
    __u32 len; /* Number of entries */
    __u16 *red; /* Red values */
    __u16 *green; /* Green values */
    __u16 *blue; /* Blue values */
    __u16 *transp; /* Transparency */
};
```

6.3.2 User level application sample

The *Figure 15* shows a simple user application that works over the frame buffer API. The program shows 3 color bands on the screen by operating on /dev/fb0, the frame buffer device node corresponding to the display.

After opening the frame buffer device node the program gets the fixed screen and variable screen information. It first deciphers the visible resolutions and the bits per pixel in a hardware-independent manner using the frame buffer ioctl FBIOGET_VSCREENINFO. This interface command gleans the display's variable parameters by operating on the 'fb_var_screeninfo' structure.

The command FB_ACTIVATE_FORCE (while put variable screen information) enables the CLCD. This line is important to activate the CLCD.

The program then goes on to mmap() the frame buffer memory and writes color data on each constituent pixel bit.

Figure 15. A sample user application

```
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <linux/fb.h>
#include <sys/mman.h>
int main()
{
   int fbfd = 0;
   struct fb_var_screeninfo vinfo;
   struct fb_fix_screeninfo finfo;
   long int screensize = 0;
   char *fbp = 0;
   int x = 0, y = 0;
   long int location = 0;

   // Open the file for reading and writing
   fbfd = open("/dev/fb0", O_RDWR);
   if (!fbfd) {
      printf("Error: cannot open framebuffer device.\n");  exit(1);
   }
   // Get fixed screen information
   if (ioctl(fbfd, FBIOGET_FSCREENINFO, &finfo)) {
      printf("Error reading fixed information.\n");  exit(2);
   }
   // Get variable screen information
   if (ioctl(fbfd, FBIOGET_VSCREENINFO, &vinfo)) {
      printf("Error reading variable information.\n");  exit(3);
   }
   // Put variable screen information to Switch ON CLCD Panel
   vinfo.activate |= FB_ACTIVATE_FORCE | FB_ACTIVATE_NOW;
   if (ioctl(fbfd, FBIOPUT_VSCREENINFO, &vinfo)) {
      printf("Error writing variable information.\n");  exit(3);
   }
   // Figure out the size of the screen in bytes
   screensize = vinfo.xres * vinfo.yres * vinfo.bits_per_pixel / 8;
   // Map the device to memory
   fbp = (char *)mmap( 0, screensize, PROT_READ | PROT_WRITE, MAP_SHARED, fbfd, 0 );
   if ((int)fbp == -1) {
      printf("Error: failed to map framebuffer device to memory.\n");  exit(4);
   }
   //Loop will draw 3 color bands: blue, green and red
   for ( location = 0, y = 0; y < vinfo.yres; y++ )
      for ( x = 0; x < vinfo.xres; x++ ) {
         if ( y < (vinfo.yres/3)  )          {
            *(fbp + location)   = 0xFF;//show Blue
            *(fbp + location+1) = 0;//no Green
            *(fbp + location+2) = 0;//no Red
            *(fbp + location+3) = 0;//No Transparency
         }
         else if ( y < (2*vinfo.yres/3)  )    {
            *(fbp + location) =  0xFF;// no Blue
            *(fbp + location + 1) = 0;// show Green
            *(fbp + location + 2) = 0;// no Red
            *(fbp + location + 3) = 0;// No transparency
         }
         else      {
            *(fbp + location)   = 0;// no Blue
            *(fbp + location + 1) = 0;  // no Green
            *(fbp + location + 2) = 0xFF;// show Red
            *(fbp + location + 3) = 0;// No transparency
         }
         location +=4;
      }
   munmap(fbp, screensize);
   close(fbfd);
   return 0;
}
```

6.4 More details on the frame buffer in Linux

6.4.1 Data structures

1. The 'fb_info' structure defines the current state of the video card. 'fb_info' is only visible from the kernel. Inside fb_info, there is a structure called 'fb_ops' which is a collection of functions needed to make the driver work.

   'struct fb_info' is the central data structure used by frame buffer drivers. This structure is defined in include/linux/fb.h as follows:

Figure 16. Structure fb_info

```
struct fb_info {
    /* ... */
    struct fb_var_screeninfo var; /* Variable screen information*/
    struct fb_fix_screeninfo fix; /* Fixed screen information*/
    /* ... */
    struct fb_cmap cmap; /* Color map*/
    /* ... */
    struct fb_ops *fbops; /* Driver operations*/
    /* ... */
    char __iomem *screen_base; /* Frame buffer's virtual address */
    unsigned long screen_size; /* Frame buffer's size */
    /* ... */
    /* From here on everything is device dependent */
    void *par; /* Private area */
};
```

   The memory for fb_info is allocated by framebuffer_alloc(), a library routine provided by the frame buffer core. This function also takes the size of a private area as an argument and appends it to the end of the allocated fb_info. This private area can be referenced using the par pointer in the fb_info structure.

2. The user application program can use the ioctl() system call to operate low level LCD hardware. Methods defined in structure 'fb_ops' are used to support these operations.

   The 'fb_ops' structure contains the addresses of all entry points provided by the low-level frame buffer driver. The first few methods in fb_ops are necessary for the functioning of the driver, while those remaining are optional ones that provide for graphics acceleration. The responsibility of each function is briefly explained in the comments:

Figure 17. Structure fb_ops

```
struct fb_ops {
    struct module *owner;
    /* Driver open */
    int (*fb_open)(struct fb_info *info, int user);
    /* Driver close */
    int (*fb_release)(struct fb_info *info, int user);
    /* ... */
    /* Sanity check on video parameters */
    int (*fb_check_var)(struct fb_var_screeninfo *var, struct fb_info *info);
    /* Configure the video controller registers */
    int (*fb_set_par)(struct fb_info *info);
    /* Create pseudo color palette map */
    int (*fb_setcolreg)(unsigned regno, unsigned red, unsigned green, unsigned blue, unsigned transp, struct fb_info *info);
    /* Blank/unblank display */
    int (*fb_blank)(int blank, struct fb_info *info);
    /* ... */
    /* Accelerated method to fill a rectangle with pixel lines */
    void (*fb_fillrect)(struct fb_info *info, const struct fb_fillrect *rect);
    /* Accelerated method to copy a rectangular area from one screen region to another */
    void (*fb_copyarea)(struct fb_info *info,const struct fb_copyarea *region);
    /* Accelerated method to draw an image to the display */
    void (*fb_imageblit)(struct fb_info *info, const struct fb_image *image);
    /* Accelerated method to rotate the display */
    void (*fb_rotate)(struct fb_info *info, int angle);
    /* Ioctl interface to support device-specific commands */
    int (*fb_ioctl)(struct fb_info *info, unsigned int cmd, unsigned long arg);
    /* ... */
};
```

6.4.2 A brief look at the source files

/dev/fb0 also allows several IOCTL commands to interface it, so that a whole range of information about the hardware can be queried and set. The color map handling works via IOCTLs, too. Look into <linux/fb.h> for more information on what IOCTLs exist and on which data structures they work. Here's just a brief overview:

Table 3. Source code information on frame buffer

Directory or files	Purpose
drivers/video/	The frame buffer core layer and low-level frame buffer drivers reside in this directory
include/linux/fb.h,	Generic frame buffer structures are defined in this directory
include/video/.	Chipset-specific headers are kept in this directory
drivers/video/fbmem.c	Creates the /dev/fbX character devices and is the front end for handling frame buffer ioctl commands issued by user applications

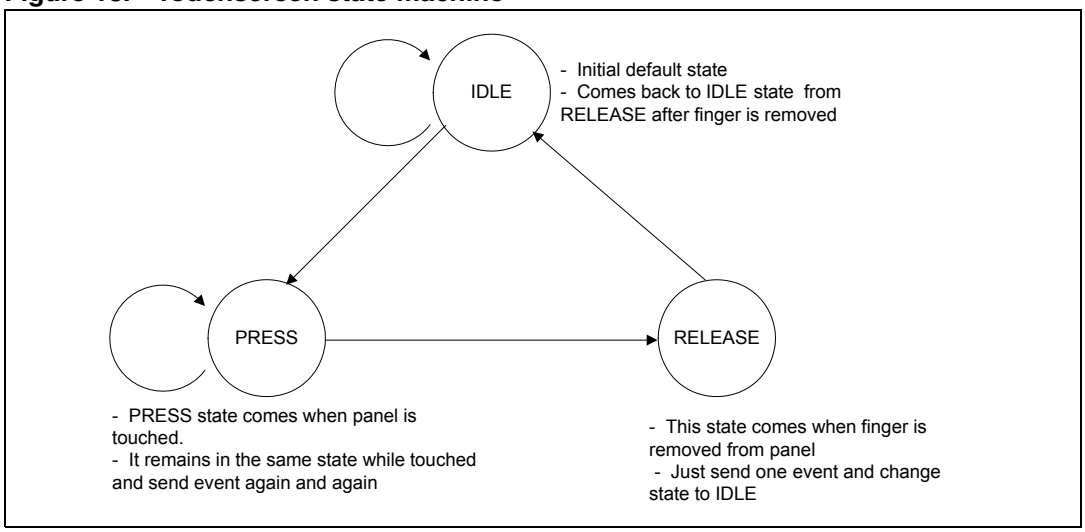
7 Touchscreen

7.1 Theory of operation

SPEAr uses a combination of ADC and GPIO for touchscreen operation. By outputting GPIO high the device-driver reads the X-coordinate from ADC channel number 5 and again by outputting GPIO low it reads the Y-coordinate from ADC channel number 6. There is no dedicated touchscreen controller in SPEAr.

The touchscreen driver has a state machine which is called every 100 milliseconds and reads each ADC channel. The state machine has 3 states and works as shown in *Figure 18*.

Figure 18. Touchscreen state machine



The touchscreen driver is probed as an input driver during Linux boot-up. At this time it only opens the ADC. The State machine starts only after the CLCD is activated (at user level this is typically done by an FB_ACTIVATE_FORCE call).

Just after Activation of the CLCD the ADC channels are read to get the idle values when touchscreen is not touched. This gives an estimated threshold value.

Once the state machine is started, it reads the values from ADC channels (by polling in a 100 millisecond loop) and once it gets values above threshold, the touchscreen is assumed to be touched (or pressed).

Once the touchscreen is pressed, the driver keeps sending coordinates unless the values from the ADC channels go lower. This event means that the touchscreen is untouched (or released).

The polling time (currently 100 milliseconds) of the driver is an important parameter because it determines the touchscreen driver response time.

The touchscreen driver can be selected from menu config as follows:

Device Drivers → Input device support → Touchscreens →

[*] Touchscreen for SPEAr based platforms

- The name of the touchscreen device is '/dev/input/ts' with the major number 13 and minor number 64.

7.2 Linux input layer

The kernel's input subsystem layer was created to unify scattered drivers that handle diverse classes of data-input devices such as keyboards, mice, and touch screens.

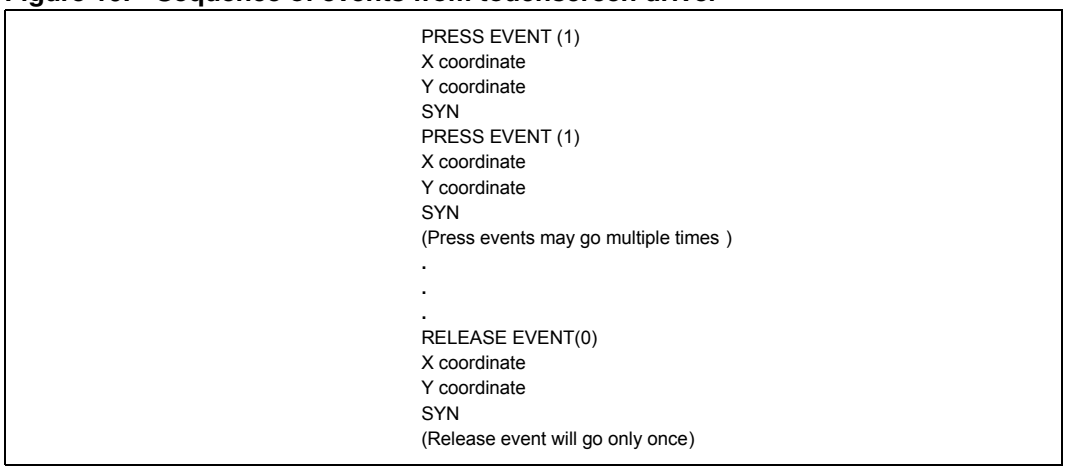
The kernel's input subsystem layer permits the uniform handling of functionally similar input devices even when they are physically different. For example, all mice, such as PS/2, USB or Bluetooth, are treated alike.

It provides an easy event interface for dispatching input reports to user applications. The driver does not have to create and manage /dev nodes and related access methods. Instead, it can simply invoke input APIs to send mouse movements, key presses, or touch events upstream to the user layer. Applications such as X Windows work seamlessly over the event interfaces exported by the input subsystem layer.

The name of the touchscreen driver is "drivers/input/touchscreen/spear_ts.c" (or "drivers/input/touchscreen/spr_ts_st.c"). To integrate with the input subsystem layer, the driver fills the structure 'struct input_dev *input_dev' and register through the call input_register_device( ).

The sequence of events sent to the application layer is as illustrated in *Figure 19*:

Figure 19. Sequence of events from touchscreen driver



7.3 Calibration process

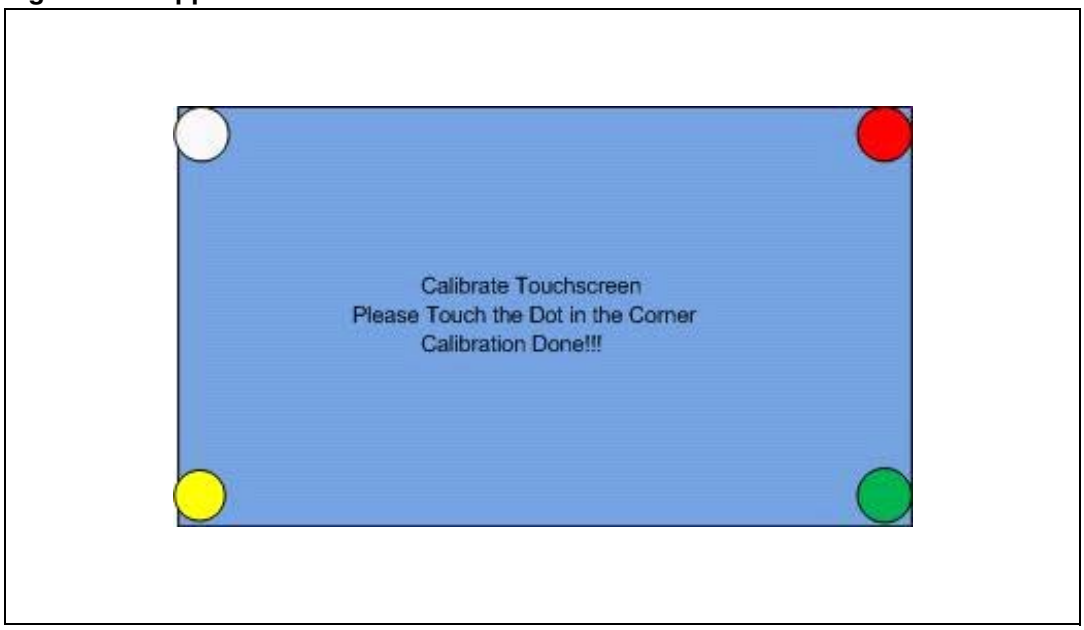
Any CLCD panel touchscreen cannot return absolutely correct coordinates, rather it only returns default values which need to be adjusted to the CLCD panel dimensions. This is the purpose of a calibration tool.

The calibration tool works with a simple concept, it asks the user to touch four corner points of the CLCD panel and stores these values. It then learns the offsets and scales used to later calculate the correct coordinates. Calibration may vary for different CLCD panels so before using the touchscreen, the CLCD panel has to be calibrated.

Once calibration is successfully performed, the calibration data is written in a file in a particular format. It is up to the user application to use the information provided in the file to calculate correct coordinates.

Figure 20 shows the calibration tool display after the calibration process is over.

Figure 20. Appearance of calibration tool for touchscreen



8 Troubleshooting

Here are some solutions to common issues that can occur during development of a CLCD driver.

8.1 Video flickering

Flickering is caused by the CLCD master not having enough bandwidth over the AHB memory system. In this case you need to configure the priority of the MPMC ports properly.

Solution: The Xloader takes care of this and puts CLCD port at the highest priority.

8.2 Video tearing

When displaying video frames if the rate is not fast enough, then while we can see a new frame, the disappearing of the old frame can also be perceived by the human eye. Such an overlapping is known as video tearing.

Solution: In this case you should change the driver implementation and add the dual frame buffer capability. One buffer will be the current one, while the CPU prepares the second one. Then the driver should switch the CLCD pointer.

8.3 White screen

If a CLCD application activates the panel at regular intervals then after a certains number of iterations (this issue has been observed after 20 iternations) a white screen appears and CLCD hangs unless power down occurs.

Solution: As described in *Section 4: Enabling and disabling the CLCD on page 16* , there two steps required for CLCD activation:

1. Control signal on
2. Data signal on.

A small delay (in milliseconds) is required between both to let the control signal stabilize.

9 Summary

In summary, before you start to integrate a new CLCD panel in the SPEAr system, you mainly need three types of information:

- Type of CLCD panel
- Dimension-related parameters
- Clock-related parameters

The next step is to make the architectural modifications in Linux. You must know exactly where the CLCD architectural information is in Linux. You should have a basic knowledge of the frame buffer layer of Linux as well. All this information is given the different sections of this application note.

The last step is to create a user level application so that the CLCD panel can be tested. This is architecture independent and should be the same for all types of LCD panel.

If you are using a touchscreen, it must be calibrated before using it as an input device.

Definitions

Table 4. Acronyms used in this document

CLCD	Color Liquid Crystal Display
CLCP	CLCD panel clock requency
CLCD_CLK	A free running reference clock
CLKSEL	Clock select
TFT	Thin film transistors
STN	Super twisted nematic
HSW	Horizontal synchronization width
HFP	Horizontal front porch
HBP	Horizontal back porch
PPL	Pixels per line
LPP	Lines per panel
VSW	Vertical synchronization pulse width
VFP	Vertical front porch
VBP	Vertical back porch
BCD	Bypass pixel divider
CPL	Clocks per line
IOE	Input output enable
IPC	Invert panel clock
HIS	Invert horizontal synchronization
IVS	Invert vertical synchronization
PCD	Panel clock divisor
ACB	AC bias pin frequency
LEE	LCD line end enable

10 Revision history

Table 5. Document revision history

Date	Revision	Changes
29-Oct-2009	1	Initial release.