**Freescale Semiconductor**
Application Note

Document Number: AN4182
Rev. 0, 08/2010

# Different Display Configurations on the i.MX31 Linux PDK

*by Multimedia Application Division*
*Freescale Semiconductor, Inc.*
*Austin, TX*

This application note provides the necessary information, considerations, and procedures to add or adapt a new Liquid Crystal Display (LCD) panel to the Board Support Package (BSP) distribution for the i.MX31 Product Development Kit (PDK). The application note describes the general LCD panel information and generalities of the display controller module. The application note also describes the development process to adapt a new LCD panel to the BSP, considering that the framework driver structure is already provided by the operating system.

**NOTE**

This application note assumes that the reader is familiar with the Linux Target Image Builder (LTIB) and Linux device driver development concepts.

**Contents**

## 1 Introduction

The i.MX31, which is a multimedia processor, supports several types of displays. The display devices are handled by a special module called the Image Processing Unit (IPU).

The IPU module handles other graphic interfaces that include cameras and 2D graphics accelerators. These are the IPU submodules that are connected by a private DMA interface (IDMA) and is used by the IPU to transfer data among the modules and between the IPU and external memory.

Figure 1 shows the functional diagram of the IPU module.

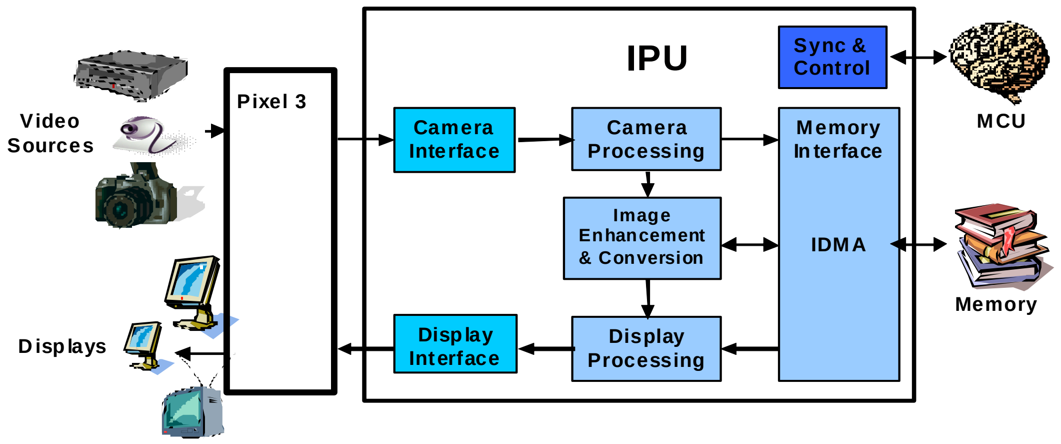


**Figure 1. IPU Functional Diagram**

The process of selecting an LCD for a mobile device involves several conflicts with respect to the requirements. Some of these conflicts are as follows:

- Large amount of data, implying high rate of data transfer and processing that requires significant resources
- Flexibility to support various use cases
- Size, cost, and power consumption

Freescale provides reference designs for the i.MX family where the functionalities of the LCD devices are demonstrated. However, developers find many reasons to replace the display in their products. Features such as screen size, resolution, weight, power consumption, and price are important in a commercial multimedia product. Also, many LCD displays become obsolete quickly and is difficult to find the same LCD panel that is included in the reference design. This application note is intended only for the dumb displays and displays that do not have the sharp synchronous interface. However, the application note also provides useful information about the smart displays.

## NOTE

Do not confuse sharp LCDs with sharp interface. There are plenty of sharp panels that do not use sharp interface.

# 2 LCD Generalities

This section describes the principles of the LCD devices.

## 2.1 LCD Basics

LCDs are electronic devices that contain array of pixels. These pixels can be either color or monochrome and are created with a special material that changes the light characteristics that passes through them. These devices are not able to emit light, and therefore, the backlight unit is also shipped with the panel to create a full functional display device.

### 2.1.1 Resolution

In this application note, the term resolution refers to the number of pixels in an LCD array. The resolution has two dimensions—horizontal and vertical. Table 1 lists the most common video resolution standards available in the market.

**Table 1. Video Resolution Standards**

| Video Name | Description | Width (pixels) | Height (pixels) | Aspect Ratio |
|---|---|---|---|---|
| CGA | Color Graphics Adapter | 320 | 200 | 8:5 |
| QVGA | Quarter VGA | 320 | 240 | 4:3 |
| VGA | Video Graphics Array | 640 | 480 | 4:3 |
| NTSC | National Television System Committee | — | — | — |
| PAL | Phase Alternating Line (TV) | — | — | 4:3 |
| WVGA | Wide VGA | 800 | 600 | 5:3 |
| SVGA | Super VGA | 800 | 600 | 4:3 |

**NOTE**

The maximum resolution supported by the i.MX31 processor is SVGA. Therefore, resolutions greater than SVGA are not included in Table 1.

The resolutions mentioned in Table 1 refer to the landscape orientation of the LCD panels. In the landscape orientation, the horizontal pixels are more than the vertical pixels. However, portrait orientated LCD panels are also available in the market with the same standard resolution where the horizontal and vertical sizes are inverted. Therefore, the portrait LCD panels have more vertical pixels than the horizontal pixels.

Figure 2 shows the portrait and landscape orientations in an LCD panel.

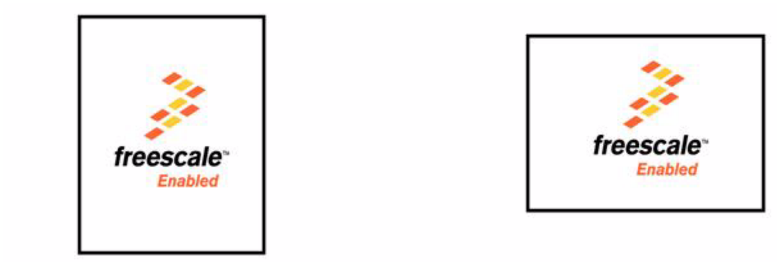


**Figure 2. Portrait and Landscape Orientations**

It is important to select a proper orientation for an LCD panel as both the electronic and optical features are optimized for the applications that use native orientation for the panel. Besides the optical characteristics, the dumb displays include an embedded LCD controller that draws the pixels from left to right and top to bottom. To show images or videos on the LCD panel using a non-native orientation, the display contents are processed to store the image in a buffer. The image is rewritten in a way that the LCD controller expects the pixel information to be sent. This operation is called rotation and the i.MX31 processor includes hardware to perform this operation.

**NOTE**

It is recommended to select an LCD panel that uses native orientation to avoid additional image processing.

Figure 3 shows the non-native portrait and landscape orientations in an LCD panel.

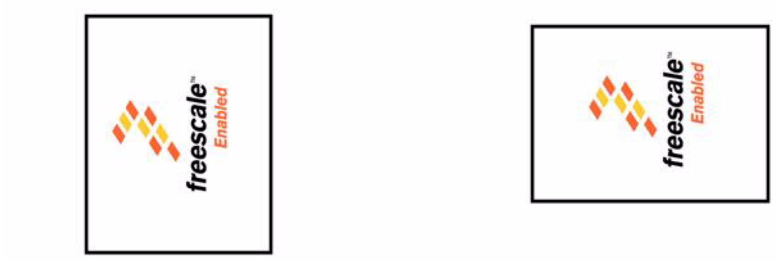


**Figure 3. Non-native Portrait and Landscape Orientations**

**NOTE**

In a non-native orientation, the rotation can be 90°, 180°, or 270°. Also, the frame should be rotated before sending to the display.

### 2.1.2 Size

The size of an LCD panel is measured diagonally in inches. Since the size directly impacts the pixel width, it is common to assume that the size of a VGA (640 × 480) panel to be larger than a QVGA (320 × 240) panel as the VGA panel has greater number of pixels compared to QVGA. However, this is not always true. LCD manufacturing processes allow the size and resolution to be independent variables. It is difficult to determine the size of a panel from its resolution alone. Large screens tend to consume more power than the smaller ones and affect the size and weight of the final product. Also, higher resolutions on smaller LCD panels can complicate the visibility of on-screen objects for the final user. Therefore, based on the information available in the data sheet, it is difficult to determine if a particular LCD panel fits the application. It is recommended to view the LCD in any other reference design or demo before selecting an LCD panel for any application.

### 2.1.3 Color Spaces

A color space is a way to represent colors. There are two main color spaces—RGB (RGB444, RGB565, RGB666, RGB888, and RGBA8888) and YUV (YUV 4:4:4, YUV 4:2:2, and YUV 4:2:0). The i.MX31 processor supports working with both the color spaces. However, the display panels can receive data only by using the RGB interface.

## 2.2 LCD Types

This section describes the different types of LCD displays.

### 2.2.1 Synchronous Panel (Dumb Display)

The dumb displays or synchronous displays are panels that require microprocessor to send data continually. In these panels, screen refresh is performed by sending the complete frame data continuously. In general, the smart displays are more expensive than the dumb displays, and this is one of the reasons for the common usage of synchronous panels in a final product. This application note focuses on Thin Film Transistor (TFT) LCD panels that belongs to a special group of synchronous panels.

### 2.2.2 Asynchronous Panel (Smart Display)

The advantage of smart displays is that the i.MX31 processor is required to send the display data only when the image is changed. In these panels, the images can be sent at any time and the screen refresh is handled by the embedded smart LCD display controller.

# 3 IPU-SDC Generalities

This section describes the IPU Synchronous Display Controller (SDC) generalities.

## 3.1 LCD Interfaces

This section describes the various LCD interfaces with examples.

### 3.1.1 Synchronous Display Interface

The i.MX31 SDC can be configured to handle four different types of devices—TFT monochrome, TFT color, YUV progressive, and YUV interlaced. However, this application note is focused only on the synchronous TFT color interface. The i.MX31 processor provides a 22-line interface for the TFT color device and is described in Table 2.

**Table 2. Synchronous Display Interface Signals**

| Signal | IPU Signal | Description |
|---|---|---|
| HSYNC | DISPB_D3_HSYNC | horizontal synchronization |
| VSYNC | DISPB_D3_VSYNC | vertical synchronization |
| DRDY | DISPB_D3_DRDY | data enable or data ready |
| PIXCLK | DISPB_D3_CLK | pixel clock |
| Red Data[5:0] | DISPB_DATA[17:12] | pixel red component |
| Green Data[5:0] | DISPB_DATA[11:6] | pixel green component |
| Blue Data[5:0] | DISPB_DATA[5:0] | pixel blue component |

The signals and interfaces in the synchronous display system are as follows:

| | |
|---|---|
| HSYNC | Horizontal Synchronization (HSYNC) signal is also known as FPLINE or LP. This signal indicates the end of a line and the following valid pixels are part of the next line. |
| VSYNC | Vertical Synchronization (VSYNC) signal is also known as FPFRAME, FLM, SPS, or TV. Active state of this signal indicates the end of the current frame. The LCD display should then restart the line index to zero to draw the next valid data in the first line of the panel. |
| DRDY | The active state of the Data Ready (DRDY) or Data Enable (DE) signal indicates the LCD device that the data in the RGB bus is valid and should be latched using the Pixel Clock (PIXCLK) signal. When the DE signal is active, a pixel corresponding to the color described in the RGB bus is drawn with every PIXCLK pulse. This signal always has the value of the screen width. |
| PIXCLK | The polarity of the PIXCLK signal specifies if the RGB data is placed on the bus during the rising or falling edges. Based on the polarities, the operations are performed in the following ways: • High polarity indicates that the data is written on the RGB bus during the falling edges and data is latched into the LCD panel during the rising edges. This is valid only when the DE signal is active. • Low polarity indicates that the data is written on the RGB bus during the rising edges and data is latched into the LCD panel during the falling edges. This is valid only when the DE signal is active. |
| RGB data | The RGB data is transferred to the display through the display interface. Though the i.MX31 processor can internally use different types of bits per pixel, such as RGB565, RGB666, RGB888, and RGBA8888, the display interface is limited to 18 lines. Therefore, any image or video that contains pixels more than 18 bits (RGB666) cannot be sent to the display. During |

| | |
|---|---|
| | the RGB to RGB conversion, the less significant bits are removed from the pixel and the remaining bits are directly sent to the display interface. |

**NOTE**

Dithering or filtering actions are not performed in this process.

| | |
|---|---|
| Extra signals | There are also other signals that are included in the panel interface. These signals are not part of the display data interface but are required for a full functional module. For example, it is common that some panels require a reset signal and initialization commands. These commands are sent by a serial interface, such as $I^2C$ or Serial Peripheral Interface (SPI). Also, the embedded touch panel and backlight unit are shipped along with the module. |
| SPI interface | Some LCD displays require an initialization routine through a serial interface that can be 3-wire, 4-wire, or 5-wire. A serial interface (SD_D_CLK, LCS1, SD_D_IO, and SD_D_I) in the i.MX31 IPU is not used to send serial commands to the LCDs. This interface is not intended for a general purpose usage and is used only by the IPU when two or more asynchronous displays are configured to use the serial interface. |

### 3.1.2 Examples of Synchronous Display Interfaces

This section describes a few synchronous display interfaces.

### 3.1.2.1 i.MX31 PDK Epson L4F00242T03 2.7" VGA LCD Interface

Figure 4 shows the interface between the i.MX31 processor and Epson L4F00242T03 VGA panel.

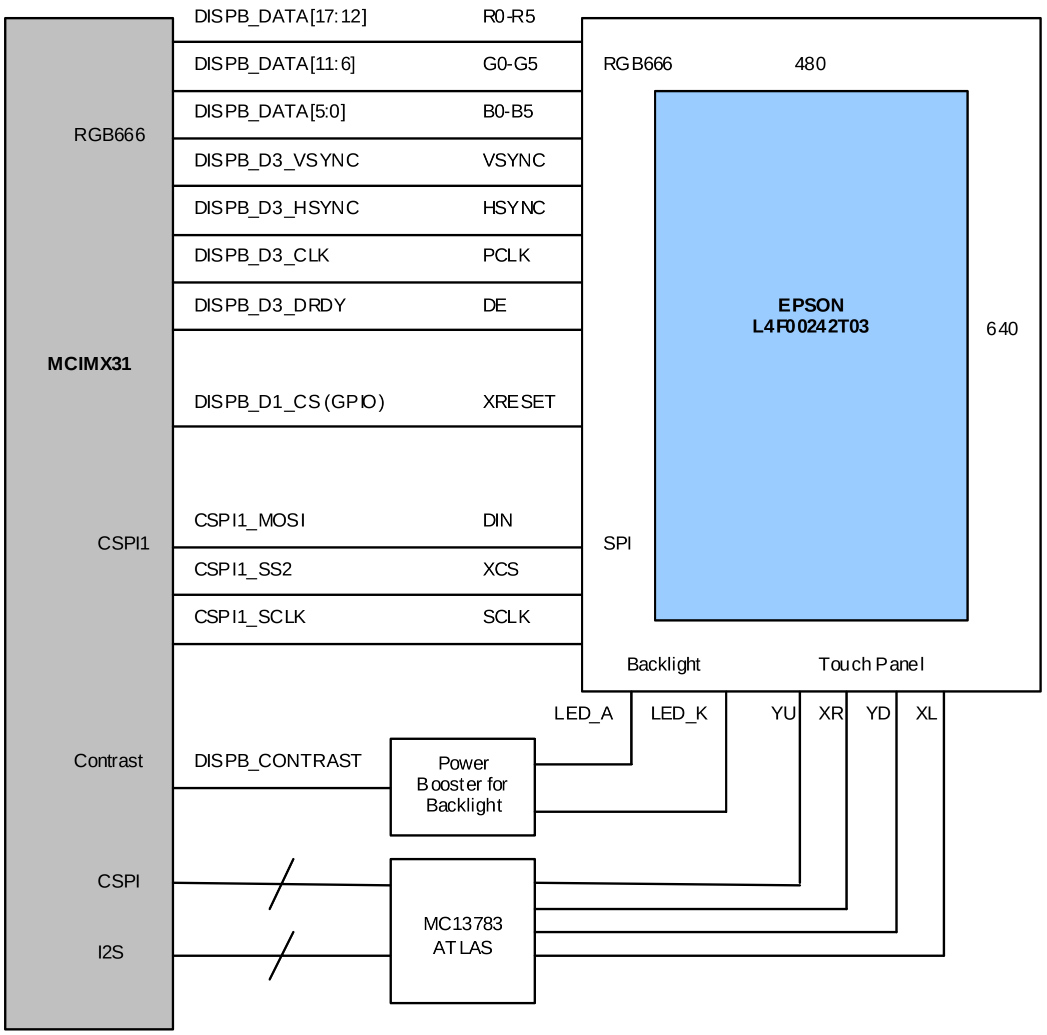


Figure 4. Interface between i.MX31 and Epson L4F00242T03 VGA Panel

The LCD panel is shipped with the i.MX31 PDK, and the panel requires the HSYNC, VSYNC, DE, and PIXCLK signals and complete RGB data interface (DISPB_DATA[17:0]). Also, a reset signal and serial interface (SPI) are required for sending the initialization routine commands. The backlight unit is controlled by using a Pulse Width Modulated (PWM) signal (contrast signal) that is generated by the i.MX31 processor, and the touch panel interface is handled by the MC13783 Atlas™ chip.

Every panel has its own interface and requirements, but this example illustrates a typical synchronous panel interface. Therefore, consider the possibility shown in Figure 4 as the base for the panel interface. The idea of base interface is useful when there are many panels that do not use the complete interface. For example, some of the panels do not require the HSYNC, VSYNC, reset, or serial initialization routine to handle the display signals. These panels use only the DRDY, PIXCLK, and RGB data and expect the microprocessor to accomplish with the waveforms as there is no way for the LCD panel to handle a different interface.

The only disadvantage with this interface is that the panel does not provide a backlight power booster and requires an external chip (MAX8595ZETA+T) to amplify the contrast signal.

### 3.1.2.2 i.MX31 PDK Chunghwa CLAA070VC01 7" WVGA LCD Interface

Figure 4 shows the interface between the i.MX31 and Chunghwa CLAA070VC01 WVGA panel.

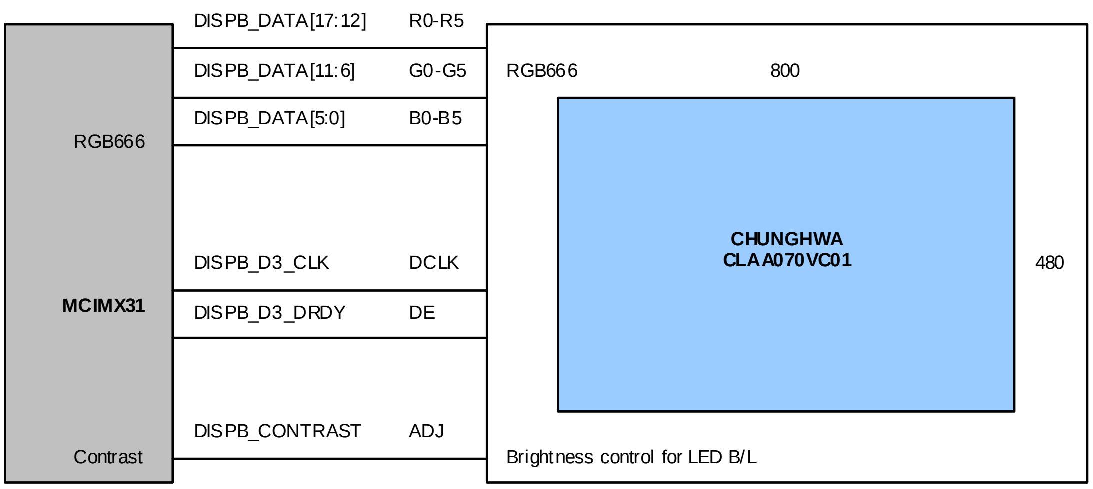


**Figure 5. Interface between i.MX31 and Chunghwa CLAA070VC01 VWGA Panel**

Figure 5 shows a simple display interface where the HSYNC and VSYNC signals are not used. Therefore, the DISPB_D3_VSYNC and DISPB_D3_HSYNC pins can be used for other purposes. Also, as the SPI interface is not required, it can be used as a chip select (CSPI1_SS2) for the other devices. Additionally, the power booster for the backlight unit is included in the module, which means that the contrast signal is directly connected to the display connector.

Apart from these advantages, the WVGA panel has the following disadvantages:

- The display module does not include a touch panel. Therefore, an external LCD panel should be added to the display module.
- The LCD module does not have a reset signal or SPI interface. Therefore, the display cannot be turned OFF. This feature is important particularly for mobile devices where power consumption is an issue. Therefore, external circuits are required to control the energy consumption of the LCD.

Based on these observations, the complete LCD circuit with the external panel and power control circuitry is shown in Figure 6.

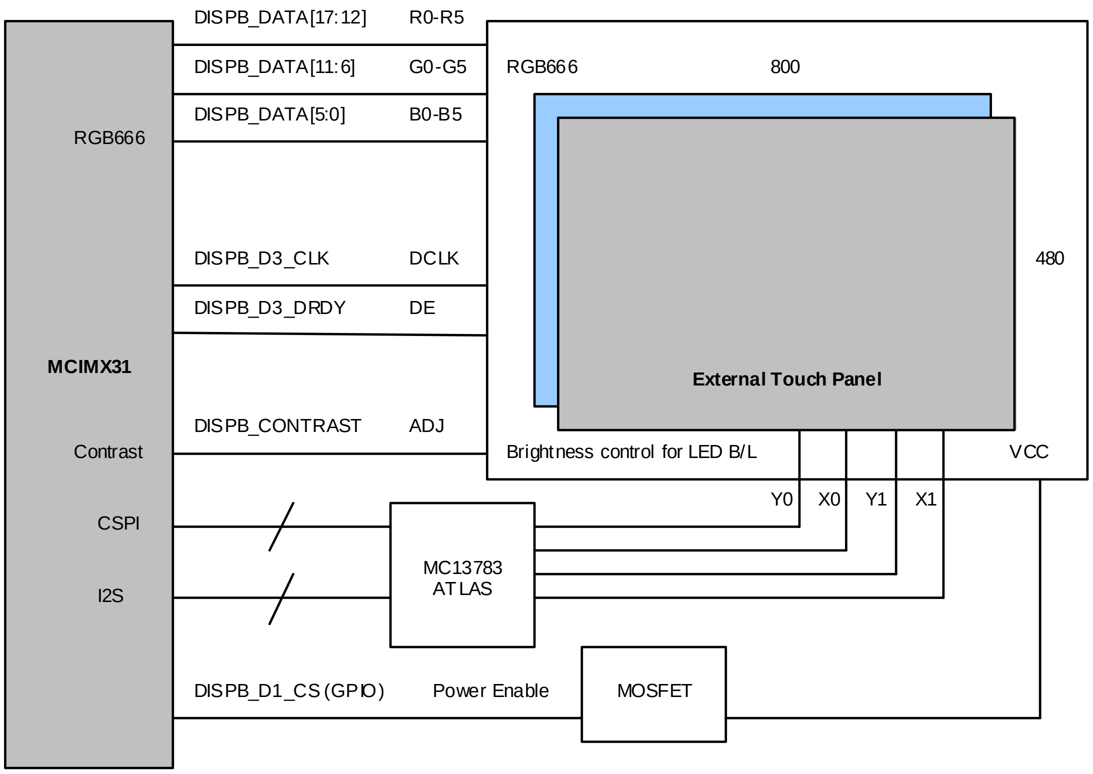


**Figure 6. Interface between i.MX31 and Chunghwa CLAA070VC01V WVGA Panel with Touch Panel and Power Control Circuitry**

**NOTE**

These examples are helpful for selecting an LCD panel.

## 3.2 Synchronous Display Timing and Signals

This section focuses on the timing and signal waveforms and describes how to configure them in the LCD panel and i.MX31 display interface. The first step to select an LCD module is to refer to the corresponding LCD module data sheet. The data sheet shows the pin interfaces, initialization routines, and timing charts for the RGB and serial interfaces if required. An abridged version of the data sheet is also available where all this information is not included. It is advisable to use the fully documented data sheet. Though there is not much difference between the preliminary and final versions, it is always better to use the final version.

### 3.2.1 Timing Concepts

Some important concepts and considerations, which form the base for the LCD interface timing, are given in Table 3.

Table 3. Timing Concepts

| Parameter | Description |
|---|---|
| Horizontal Back Porch (HBP) | Number of PIXCLK cycles between the HSYNC signal and first valid pixel data |
| Horizontal Front Porch (HFP) | Number of PIXCLK cycles between the last valid pixel data in the line and the next HSYNC pulse |
| Vertical Back Porch (VBP) | Number of lines (HSYNC pulses) between the asserted VSYNC signal and the next valid line |
| Vertical Front Porch (VFP) | Number of lines (HSYNC pulses) between the last valid line of the frame and the next VSYNC pulse |
| VSYNC pulse width | Number of lines (HSYNC pulses) between the last valid line of the frame and the next VSYNC pulse |
| HSYNC pulse width | Number of PIXCLK pulses when the HSYNC signal is active |
| Active frame width | Is the horizontal resolution, which is the number of pixels in a line. For a WVGA display of resolution, 800H $\times$ 480V, the frame width is equal to 800 pixels. |
| Active frame height | Is equal to the vertical resolution of the LCD. For a WVGA display of resolution, 800H $\times$ 480V, the value of the frame height is 480 lines. |
| Screen width | For the i.MX31 processor, screen width is the number of pixel clock periods between the last HSYNC and new HSYNC. This value not only includes the valid pixels but also the HBP and HFP. Note that this value is not equal to the horizontal resolution of the LCD panel (numbers of pixels in one line). Equation 1 gives the formula to calculate the screen width. $SCREEN\_WIDTH = ACTIVE\_FRAME\_WIDTH + HBP + HFP$ **Eqn. 1** |
| Screen height | For the i.MX31 processor, screen height is the number of rows between the last VSYNC pulse and new VSYNC pulse. This value includes all the valid lines, VBP, and VFP. Equation 2 gives the formula to calculate the screen height. $SCREEN\_HEIGHT = ACTIVE\_FRAME\_HEIGHT + VBP + VFP$ **Eqn. 2** |
| VSYNC polarity | Is the value held by the VSYNC signal to indicate the beginning of a new frame. The VSYNC polarity can be active low or active high. |
| HSYNC polarity | Is the value held by the HSYNC signal to indicate the beginning of a new line. The HSYNC polarity can be active low or active high. |

### 3.2.2 Timing Charts

The following charts are reviewed to clarify the timing issues in an LCD interface:

- Vertical timing chart
- Horizontal timing chart
- Pixel clock chart

Additionally, if the display uses a serial interface, refer to another chart that describes the serial interface and reset. This information should be extracted from the data sheet when a support for a new LCD panel is added. This is described with the following example. Consider a VGA (480H × 680V) LCD panel that uses the EPSON L4F00242T03 panel interface, which is shown in Figure 4. This means that the display uses the RGB666, VSYNC, HSYNC, DE, pixel clock, and also requires a reset signal and serial interface.

### 3.2.2.1 Vertical Timing Charts

This section describes the VGA and WVGA vertical timing charts.

#### VGA Vertical Timing Chart

Figure 7 shows the vertical timing chart for a hypothetic synchronous VGA display (480H × 680V).

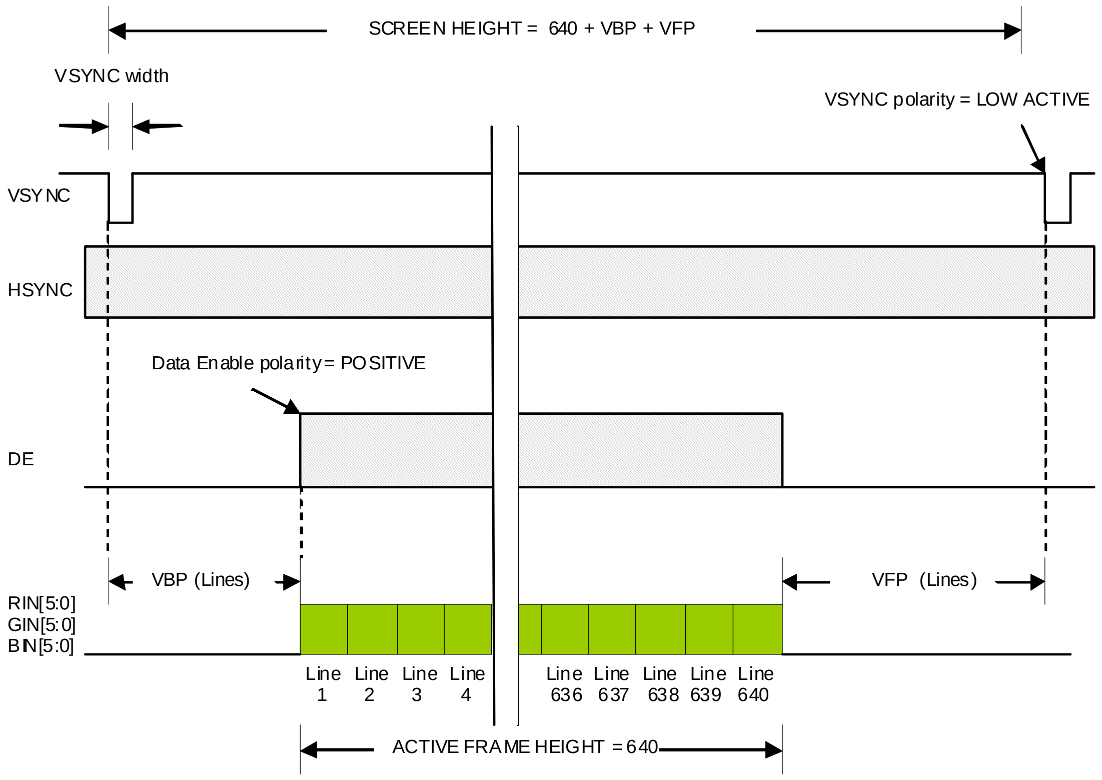


**Figure 7. VGA Vertical Timing Chart**

The vertical timing chart is analyzed in the following three stages:

- The beginning of a frame is indicated by the VSYNC signal (in this case, when the VSYNC signal goes low). The VSYNC period involves a complete frame cycle. During this period, all the pixels and lines in the frame are sent to the panel. When the VSYNC signal goes low (as the VSYNC signal is active low), the HSYNC signal immediately marks the beginning of the first line (in this case, the HSYNC signal goes low). However, to maintain the LCD timing, the first few lines are designated for the VBP. During the VBP, the DE signal is not present and the panel ignores the pixel data in the bus in all these lines.

- After the VBP, the DE signal appears in the boundaries of the HSYNC period (refer to Section 3.2.2.2, "Horizontal Timing Charts," for more information regarding the DE signal during the line cycle). Consequently, the DE signal appears during all the valid lines (with a vertical resolution of 640). During this time (ACTIVE FRAME HEIGHT), the LCD panel latches the RGB data on all the lines and draws them to the screen.
- During the VFP, extra lines (HSYNC cycles) appears. In this stage, the DE signal remains inactive and the panel discards any information in the RGB bus. The frame ends when the next VSYNC signal is asserted (goes low).

Table 4 gives the range of the timing features that is similar to the one provided the data sheet.

**Table 4. VGA Vertical Timing**

| Parameter | Symbol | Minimum | Typical | Maximum | Unit |
|---|---|---|---|---|---|
| Screen Height or Vertical Cycle | VP | — | 665 | — | Line |
| VSYNC Pulse Width | VSW | 5 | 10 | 15 | Line |
| Vertical Back Porch | VBP | 5 | 10 | 15 | Line |
| Vertical Front Porch | VFP | 5 | 5 | 5 | Line |
| Active Frame Height | VDISP | — | 640 | — | Line |
| Vertical Refresh Rate | FV | — | 60 | — | Hertz |

From Figure 7, the following features are verified:

- The VSYNC polarity is active low, which means that the vertical synchronization is normally in the high state and goes low to indicate the beginning of a new frame.
- The VSYNC pulse width (VSW) has certain flexibility. Therefore, the timing can be set using more than one value. It is recommended to use the typical values or values close to them. In this example, the typical value is taken as 10 lines.
- The VBP and VFP values are measured in lines as in HSYNC pulses. In this example, VBP is 10 lines and VFP is 5 lines width.
- The VSYNC width is included in the VBP stage, which means that the VBP starts when the VSYNC signal is asserted and not when the VSYNC signal returns to normal state.

Using these values, the screen height or vertical cycle (VP) is calculated as 665 lines. In some cases, the value of the VBP and VFP are not given in lines, but in nanoseconds (ns) or milliseconds (ms). In this case, additional calculations should be performed to find the number of lines required to accomplish the corresponding timings.

## WVGA Vertical Timing Chart

If an LCD panel, such as the hypothetical WVGA (800H × 480V), is used as described in Figure 5 and Figure 6, which does not use the HSYNC and VSYNC signals, the waveforms are analyzed in another perspective as shown in Figure 8.

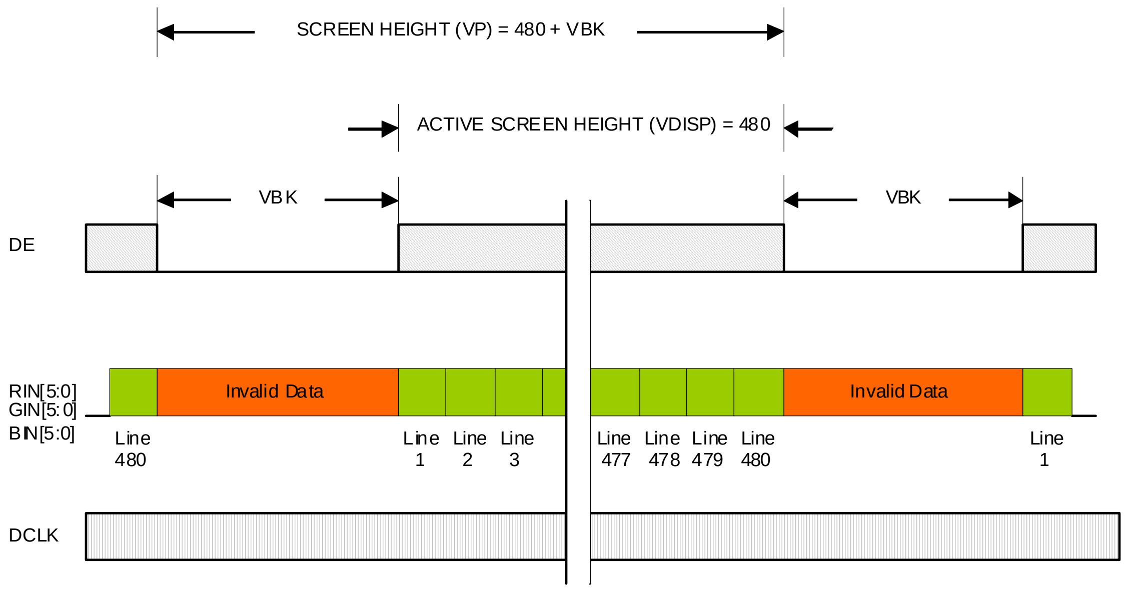


**Figure 8. WVGA Vertical Timing Chart**

Table 5 gives the range of the timing features used in Figure 8.

**Table 5. WVGA Vertical Timing**

| Parameter | Symbol | Minimum | Typical | Maximum | Unit |
|---|---|---|---|---|---|
| Screen Height or Vertical Cycle | VP | 490 | 500 | 520 | Line |
| Vertical Blank | VBK | 10 | 20 | 40 | Line |
| Active Screen Height | VDISP | 480 | 480 | 480 | Line |
| Vertical Refresh Rate | FV | 55 | 60 | 65 | Hz |

In this cases, the VSYNC width, VSYNC polarity, VBP, and VFP are not exposed in the chart. Even if the VSYNC signal is not used, these values are required to configure the i.MX31 display interface. These waveforms are used to understand the vertical cycle behavior. For the i.MX31 processor, the sequence remains the same. That is, the vertical cycle starts with the VSYNC signal, then comes the VBP, followed by the active frame area and the VFP appears until the VSYNC signal is asserted again. The tip to find the VSYNC width, VBP, and VFP is based on the fact that these events happen during the vertical blank period.

Figure 9 shows the WVGA vertical timing chart with the imaginary VSYNC signal.

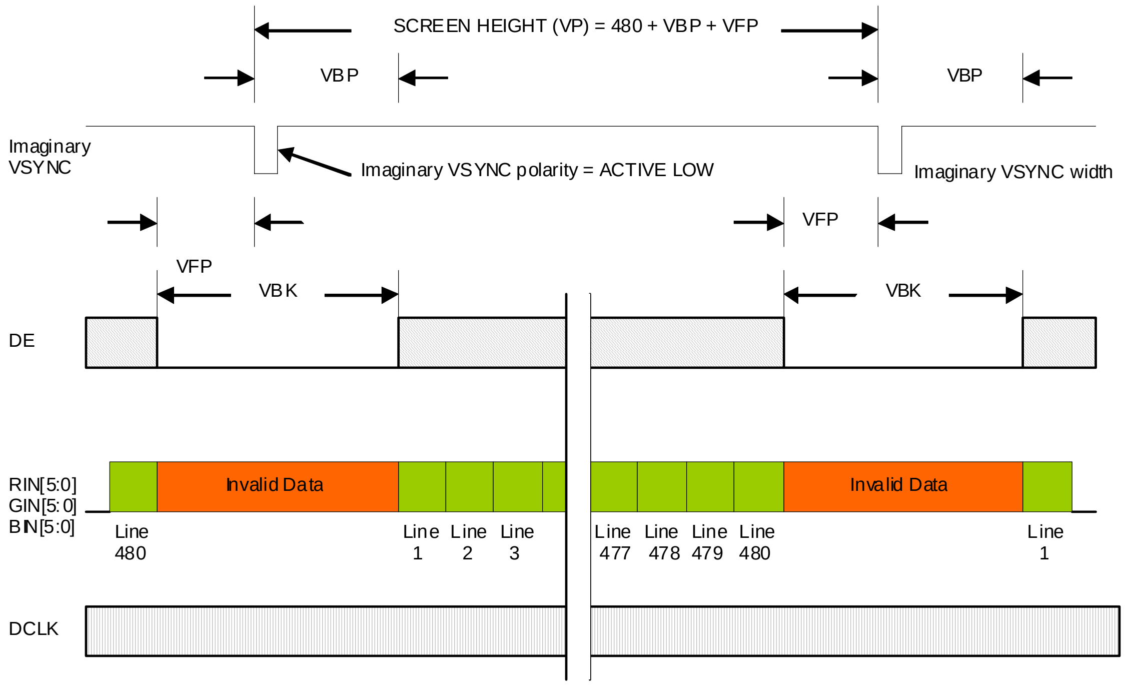


**Figure 9. WVGA Vertical Timing Chart with Imaginary VSYNC Signal**

The VSYNC signal may not be mapped to any pin. This signal is used only as the base to calculate the VFP and VBP. As the VSYNC signal is not used, the signal can be set as active low or active high. However, it is recommend to set the VSYNC signal as active low. Though the VSYNC width is not a strict criterion, in these cases, the VSYNC width is taken as one line length (VSW = 1).

To find the VBP and VFP, the Vertical Blank (VBK) period is split into two parts—the first part is for the VFP before the VSYNC signal is asserted and the other part is for the VBP. The sum of these two values should fit in the VBK period. Though the VBK period can be split in any ratio, it is suggested to leave the imaginary VSYNC in the middle of the blank period, which means that the VBP and VFP values should be equal or almost equal.

Using this example, and considering that VBK is 20 lines (typical), the VBP and VFP values should be equal to 10 lines. Based on this information, the vertical timings are given in Table 6.

**Table 6. WVGA Vertical Timing and Porches**

| Parameter | Symbol | Minimum | Typical | Maximum | Unit |
|---|---|---|---|---|---|
| Screen Height or Vertical Cycle | VP | 490 | 500 | 520 | Line |
| VSYNC Pulse Width | VSW | 1 | 1 | 1 | Line |
| Vertical Back Porch | VBP | 1 | 10 | 40 | Line |
| Vertical Front Porch | VFP | 0 | 10 | 39 | Line |
| Vertical Blank | VBK | 10 | 20 | 40 | Line |

Table 6. WVGA Vertical Timing and Porches (continued)

| Parameter | Symbol | Minimum | Typical | Maximum | Unit |
|---|---|---|---|---|---|
| Active frame height | VDISP | 480 | 480 | 480 | Line |
| Vertical refresh rate | FV | 55 | 60 | 65 | Hz |

### 3.2.2.2 Horizontal Timing Charts

This section describes the VGA and WVGA horizontal timing charts.

#### VGA Horizontal Timing Chart

The data sheet also includes another chart that describes the line period and is shown in Figure 10.

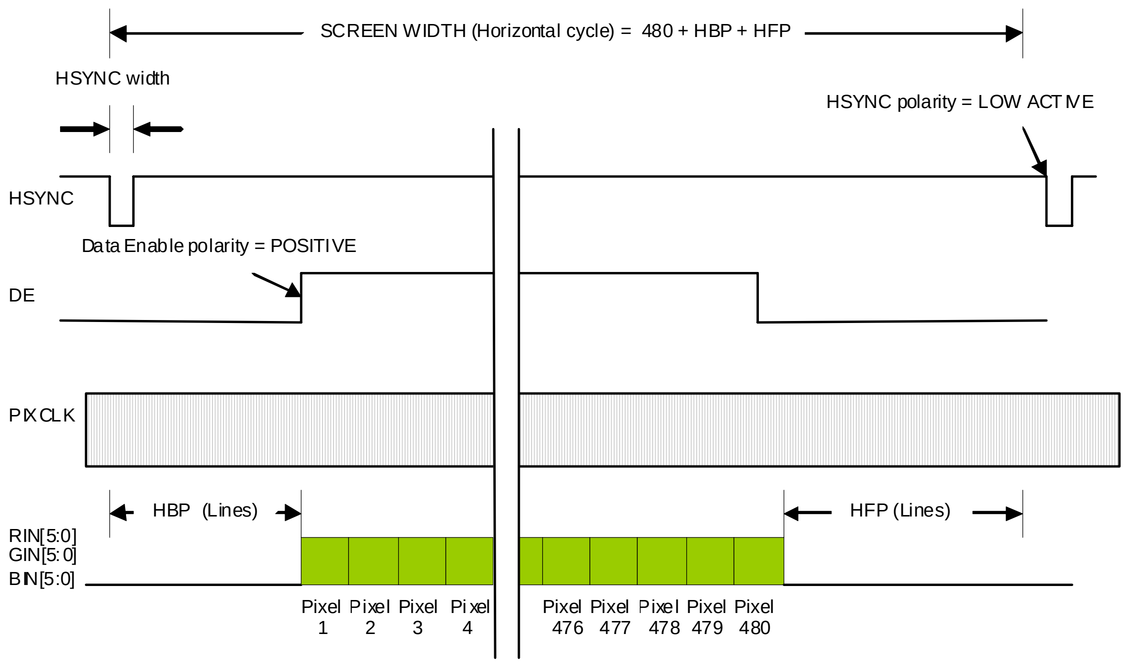


Figure 10. VGA Horizontal Timing Chart

This chart is analyzed in the following three stages:

- The line cycle begins when the HSYNC signal is asserted (in this case, when the signal goes low). Then, the HBP stage appears. During this time, the DE signal remains inactive.
- The horizontal active area (active frame width) begins when the DE (which is active high) signal is asserted. When the DE signal is active, the panel latches the RGB data placed in the bus and draws a new pixel on the screen for every PIXCLK pulse.
- The DE width is always equal to the horizontal resolution of the panel, (in this case, the DE width is 480 pixels length) and the active area ends when the DE signal is deasserted. After the active area, the HFP occurs and all the pixels in the line are drawn. The line cycle ends when the new HSYNC pulse is asserted.

The horizontal timings are given in Table 7.

**Table 7. VGA Horizontal Timing**

| Parameter | Symbol | Minimum | Typical | Maximum | Unit |
|---|---|---|---|---|---|
| Screen Width or Horizontal cycle | HP | 495 | 601 | 741 | PIXCLK |
| HSYNC pulse width | HSW | 5 | 20 | 75 | PIXCLK |
| Horizontal back Porch | HBP | 5 | 60 | 75 | PIXCLK |
| Horizontal front porch | HFP | 5 | 41 | 75 | PIXCLK |
| Active Frame Width | HDISP | — | 480 | 111 | PIXCLK |

## WVGA Horizontal Timing Chart

The WVGA horizontal timing chart and table in the data sheet are similar to the WVGA (800 H × 480 V) example shown in Figure 11 and Table 8, respectively.

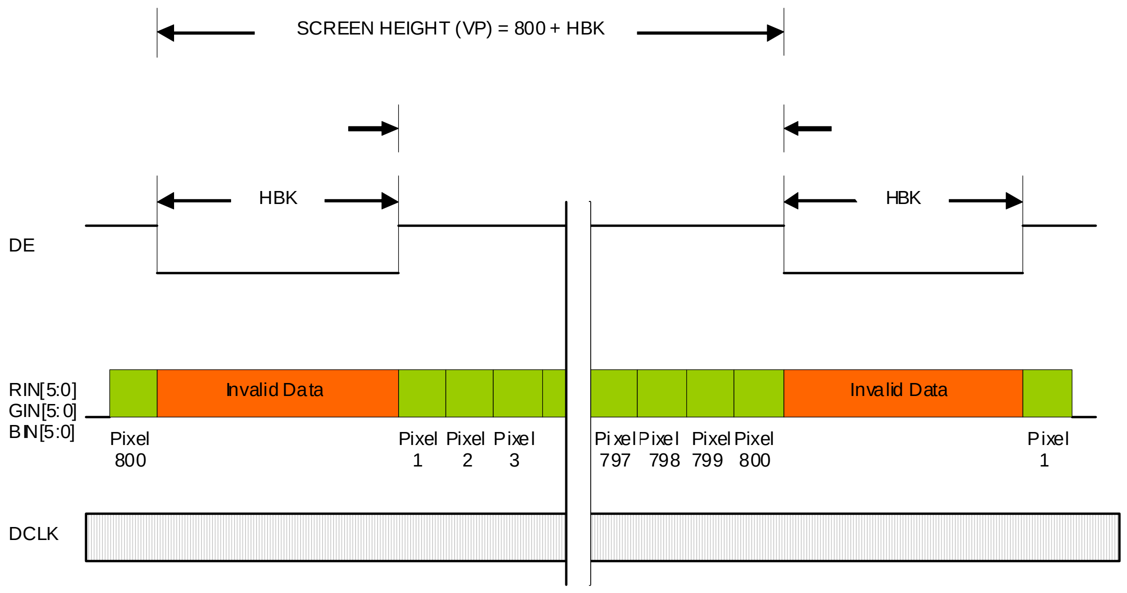


**Figure 11. WVGA Horizontal Timing Chart**

**Table 8. WVGA Horizontal Timing**

| Parameter | Symbol | Minimum | Typical | Maximum | Unit |
|---|---|---|---|---|---|
| Screen Width or Horizontal Cycle | HP | 850 | 900 | 950 | PIXCLK |
| Horizontal Blank Period | HBK | 50 | 100 | 150 | PIXCLK |
| Active Frame Width | HDISP | 800 | 800 | 800 | PIXCLK |

The values of the HBP, HFP, and HSYNC width are calculated using the same procedure used in the Section , “WVGA Vertical Timing Chart.”

The horizontal timing diagram with the imaginary HSYNC signal is shown in Figure 12.

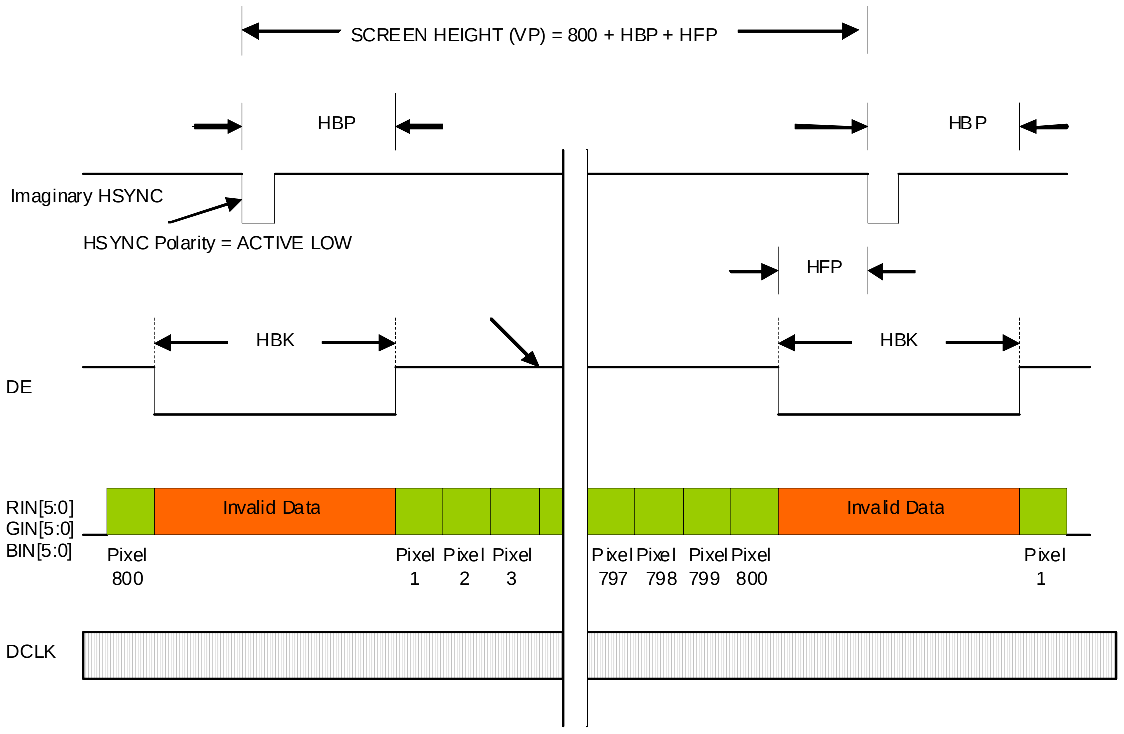


**Figure 12. WVGA Horizontal Timing Chart with Imaginary HSYNC Signal**

Table 9 gives the range of the timing features used in Figure 12.

**Table 9. WVGA Horizontal Timing and Porches**

| Parameter | Symbol | Minimum | Typical | Maximum | Unit |
|---|---|---|---|---|---|
| Screen width or Horizontal cycle | HP | 850 | 900 | 950 | PIXCLK |
| HSYNC pulse width | HSW | 1 | 1 | 1 | PIXCLK |
| Horizontal back porch | HBP | 1 | 50 | 150 | PIXCLK |
| Horizontal front porch | HFP | 0 | 50 | 149 | PIXCLK |
| Horizontal blank period | HBK | 50 | 100 | 150 | PIXCLK |
| Active frame width | HDISP | 800 | 800 | 800 | PIXCLK |

### 3.2.2.3 Pixel Clock Timing Charts

This section describes the VGA and WVGA pixel clock timing charts.

### VGA Pixel Clock Timing Chart

The waveform characteristics and table for the VGA pixel clock in the data sheet are similar to Figure 13 and Table 10, respectively.

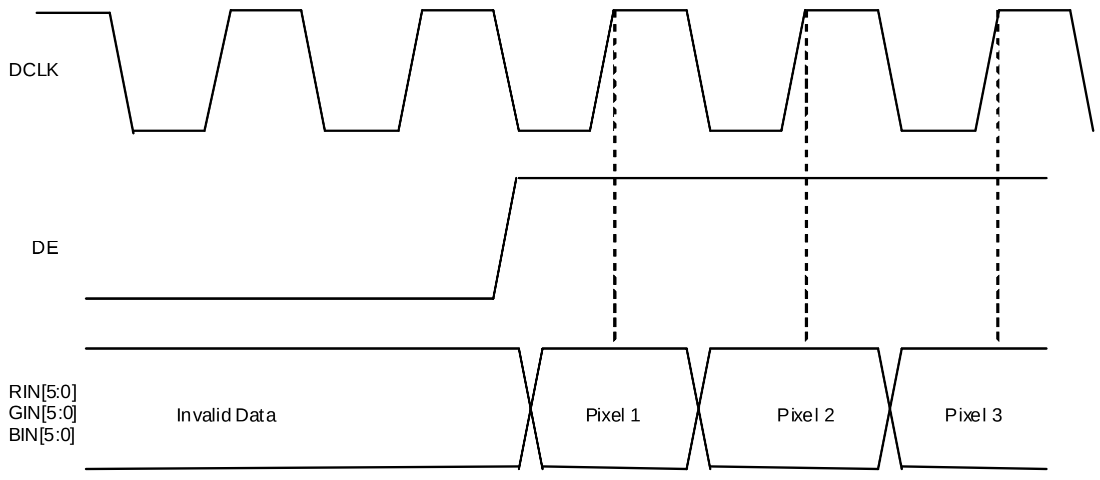


**Figure 13. VGA Pixel Clock Timing Chart**

**Table 10. VGA Pixel Clock Timing**

| Parameter | Symbol | Minimum | Typical | Maximum | Unit |
|---|---|---|---|---|---|
| Pixel clock frequency | PCLK | — | 24 | — | MHz |

An important feature regarding the pixel clock (DCLK) is to understand when the LCD panel latches the RGB data. This characteristic is important because the i.MX31 processor must prepare the data one edge before the LCD latches the data into the bus. In this case, the data is latched by the LCD panel in DCLK rising edges. Therefore, the i.MX31 processor should be configured to write the RGB data into the bus during the DCLK falling edge. In this manner, the data is made ready and stable for the panel to read the data.

The clock polarity is set in the DI_DISP_SIG_POL i.MX31 register, which is located under the D3_CLK_POL bit field. Figure 13 shows the typical inverse clock polarity. Also, the user should be aware that the maximal display clock rate cannot be greater than a quarter of the high speed processing clock rate. For example, the HSP_CLK signal in the i.MX31 PDK BSP is 133 MHz. Therefore, the maximum pixel clock is 133 MHz ÷ 4 = 33.25 MHz. However, most of the LCD displays work at a frequency lesser than the typical value.

### WVGA Pixel Clock Timing Chart

The waveform characteristics and table for the WVGA pixel clock in the data sheet is similar to Figure 14 and Table 11, respectively:

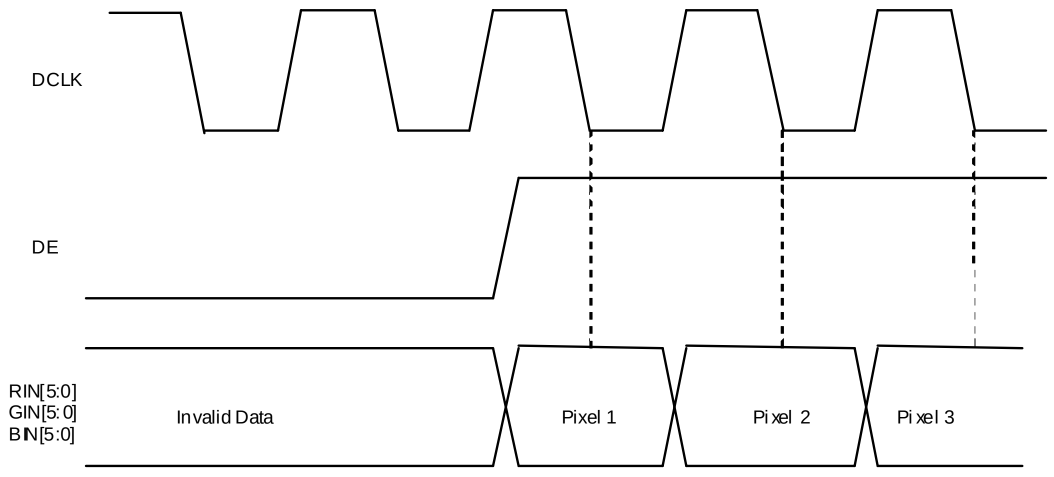


**Figure 14. WVGA Pixel Clock Timing Chart**

**Table 11. WVGA Pixel Clock Timing**

| Parameter | Symbol | Minimum | Typical | Maximum | Unit |
|---|---|---|---|---|---|
| Pixel clock frequency | PCLK | 25 | 27 | 32 | MHz |

In contrast to the VGA panel, the WVGA latches the RGB in the falling edges of the DCLK signal. Therefore, the i.MX31 processor should be configured to write the RGB data into the bus during the DCLK rising edge. In this manner, the data is made ready and stable for the panel to read the data. Note that the waveform characteristics in Figure 14 shows the straight clock polarity.

### Data Polarity

Data polarity is the value of the signals in the RGB bus that the LCD recognizes as active. This can be described with an example where the i.MX31 processor tries to draw a red pixel (only the red component) by using an RGB565 interface with the following data polarity:

- If the LCD uses straight polarity, the value in the bus becomes 0xF8000, which means that all the RGB bits are high and the rest of the bits are low.
- If the LCD uses inverse data polarity, the value in the bus becomes 0x07FF, which means that the red bits are low and all other bits are high.

Both the values, 0xF8000 and 0x07FF, represent the red color, and the difference in the values is caused by the data polarity in the LCD panel. This feature is configured using the D3_DATA_POL bit field in the DI_DISP_SIG_POL i.MX31 register.

### 3.2.3 Custom LCD Timing

This section describes the VGA and WVGA custom LCD timings.

#### 3.2.3.1 VGA Custom LCD Timing

This section describes the reset signal and serial command interface.

**Reset**

Many LCD panels include an LCD controller that requires an external system reset. If the LCD mentions the usage of this signal, then the user should find the timing regarding this pulse.

The reset signal and its timing are shown in Figure 15 and Table 12, respectively.

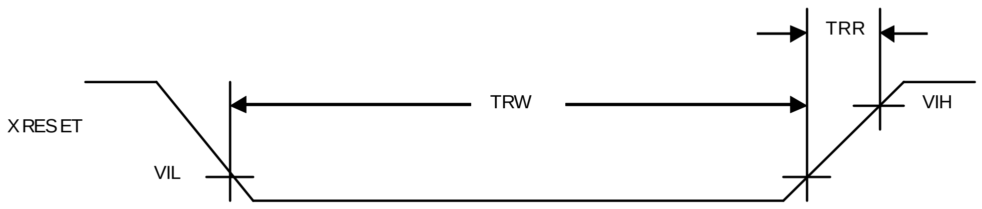


**Figure 15. Reset Signal**

**Table 12. Reset Signal Timings**

| Parameter | Symbol | Minimum | Typical | Maximum | Unit |
|---|---|---|---|---|---|
| Reset width | TRW | 15 | — | — | ns |
| Reset rising time | TRR | — | — | 10 | ns |

From Figure 15, it can be observed that the reset signal is active low. This means that reset signal is high during normal operations. The reset signal should be low for at least 15 ns for it to be considered as a valid reset. Also, the rise time of the reset signal is restricted to 10 ns. Generally, the reset pin is controlled by the General Purpose Input Output (GPIO) in the i.MX31 processor.

**NOTE**

It is recommended not to use the RC circuit to generate the reset signal as it restricts the rise time of the signal to 10 ns.

## Serial Command Interface

If the LCD panel has a serial command interface, a chart similar to Figure 16 is included in the data sheet.

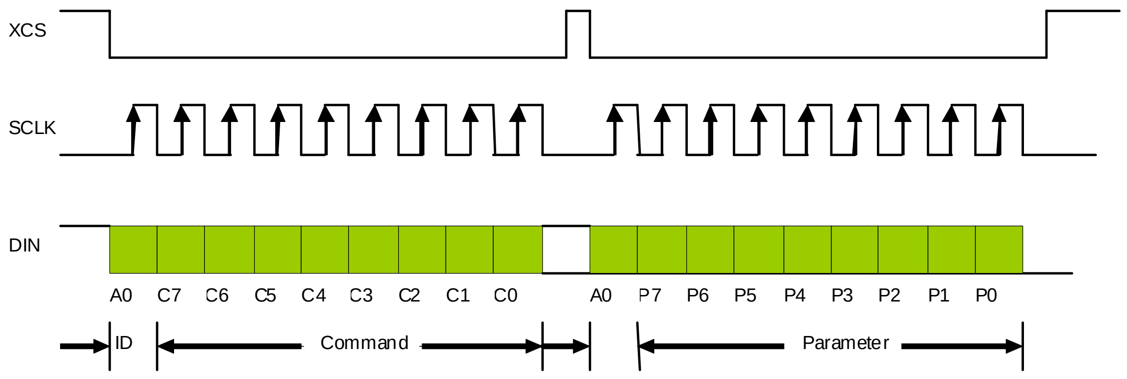


**Figure 16. SPI Command Interface Signals**

This application note does not review all the serial interfaces of an LCD. The protocols and data formats are described in the data sheet, and the user should have knowledge about the synchronous serial interfaces to program the serial interface settings. For more information, refer to the Configurable Serial Peripheral Interface (CSPI) chapter in the *MCIMX31 and MCIMX31L Applications Processors Reference Manual* (MCIX31RM)

### 3.2.3.2 WVGA Custom LCD Timing

There are no custom signals regarding the panel interface as the WVGA panel does not require any reset or initialization.

## 3.3 LCD Panels Supported by i.MX31

The i.MX31 processor can support up to four displays simultaneously. Table 13 lists the various types of displays that are handled by the display controllers.

**Table 13. Displays Supported by i.MX31**

| Display | Display Type | Interface |
|---|---|---|
| DISP0 | Asynchronous | Parallel interface only |
| DISP1 | Asynchronous | Serial and parallel interface |
| DISP2 | Asynchronous | Serial and parallel interface |
| DISP3 | Synchronous | RGB interface (HSYNC, VSYNC, PIXCLK, up to RGB666) |

Only one of the LCD display controllers in the i.MX31 processor is synchronous (dumb display). Therefore, the application note focuses on the DISP3 controller. The DISP3 RGB interface is multiplexed with all the other asynchronous parallel interfaces. This means that the data cannot be sent to the synchronous display (DISP3) and another parallel device at the same time. Instead, the i.MX31 processor sends data to the asynchronous panel (smart display) when the synchronous interface is inactive (during HFP, HBP, VFP, and VBP). This is the reason for the frame rate in smart displays gets affected when multiple displays are attached to the i.MX31 processor.

The synchronous LCD interface in the i.MX31 processor is flexible and handles various types of LCD devices with the following characteristics:

- Synchronous display (dumb display)
- RGB interface (RGB666 maximum)
- Resolution not greater than SVGA
- Utilizes at least the DE and pixel clock signals to latch the RGB data (some LCDs require the HSYNC and VSYNC signals, which are also supported by the i.MX31 processor)
- Pixel clock frequency lower than 33.25 MHz

Also, the i.MX31 processor can handle dump displays with the sharp interface. However, this support is limited to certain models only. For more information regarding the timing restrictions, refer to the Interface to Sharp HR-TFT Panels section in the *MCIMX31 and MCIMX31L Multimedia Applications Processors* (MCIMX31_5).

# 4 Display Configuration in Linux

The following sections describe how to add a new panel to the operating system. This includes the general display infrastructure in Linux, implementation of the display infrastructure in the i.MX family, and a few examples.

## 4.1 Linux Framebuffer Overview

This section describes the basic concepts and structures used in the Linux framebuffer.

### 4.1.1 Definition and Concept

Framebuffer concept is related to the graphics display video controller. Framebuffer can be viewed as a memory buffer for the video controller that contains a full frame of data. This frame of data becomes the information, which is shown on the display. The information provided to the frame consists of color values for each pixel.

In Linux, the framebuffer is implemented as a char device. An accurate definition for the Linux framebuffer implementation is provided by Geert Uytterhoeven in the kernel package document, —`../linux-2.6.xx/Documentation/fb/framebuffer`, and is as follows:
The framebuffer device provides an abstraction for the graphics driver. It represents the framebuffer of some video hardware and allows application software to access the graphic hardware through a well-defined interface. Therefore, the software is not required to know anything about the low-level interfaces.

The framebuffer infrastructure is easy to use and provides freedom for the user applications to access the video memory directly (by using the `mmap()` function). As the framebuffer is implemented as a char device, the user applications can interface with the device by using the common system calls such as `open()`, `read()`, `write()`, `ioctl()`, and so on (these functions are part of the file operation interface that every char device should have). The `mmap()` function maps files or devices into the program memory. In this case, the video buffer area is resource mapped. Therefore, with the usage of the `mmap()` function, the function gets access to the user space memory, which is equivalent to the hardware video framebuffer. As a result, the user gets a pointer to the framebuffer memory and the changes can be made directly in this memory that gets reflected on the display. A similar procedure can be done using the `write()` and `seek()` operations. However, the operations performed by these functions are time consuming as they are required to be called several times to perform or cover a determined area in the framebuffer. These operations include change in a particular section of the framebuffer, change in the display area, and so on.

### 4.1.2 Linux Framebuffer Structures

The framebuffer in Linux provides a set of structures that are employed for user-space applications. These are the most important elements that should be considered when a new panel driver is developed.

The most important data structures in the Linux framebuffer are described as follows:

**NOTE**

For more information and full definition of each structure, refer to the `.../include/linux/fb.h` file.

- `struct fb_fix_screeninfo`—contains fixed parameters for the graphics card and its controller. An example of a similar parameter is `unsigned long smem_start` (used at start of the framebuffer memory). This structure can be used in the user applications.
  Some of the other important parameters that are used in the structure are as follows:
  — `__u32 smem_len`—refers to the length of the framebuffer memory
  — `__u32 type`—refers to the pixel format
- `struct fb_var_screeninfo`—contains variables or alterable parameters for the graphics card/controller. These parameters refer to the features such as resolution and number of bits per pixel (`__u32 bits_per_pixel`) that are configured by the user. These parameters also contain structures that define the length and bit offset for each colors (for example, `struct fb_bitfield`). The `fb_var_screeninfo` structure can be used in the user applications.
  Some of the other important parameters that are used in the structure are as follows:
  — `__u32 xres`—visible resolution along the x axis
  — `__u32 yres`—visible resolution along the y axis
  — `__u32 xoffset`—offset from virtual to visible resolution along the x axis
- `struct fb_bitfield`—contains the detailed information of each color in a pixel (blue, green, and red). The fields of this structure include `__u32 offset` (refers to beginning of the bit field), `__u32 length` (refers to the length of the bit field), and the most significant bit flag. The `fb_var_screeninfo` structure contains one structure for each color (red, green, and blue).

  The `fb_bitfield` structure is as follows:

```
struct fb_bitfield {
         __u32 offset;/* beginning of bitfield*/
         __u32 length;/* length of bitfield*/
```

```
        __u32 msb_right; /* != 0 : Most significant bit is */
        /* right */
};
```

An example for the structure fields that contains the detailed information of each color in a pixel is as follows:

Consider a case in a configuration where the mode is RGB888 with a pixel width of 3 bytes. Then, the structure parameters takes the following values:

```
red.length= 8 ȝred.offset= 24
green.length= 8 ' green.offset = 16
blue.length= 8 ' blue.offset = 8
```

- `struct fb_ops`—contains function pointers to the framebuffer operations. These operations are from the basic or common driver functions such as the open and release function operations that are oriented to the parameter settings or `ioctl()` calls. Some of these functions are as follows:
  - `int (*fb_open)(struct fb_info *info, int user)`—opens the fb device and passes the `fb_info` structure pointer as the argument
  - `int (*fb_set_par)(struct fb_info *info)`—sets the video mode and other parameters according to the contents of the `var` (`fb_var_screeninfo`) element, which is located in the `fb_info` structure pointer. This structure pointer is passed as an argument in this function
  - `int (*fb_blank)(int blank, struct fb_info *info)`—operates to blank the display by using the `fb_info` structure pointer as an argument
  - `int (*fb_ioctl)(struct fb_info *info, unsigned int cmd,unsigned long arg)`—performs the input/output operations such as sending request for the structure values or configuring structure values by passing the `fb_info` structure pointer as the argument
  - `int (*fb_mmap(struct fb_info *info, struct vm_area_struct *vma)`—executes the mmap instructions by passing the `fb_info` structure and virtual memory area structure pointers as the arguments
- `struct fb_videomode`—is used when the user requires to add support for a new panel. Various information about the new panel such as the name, resolution, pixel clock, synchronization timings, and margins are referred in this structure. For more information about this structure, refer to Section 4.3.1.2, "Panel Configuration."

  The `fb_videomode` structure is as follows:

```
struct fb_videomode {
const char *name;/* optional */
u32 refresh;/* optional */
u32 xres;
u32 yres;
u32 pixclock;
u32 left_margin;
u32 right_margin;
u32 upper_margin;
u32 lower_margin;
u32 hsync_len;
u32 vsync_len;
u32 sync;
u32 vmode;
u32 flag;
};
```

- `struct fb_info`—is the most important structure in the framebuffer framework. The `fb_fix_screeninfo`, `fb_var_screeninfo`, `fb_bitfield`, `fb_ops`, and `fb_videomode` structures are declared in this structure. Many structures and elements such as pointers to devices, event queue, or monitor specifications are declared in this structure. The structure also contains structures that are enabled depending on the conditional building such as if the support for backlight is enabled.

  When a framebuffer driver gets registered to the kernel, the framebuffer driver uses a pointer to the structures that contain information (into several different structures) about the specific hardware panel, which is in use. This structure is visible only to the kernel and is as follows:

```
struct fb_info {
    …
    …
    struct fb_var_screeninfo var;/* Current var */
    struct fb_fix_screeninfo fix;/* Current fix */
    struct fb_monspecs monspecs;/* Current Monitor specs */
    struct work_struct queue;/* Framebuffer event queue */
    …
    struct fb_cmap cmap;                             /* Current cmap */
    …
struct fb_videomode *mode;/* current mode */
…
struct fb_ops *fbops;
    struct device *device;                      /* This is the parent */
    struct device *dev;                     /* This is this fb device */
…
};
```

Important points about the framebuffer framework are as follows:

- The framebuffer device is similar to the `/dev/mem` char device (or `/dev/fb*` char device) when the device is used in a user-space. In this case, the file operations performed in any char device (`open`, `read`, `write`, and `mmap`) can be performed in the framebuffer device.
- A good example for a framebuffer driver is the virtual framebuffer that is located in the `…/drivers/video/vfb.c` file. This implementation requires some actions that should be followed for the development of any framebuffer driver. These actions are as follows:
  - — Filling the fix and var structures in the `fb_info` structure that is targeted to the panel in use.
  - — Filling the file operation structures and the driver information for the `fb_info` structure. This action initializes the hardware and memory area and registers the framebuffer driver using a pointer to the `fb_info` structure.

## 4.2 Linux Framebuffer for i.MX

This section accounts the implementation side that describes the framebuffer main file (`…/drivers/video/fbmem.c`) and all the i.MX framebuffer implementation sources. These implementation sources provide information about the most important functions, structures, and a general flow chart for the initialization process.

The LCD driver and framebuffer implementation in the i.MX family uses the framebuffer framework for the hardware accessibility. Once the LCD driver is loaded (the LCD driver is selected in the kernel configuration screen for the graphics support while the kernel is configured), the hardware can be accessed using the special node, `fb*`, (which is like any other char device—refer to Section 4.1.2, “Linux Framebuffer Structures,”) that is located in the `/dev` directory. This directory resides in the downloadable Linux image.

The usage of the `/dev/fb*` node as an access mechanism allows some of the ioctl functions to interact, set, or get information from the device. Therefore, creation of a software abstraction layer helps the user to operate without the low-level knowledge.

The `ioctl` function performs the following operations:

- Requests information such as name, organization, addresses, and length
- Requests and changes the variable information about the hardware such as geometry, depth, color, and timing
- Obtains and sets the color map parts

The i.MX framebuffer implementation interacts with the generic framebuffer driver and is described in Section 4.2.1, “Initialization Process.” This section provides a general description of the initialization process with a descriptive flow chart that shows the different stages of the framebuffer implementation.

### 4.2.1 Initialization Process

This section describes the initialization flow for an LCD panel, which includes graphics element reviews for the framebuffer implementation. This section also describes files that are important for the framebuffer implementation.

The framebuffer initialization process contains several steps where specific functions for resource initialization and hardware probing or testing are called. Most of the files that take an important part in the framebuffer startup should be initialized and passed through the binding process. Therefore, functions such as `init()` and `probe()` are constantly described since these take an most part of the initialization startup.

The steps for the framebuffer initialization are as follows:

1. Kernel startup—the framebuffer initialization process, which is performed in the `mx3_3Stack.c` file, starts when the kernel calls the functions associated with the board configuration (in this case, with the i.MX31 3-Stack board). The `mxc_board_init()` function does the initialization of the most important systems on the board. Among these systems, the framebuffer is started by registering a platform device.
2. Framebuffer initialization—is performed in the `fbmem.c` file, and the framebuffer driver initialization is done in this phase. However, the framebuffer driver is not attached to a specific platform in this phase. The framebuffer driver starts the resources and important structures (most of the structures are described in Section 4.1.2, “Linux Framebuffer Structures,”) using the `fbmem_init` function (that registers the device).
3. IPU initialization—is performed in the `ipu_common.c` file. In this stage, the IPU modules are set up that includes the following:
   - — Registration of the IPU modules as a device in the system
   - — Initial configuration of the IPU modules and processes dedicated to attend the IPU tasks, such as the camera and LCD event handling

4. i.MX framebuffer initialization—is performed in the `mxcfb.c` file. In this phase, the next software layer, which is related to the framebuffer implementation in the i.MX family, is initialized and tested with all the components of that layer. This initialization process as well as most of the activities performed by the framebuffer calls to the IPU functions. The initialization process includes the registration of the framebuffer for the i.MX as a platform driver. However, most of the process is covered by the probe function where the framebuffer gets registered among the other resource settings. The framebuffer probe function executes the tests related to the IPU, such as initializing modes for the SDC module, executing tests for `IPU_IRQ`, enabling channels, disabling channels, and registering the framebuffer.

   For new LCD panels, a specific panel driver, which is similar to the `mxcfb.c` file, is made and is described as follows:
   Specific panel driver (`mxc_claa_wvga.c` or `mxc_epson_vga.c`)—The driver for the i.MX platform can be seen as a generic driver that works for several panels (see the `mxc_modedb.c` file). As the new panel has a different interface, a specific driver should be made for this type. In this case, the driver should be similar to the `mxcfb.c` file. However, the new driver should be more specific. Therefore, the usage of the initialization and probe functions are similar in this new driver. However, this driver does not replace the `mxcfb.c` file, but is complementary to this file. Therefore, to enable the panel for the proper functionality, both the drivers should be enabled.
5. Video4Linux (v4l) initialization—is performed in the `mxc_v4l2_output.c` file. At this point, all the elements to set up the framebuffer and display are set. However, to use the video, an extra step is required. The packages that contain the generic drivers follow the v4l2 standard. This driver can be used for signal capturing as well as for the output. The driver is loaded after setting up the framebuffer and IPU. The output driver makes use of the IPU post-processing functions for its usage. This driver also contains the initialization and probe routines.

Figure 17 shows the initialization flow chart for an LCD panel.

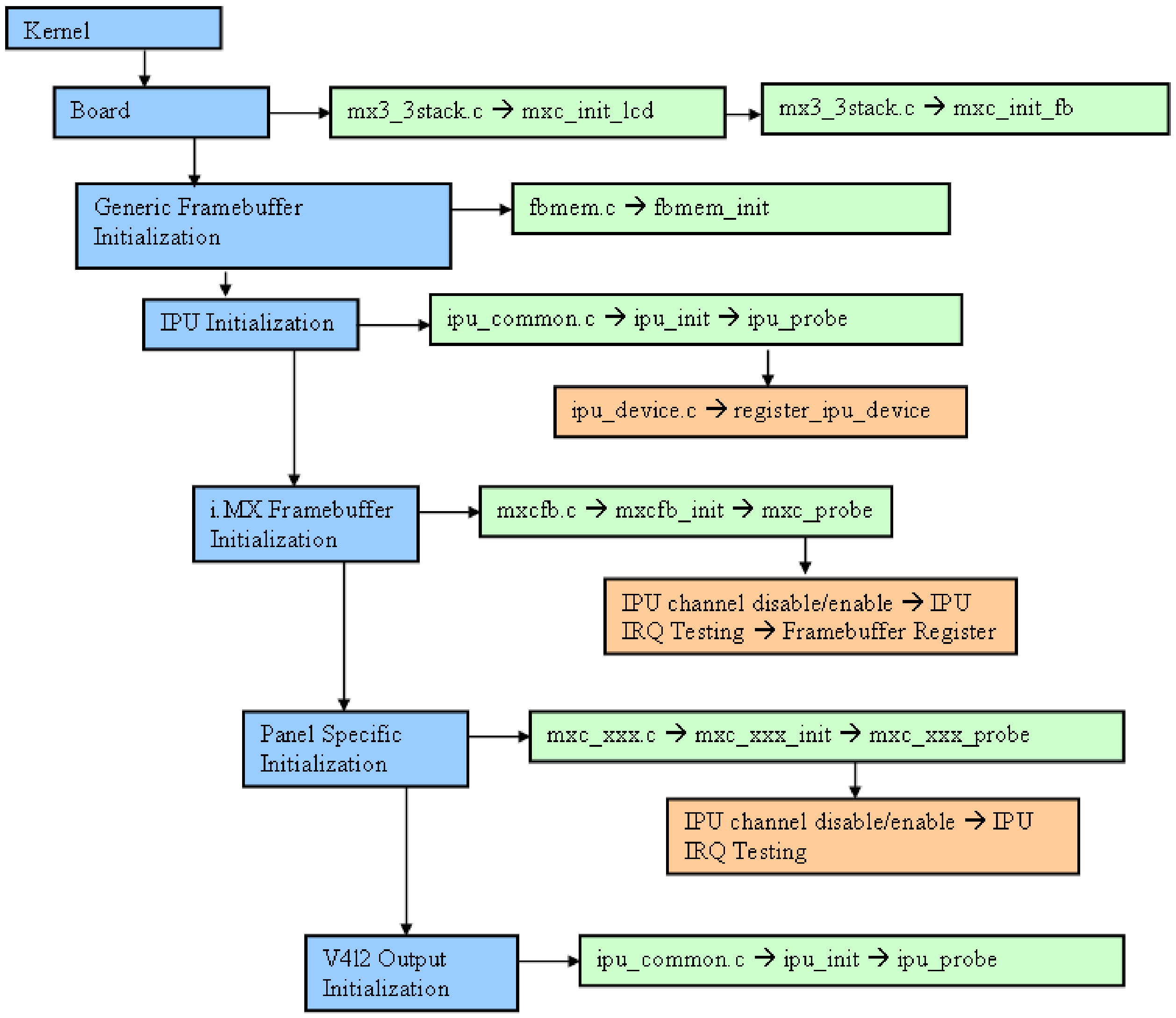

Kernel
Board
mx3_3stack.c → mxc_init_lcd
mx3_3stack.c → mxc_init_fb
Generic Framebuffer Initialization
fbmem.c → fbmem_init
IPU Initialization
ipu_common.c → ipu_init → ipu_probe
ipu_device.c → register_ipu_device
i.MX Framebuffer Initialization
mxcfb.c → mxcfb_init → mxc_probe
IPU channel disable/enable → IPU IRQ Testing → Framebuffer Register
Panel Specific Initialization
mxc_xxx.c → mxc_xxx_init → mxc_xxx_probe
IPU channel disable/enable → IPU IRQ Testing
V4l2 Output Initialization
ipu_common.c → ipu_init → ipu_probe

**Figure 17. Initialization Flow Chart**

### 4.2.2 Files and Important Elements

This section describes the files mentioned in Section 4.1.1, “Definition and Concept,” and Section 4.2.1, “Initialization Process,” and the files that provide important information about the framebuffer infrastructure. These files are as follows:

- `/arch/arm/mach-mx3/mx3_3stack.c`:

  This file contains the initialization and set up routines for the i.MX31 3-Stack board. These routines are called at the time of the kernel startup. The most important routine in this file is `mxc_board_init`. This routine calls several module specific routines, which includes the `mxc_init_fb` routine that registers a platform device for the framebuffer. Information such as name, DMA mask, and some platform data (a char variable of type, panel) are provided to this routine.

The `mxc_board_init` routine is set among the clock, GPIO modules, Power Management IC (PMIC), and so on. Also, there is a small LCD structure initialization in the `mxc_init_lcd` function. This registers the platform device structure for the LCD.

The `mx3_3stack.c` file is as follows:

```
#if defined(CONFIG_FB_MXC_SYNC_PANEL) || defined(CONFIG_FB_MXC_SYNC_PANEL_MODULE)
static const char fb_default_mode[] = "Epson-VGA";

/* mxc lcd driver */
static struct platform_device mxc_fb_device = {
.name = "mxc_sdc_fb",
.id = 0,
.dev = {
         .release = mxc_nop_release,
         .platform_data = &fb_default_mode,
         .coherent_dma_mask = 0xFFFFFFFF,
         },
};

static void mxc_init_fb(void)
{
(void)platform_device_register(&mxc_fb_device);
}


static struct platform_device lcd_dev = {
.name = "lcd_claa",
.id = 0,
.dev = {
         .release = mxc_nop_release,
         .platform_data = (void*)&lcd_data,
         },
};

static void mxc_init_lcd(void)
{
platform_device_register(&lcd_dev);
}
```

Figure 18 shows the flow chart of the `mx3_3stack.c` file.

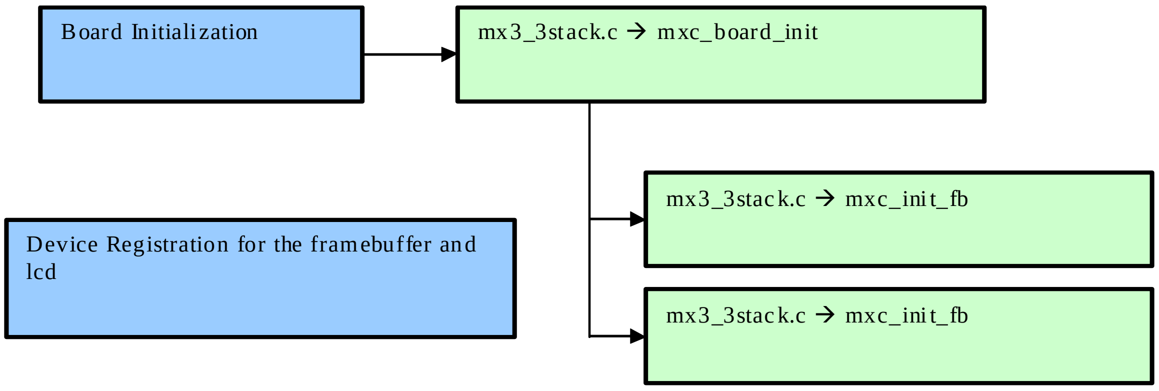

Board Initialization
mx3_3stack.c → mxc_board_init
mx3_3stack.c → mxc_init_fb
Device Registration for the framebuffer and lcd
mx3_3stack.c → mxc_init_fb

**Figure 18. mx3_3stack Flow Chart**

- /drivers/video/fbmem.c:
  This file includes the following:
  - Framebuffer subsystem initialization for the Linux
  - All the file operations (fops) that are common for a char device
  - mmap() memory map function that is widely used in this case (not always used in regular char devices)
  - Panel framebuffer driver routines for the registered and unregistered framebuffer devices
  - Functions related to the display of logos

  Important functions and structures in the fbmem.c file are described as follows:
  - Module definition and registering functions:
    - struct fb_info *registered_fb[FB_MAX] __read_mostly—is the global fb_info structure for the framebuffer system
    - int num_registered_fb __read_mostly—is an int variable and holds the quantity of framebuffers that are registered in the system
    - static int __init fbmem_init(void)—initializes the framebuffer subsystem as a char driver that is passed as a parameter into the fops table. These parameters are passed with the pointers to each fops function. This function also creates a class for the graphics.
    - static void __exit fbmem_exit(void)—terminates the char driver and deletes the graphics class
    - int register_framebuffer(struct fb_info *fb_info)—registers the framebuffer device by using the fb_info structure as the argument
    - int unregister_framebuffer(struct fb_info *fb_info)—unregisters the framebuffer device by using the fb_info structure as the argument
  - fops:
    - file_operations fb_fops—is a structure that contains the fops for the framebuffer subsystem, which is implemented as a char driver. The common read, write, open, and ioctl operations are declared in this structure. The mmap operation is also an important member in this structure.

      The file_operations fb_fops structure is as follows:

```
static const struct file_operations fb_fops = {
     .owner =                                             THIS_MODULE,
     .read =                                                  fb_read,
     .write =                                                fb_write,
     .ioctl =                                                fb_ioctl,

#ifdef CONFIG_COMPAT
     .compat_ioctl =                fb_compat_ioctl,
#endif

     .mmap =                                                  fb_mmap,
     .open =                                                  fb_open,
     .release =                                            fb_release,

#ifdef HAVE_ARCH_FB_UNMAPPED_AREA
     .get_unmapped_area = get_fb_unmapped_area,
#endif

#ifdef CONFIG_FB_DEFERRED_IO
     .fsync =                                   fb_deferred_io_fsync,
#endif
     };
```

– `static int fb_ioctl(struct inode *inode, struct file *file, unsigned int cmd,unsigned long arg)`—is a regular ioctl function for the char devices. In this function, the command to be executed is passed as an argument and acts depending on the selected case. The cases—`FBIOGET_VSCREENINFO` and `FBIOGET_FSCREENINFO`—are used to get the information about the variable or fixed values in the `fb_information` structure.
– `static int fb_mmap(struct file *file, struct vm_area_struct * vma)`—is a function that performs the mmap implementation for the framebuffer char driver

Figure 19 shows the flow chart of the `fbmem.c` file.

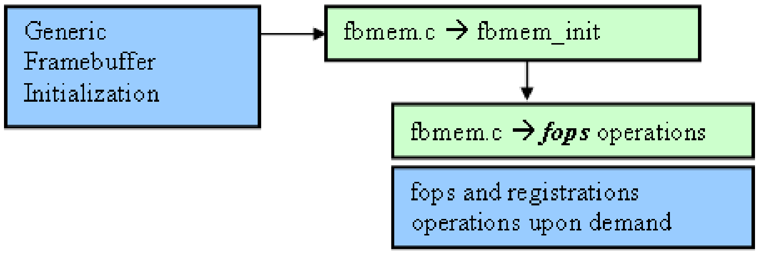

Generic Framebuffer Initialization
fbmem.c → fbmem_init
fbmem.c → fops operations
fops and registrations operations upon demand

**Figure 19.** fbmem Flow Chart

The IPU files are as follows:

- `drivers/mxc/ipu/ipu_common.c`:

  This file contains the common software routines—channel, buffer, and Interrupt Request (IRQ) management—that are required for the IPU functionality. The file also contains the `platform_driver` structure that is implemented for the IPU and the init and exit functions for the module.

  Important functions and structures in the `ipu_common.c` file are as follows:

  — Module definition and registering functions:

  – `platform_driver mxcipu_driver`—is a structure that contains the power management pointers to the routines that are used for testing the platform driver behavior in low-power modes. The `platform_driver mxcipu_driver` structure is as follows:

```
    static struct platform_driver mxcipu_driver = {
        .driver = {
                .name = "mxc_ipu",
                },
        .probe = ipu_probe,
        .suspend = ipu_suspend,
        .resume = ipu_resume,
        };
```

  – `int32_t __init ipu_gen_init(void)`—is the initialization routine for the IPU platform driver. This routine registers the `mxcipu_driver` structure.

    - `static void __exit ipu_gen_uninit(void)`—is the exit routine for the IPU platform driver. This routine frees the IPU IRQs and unregisters the `mxcipu_driver` structure.
    - `static int ipu_probe(struct platform_device *pdev)`—is the probe function for the `mxcipu_driver` structure and the function is called when the registration is performed. This function sets the IRQ request and clocks for the IPU and also registers the IPU device (see the `ipu_device.c` file for more information)
- Common IPU functions:
    - `ipu_request_irq`—registers an interrupt handler for the specified interrupt line. This interrupt line is defined in the `ipu_irq_line` enumeration that is located in the `ipu.h` file.
    - `ipu_disable_irq`—disables the interrupt for the specified interrupt line.
    - `ipu_enable_irq`—enables the interrupt for the specified interrupt line.
    - `ipu_init_channel_buffer`—initializes a buffer for the logical IPU channel. The function parameters include the physical addresses for the buffers, such as the buffer type, logical channel ID, width and height in pixels, and so on.
    - `ipu_select_buffer`—sets the channel buffers as ready. The parameters for this function are the logical channel ID and buffer type.
    - `ipu_init_channel`—initializes logical IPU channel. The function parameters include the logical channel ID and a union, along with the channel initialization parameters (the channel initialization parameter, `ipu_channel_params_t`, is included in the `ipu.h` file).
    - `ipu_uninit_channel`—uninitializes the logical IPU channel.
    - `ipu_link_channels`—links two channels together for automatic frame synchronization. This function has two parameters—the source logical channel ID and destination logical channel ID. The source channel output is linked to the destination channel input.
    - `ipu_unlink_channels`—unlinks the two channels that is set up for the automatic frame synchronization and also disables the automatic frame synchronization.
    - `ipu_enable_channel`—enables the logical channel corresponding to the channel ID, which is sent as the input parameter.
    - `ipu_disable_channel`—disables the logical channel corresponding to the channel ID, which is sent as the input parameter.

Figure 20 shows the flow chart for the `ipu_common.c` file.

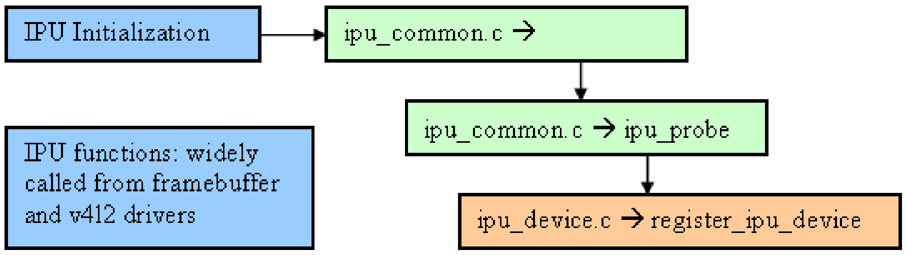

IPU Initialization
ipu_common.c →
ipu_common.c → ipu_probe
ipu_device.c → register_ipu_device
IPU functions: widely called from framebuffer and v412 drivers

**Figure 20. ipu_common Flow Chart**

- `drivers/mxc/ipu/ipu_device.c`:

This file contains structures and functions for the fops operations that are related to the `mxc_ipu` device. The file also contains a generic interrupt handler for the IPU related IRQs.

Important functions and structures in the `ipu_device.c` file are as follows:

— Registering and other important functions:

- `int register_ipu_device()`—registers `mxc_ipu` as a char device and provides the fops table and name. The function also creates a class in the device model structure. This function is called using the `probe()` function, which is located in the `ipu_common.c` file.
- `static irqreturn_t mxc_ipu_generic_handler(int irq, void *dev_id)`—is a generic handler for any IRQ that the IPU should process.

— fops:

- `file_operations mxc_ipu_fops`—is a structure that contains the fops for the `mxc_ipu` device. The `file_operations mxc_ipu_fops` structure is as follows:

```
static struct file_operations mxc_ipu_fops = {
     .owner = THIS_MODULE,
     .open = mxc_ipu_open,
     .release = mxc_ipu_release,
     .ioctl = mxc_ipu_ioctl
};
```

- `static int mxc_ipu_ioctl(struct inode *inode, struct file *file, unsigned int cmd, unsigned long arg)`—is the ioctl function for the `mxc_ipu` device. In this function, the commands are passed as arguments. Most of these commands—`IPU_INIT_CHANNEL`, `IPU_LINK_CHANNELS`, and so on—are described in the `ipu_common.c` file.

Figure 21 shows the `ipu_device.c` flow chart.

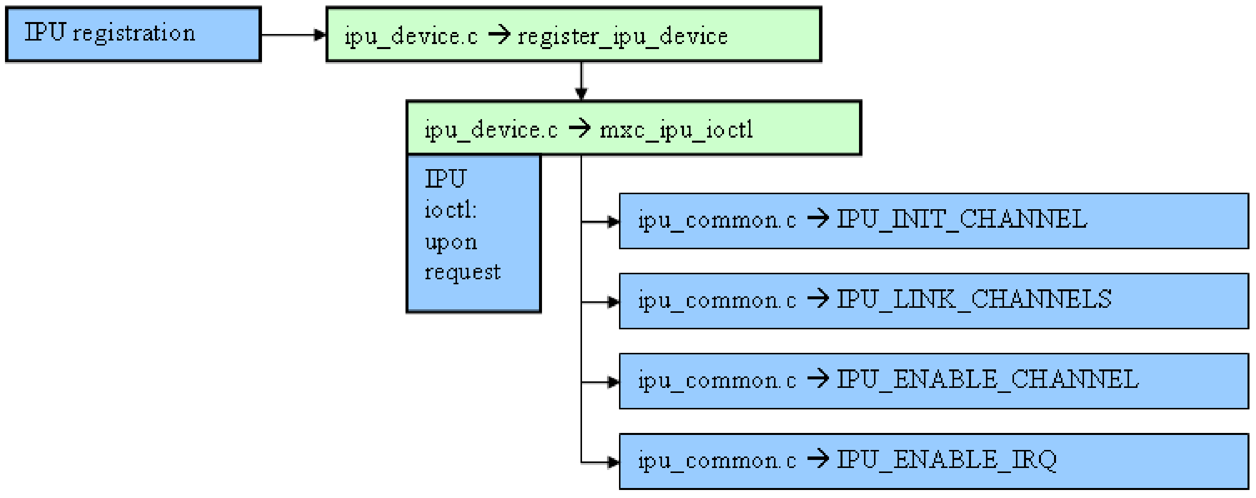

IPU registration
ipu_device.c → register_ipu_device
ipu_device.c → mxc_ipu_ioctl
IPU ioctl: upon request
ipu_common.c → IPU_INIT_CHANNEL
ipu_common.c → IPU_LINK_CHANNELS
ipu_common.c → IPU_ENABLE_CHANNEL
ipu_common.c → IPU_ENABLE_IRQ

**Figure 21. ipu_device Flow Chart**

- `drivers/mxc/ipu/ipu_sdc.c`:

  This file contains routines that are related to the SDC module that is located in the IPU module. The set of routines ranges from `sdc_init` to the routines that are used for setting the alpha blending modes or color keys in the SDC plane.

- `drivers/mxc/ipu/ipu_ic.c`:

This file contains routines that does the color conversion and resizes all the IPU submodules.

- `drivers/video/mxc/mxcfb_modedb.c`:

  This file contains an array of `fb_videomode` structure declarations that are related to the MXC framebuffer implementation. These structures include the Sharp-VGA, NEC-VGA, TV out modes, and so on. The data in each structure refers to the most important elements—resolution, size, timing delays, and so on—that describe the panel. More information about the structure parameters are available in the `include/linux/fb.h` file.

- `drivers/video/mxc/mxcfb.c`:

  This driver contains the registering and initialization routines for the framebuffer implementation that are oriented to the i.MX family. The `mxcfb.c` file initializes the following:

  - Most important structures that are related to the framebuffer
  - Functions for registering the driver and setting up the framebuffer system for the normal framebuffer data structures as well as for the overlay, if the overlay is enabled

  Important functions and structures in the `mxcfb.c` file are as follows:

  - Module definition and registering functions:
    - `platform_driver mxcfb_driver`—this structure is created for the framebuffer driver implementation. The structure consists of pointers to the functions that are related to power management callbacks, such as `probe` and `suspend`. The `platform_driver mxcfb_driver` structure is as follows:

```
static struct platform_driver mxcfb_driver = {
    .driver = {
           .name = MXCFB_NAME,
           },
    .probe = mxcfb_probe,
    .suspend = mxcfb_suspend,
    .resume = mxcfb_resume,
    };
```

    - `mxcfb_data`—encapsulates the two `fb_info` structures, in which one acts for the normal framebuffer and the other for the overlay. The `mxcfb_data` structure also includes flags. The `mxcfb_data` structure is as follows:

```
struct mxcfb_data {
        struct fb_info *fbi;
        struct fb_info *fbi_ovl;
        volatile int32_t vsync_flag;
        wait_queue_head_t vsync_wq;
        wait_queue_head_t suspend_wq;
        bool suspended;
        int backlight_level;
    };
```

    - `int __init mxcfb_init(void)`—is the entry function for the framebuffer. This function registers the platform driver structure that contains callback functions for the power management and shutdown conditions.
    - `void mxcfb_exit(void)`—is the exit function for the framebuffer. This function unmaps the video memory for the framebuffer structures, unregisters the `fb_info` structure from the framebuffer and overlay devices (these devices are unregistered), and unregisters the platform driver structure for the framebuffer.

   – `static int mxcfb_probe(struct platform_device *pdev)`—is a function member of the platform driver structure pointers. This probe function verifies if the specified device hardware exists and executes several processes such as framebuffer initialization, memory allocation, and framebuffer registration (`fb_info` structures for the normal and overlay structures). This function also performs the IPU initialization that involves transparent color key setting for the SDC graphic plane and foreground/background alpha blending modes.
   – `static struct fb_info *mxcfb_init_fbinfo(struct device *dev, struct fb_ops *ops)`—is a function that is called by the `mxc_probe()` function and performs functions that include the memory allocation for the `fb_info` structure and filling the fields related to this structure with information such as color maps, bits per pixel, and so on.
   – `platform_set_drvdata(pdev, &mxcfb_drv_data)`—is a function that passes address, which is obtained from the information in the `mxcfb_drv_data` structure, to the platform device.
— fops:
   – `fb_ops mxcfb_ops`—is a structure that contains pointers to the functions that are used by the framebuffer driver to perform functions such as rectangle filling, cursor definitions, and so on. This structure is used for normal framebuffer implementation and is as follows:

```
static struct fb_ops mxcfb_ops = {
     .owner = THIS_MODULE,
     .fb_set_par = mxcfb_set_par,
     .fb_check_var = mxcfb_check_var,
     .fb_setcolreg = mxcfb_setcolreg,
     .fb_pan_display = mxcfb_pan_display,
     .fb_ioctl = mxcfb_ioctl,
     .fb_fillrect = cfb_fillrect,
     .fb_copyarea = cfb_copyarea,
     .fb_imageblit = cfb_imageblit,
     .fb_blank = mxcfb_blank,
};
```

   – `fb_ops mxcfb_ovl_ops`—is a structure that contains pointers to the functions that are used by the framebuffer driver to perform functions such as rectangle filling, cursor definitions, and so on. This structure is used for overlay framebuffer implementation and is as follows:

```
static struct fb_ops mxcfb_ovl_ops = {
     .owner = THIS_MODULE,
     .fb_set_par = mxcfb_set_par,
     .fb_check_var = mxcfb_check_var,
     .fb_setcolreg = mxcfb_setcolreg,
     .fb_pan_display = mxcfb_pan_display,
     .fb_ioctl = mxcfb_ioctl_ovl,
     .fb_mmap = mxcfb_mmap,
     .fb_fillrect = cfb_fillrect,
     .fb_copyarea = cfb_copyarea,
     .fb_imageblit = cfb_imageblit,
     .fb_blank = mxcfb_blank_ovl,
};
```

   – `static int mxcfb_mmap(struct fb_info *fbi, struct vm_area_struct *vma)`—is a function that handles the `mmap` function for the MXC framebuffer.
   – `static int mxcfb_ioctl_ovl(struct fb_info *fbi, unsigned int cmd, unsigned long arg)`—is a function that handles the ioctl commands for the framebuffer and is used for the overlay framebuffer structure.

- `static int mxcfb_ioctl(struct fb_info *fbi, unsigned int cmd, unsigned long arg)`—is a function that handles the ioctl commands for the framebuffer and is used for the normal framebuffer structure.
- `static int mxcfb_set_par(struct fb_info *fbi)`—is a function that sets the parameter (most of the parameters are from the videomode structure) to the processor registers by calling the `ipu_sdc_init_panel()` function. This function also changes the operating mode.

Figure 22 shows the flow chart for the `mxcfb.c` file.

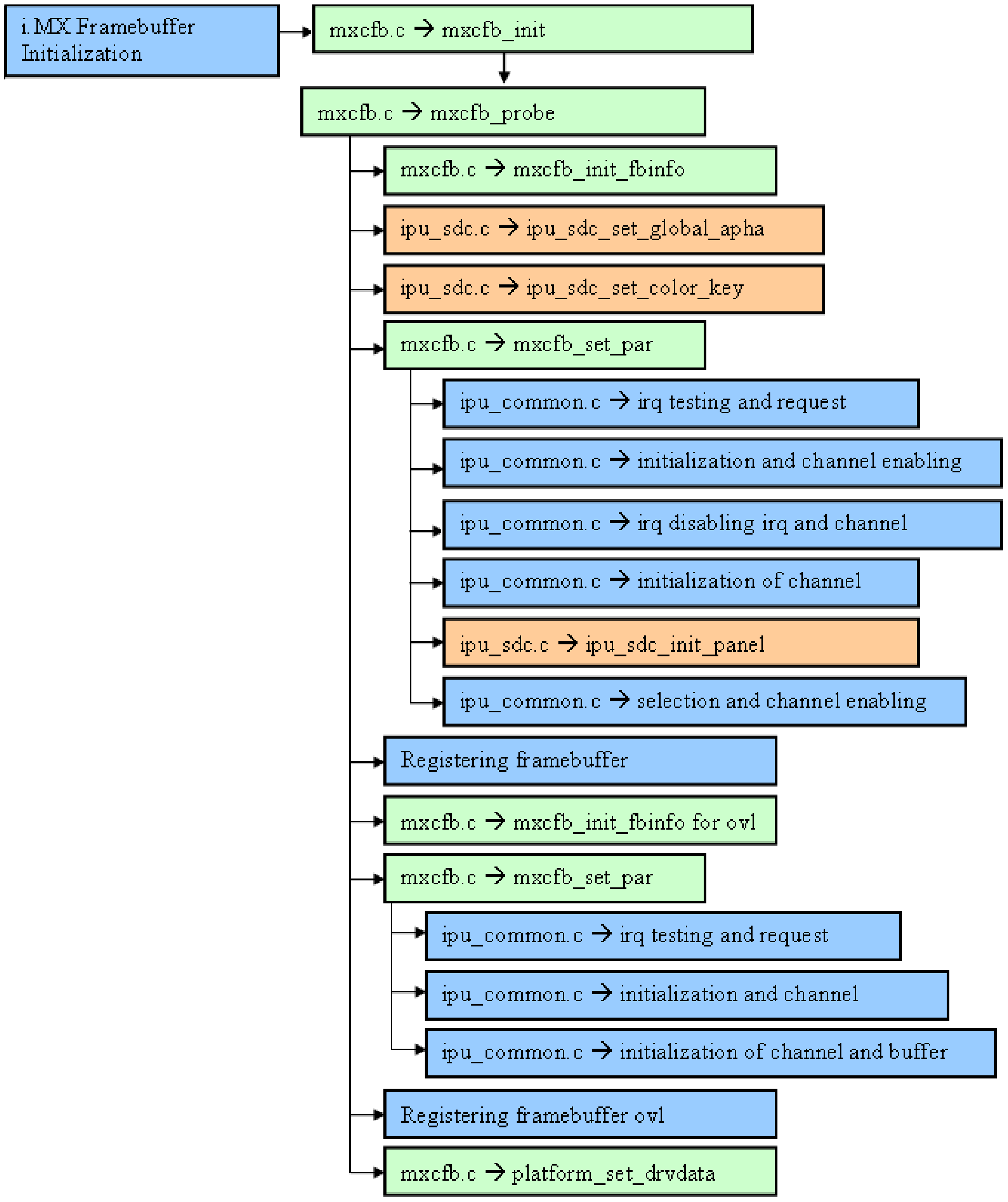

i.MX Framebuffer Initialization
mxcfb.c → mxcfb_init
mxcfb.c → mxcfb_probe
mxcfb.c → mxcfb_init_fbinfo
ipu_sdc.c → ipu_sdc_set_global_apha
ipu_sdc.c → ipu_sdc_set_color_key
mxcfb.c → mxcfb_set_par
ipu_common.c → irq testing and request
ipu_common.c → initialization and channel enabling
ipu_common.c → irq disabling irq and channel
ipu_common.c → initialization of channel
ipu_sdc.c → ipu_sdc_init_panel
ipu_common.c → selection and channel enabling
Registering framebuffer
mxcfb.c → mxcfb_init_fbinfo for ovl
mxcfb.c → mxcfb_set_par
ipu_common.c → irq testing and request
ipu_common.c → initialization and channel
ipu_common.c → initialization of channel and buffer
Registering framebuffer ovl
mxcfb.c → platform_set_drvdata

**Figure 22. mxcfb Flow Chart**

- `drivers/media/video/mxc/output/mxc_v4l2_output.c`:

  This file implements the v4l2 standard for the output devices that are targeted to the i.MX family. This file includes the following:

  — Common char driver infrastructure, which includes the initialization routine for the drivers that are registered

- — Probe function and standard fops that are related to the v4l2 implementation
- — Functions that enable or disable the playback of video

Important functions and structures in the `mxc_v4l2_output.c` file are as follows:

- — Module definition and registering functions:
  - – `platform_driver mxc_v4l2out_driver`—is used for the v4l2 output driver. This structure contains pointers to functions that are related to power management, such as `probe`, `remove`, and so on. The `platform_driver` structure is as follows:

    ```
    static struct platform_driver mxc_v4l2out_driver = {
         .driver = {
                 .name = "MXC Video Output",
                 },
         .probe = mxc_v4l2out_probe,
         .remove = mxc_v4l2out_remove,
    };
    ```

  - – `platform_device mxc_v4l2out_device`—is used for the v4l2 output driver. This structure contains the name and ID of the device. The `platform_device` structure is as follows:

    ```
    static struct platform_device mxc_v4l2out_device = {
         .name = "MXC Video Output",
         .id = 0,
    };
    ```

  - – `static int mxc_v4l2out_init(void)`—is the function that initializes the driver where the registration of the platform driver and platform device is done.
  - – `static void mxc_v4l2out_clean(void)`—is the exit function for the driver where the platform device, platform driver, and video device are unregistered.
  - – `static int mxc_v4l2out_probe(struct platform_device *pdev)`— is the probe function for the v4l2 driver that contains the setup for the outputs and cropping commands and video device register with `video_register_device`.
- — Fops operations:
  - – `file_operations mxc_v4l2out_fops`—this is the fops structure for the `mxc_v4l2_output.c` file that contains pointers to common functions, which includes `open()`, `close()`, `ioctl()`, and the `mmap` implementation for the driver. The `file_operations` structure is as follows:

    ```
    static struct file_operations mxc_v4l2out_fops = {
         .owner = THIS_MODULE,
         .open = mxc_v4l2out_open,
         .release = mxc_v4l2out_close,
         .ioctl = mxc_v4l2out_ioctl,
         .mmap = mxc_v4l2out_mmap,
         .poll = mxc_v4l2out_poll,
    };
    ```

  - – `static int mxc_v4l2out_mmap(struct file *file, struct vm_area_struct *vma)`—is the function that performs the `mmap` implementation for the v4l2 driver, which is a part of the fops.
  - – `static int mxc_v4l2out_do_ioctl(struct inode *inode, struct file *file, unsigned int ioctlnr, void *arg)`—is called by the `video_usercopy()` function (this function is called by the `mxc_v4l2out_ioctl` function). This function performs the commands sent by the application while performing some specific `ioctl()` calls.

— Other operations:

– `static int mxc_v4l2out_streamon(vout_data * vout)`—is the function that initiates the framebuffer or display playback and uses many calls and modifications directly to the IPU. These calls refer to the usage of the IPU channels, such as selecting buffer initializing channels, enabling IRQs, and so on.

– `static int mxc_v4l2out_streamoff(vout_data * vout)`—is the function that terminates the framebuffer or display playback. This function uses the functions implemented in the lower layer of the IPU configuration to un-initialize and unlink the channels and disable the IPU IRQs.

Figure 23 shows flow chart for the `mxc_v412_output.c` file.

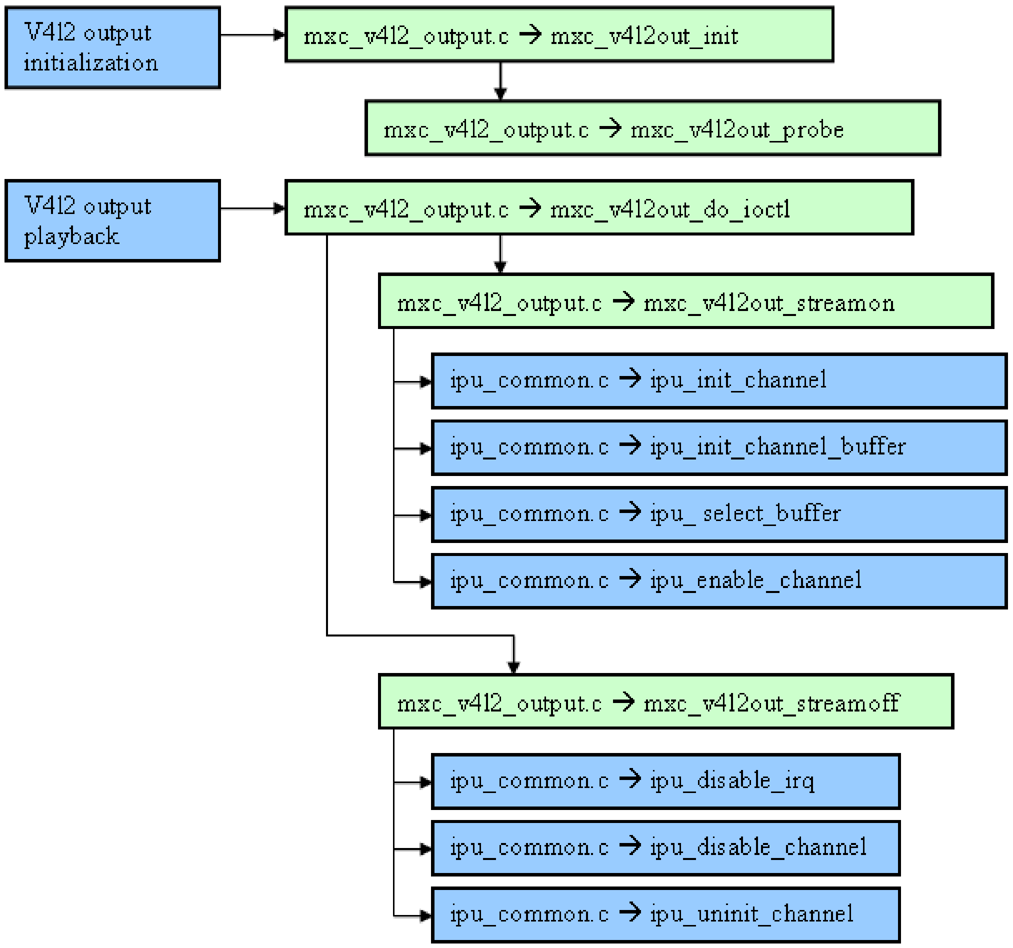

V4l2 output initialization
mxc_v4l2_output.c → mxc_v4l2out_init
mxc_v4l2_output.c → mxc_v4l2out_probe
V4l2 output playback
mxc_v4l2_output.c → mxc_v4l2out_do_ioctl
mxc_v4l2_output.c → mxc_v4l2out_streamon
ipu_common.c → ipu_init_channel
ipu_common.c → ipu_init_channel_buffer
ipu_common.c → ipu_ select_buffer
ipu_common.c → ipu_enable_channel
mxc_v4l2_output.c → mxc_v4l2out_streamoff
ipu_common.c → ipu_disable_irq
ipu_common.c → ipu_disable_channel
ipu_common.c → ipu_uninit_channel

**Figure 23. mxc_v4l2_output Flow Chart**

## 4.3 Panel Configurations

This section describes how to develop a system with a new LCD panel where the driver is not configured. To develop this, a new driver should be implemented in the Linux kernel. The driver configuration should be done by taking advantage of all the processor hardware designed for the respective tasks. For example, the IPU can be used in such a way to enhance the processor performance.

### 4.3.1 Case—WVGA Panel

This section describes how to develop a system with a new WVGA panel where the driver is not configured.

#### 4.3.1.1 Panel Generalities

In this case, CLAA070VC01 is used, which is a 7" color TFT-LCD module and is composed of the following:

- LCD panel
- Driver ICs
- Control circuit
- LED backlight

The 7.0" screen of the CLAA070VC01 module, which is composed of 800 × 480 pixel elements in a stripe arrangement, produces a high resolution image. The module has a 262 K color display with a 6-bit RGB signal input.

Table 14 gives the timing parameters of the CLAA070VC01 module.

**Table 14. Timing Parameters**

| Parameter | Symbol | Minimum | Typical | Maximum | Unit |
|---|---|---|---|---|---|
| Screen width or Horizontal cycle | HP | 850 | 900 | 950 | PIXCLK |
| Horizontal blank period | HBK | 50 | 100 | 150 | PIXCLK |
| Active frame width | HDISP | 800 | 800 | 800 | PIXCLK |
| Screen height or Vertical cycle | VP | 490 | 500 | 520 | Line |
| Vertical blank | VBK | 10 | 20 | 40 | Line |
| Active frame height | VDISP | 480 | 480 | 480 | Line |
| Vertical refresh rate | FV | 55 | 60 | 65 | Hz |

Table 15 gives the pin connection of the CLAA070VC01 module.

**Table 15. Pin Connection**

| Pin Number | Description | Parameter |
|---|---|---|
| 1 | VSS | Power ground |
| 2 | VSS | Power ground |
| 3 | ADJ | Brightness control for LED B/L |
| 4 | VDD | Power Supply for LED driver circuit |
| 5 | VDD | Power supply for LED driver circuit |
| 6 | VDD | Power supply for LED driver circuit |
| 7 | VCC | Power supply for digital circuit |

**Table 15. Pin Connection (continued)**

| Pin Number | Description | Parameter |
|---|---|---|
| 8 | VCC | Power Supply for digital Circuit |
| 9 | DE | Data Enable |
| 10 | VSS | Power Ground |
| 11 | VSS | Power Ground |
| 12 | VSS | Power Ground |
| 13 | B5 | Blue Data 5 (MSB) |
| 14 | B4 | Blue Data 4 |
| 15 | B3 | Blue Data 3 |
| 16 | VSS | Power Ground |
| 17 | B2 | Blue Data 2 |
| 18 | B1 | Blue Data 1 |
| 19 | B0 | Blue Data 0 (LSB) |
| 20 | VSS | Power Ground |
| 21 | G5 | Green Data 5 (MSB) |
| 22 | G4 | Green Data 4 |
| 23 | G3 | Green Data 3 |
| 24 | VSS | Power Ground |
| 25 | G2 | Green Data 2 |
| 26 | G1 | Green Data 1 |
| 27 | G0 | Green Data 0 (LSB) |
| 28 | VSS | Power Ground |
| 29 | R5 | Red Data 5 (MSB) |
| 30 | R4 | Red Data 4 |
| 31 | R3 | Red Data 3 |
| 32 | VSS | Power Ground |
| 33 | R2 | Red Data 2 |
| 34 | R1 | Red Data 1 |
| 35 | R0 | Red Data 0 (LSB) |
| 36 | VSS | Power Ground |
| 37 | VSS | Power Ground |
| 38 | DCLK | Clock Signals: Latch Data at falling edge |
| 39 | VSS | Power Ground |
| 40 | VSS | Power Ground |

### 4.3.1.2 Panel Configuration

To create a driver for the new panel, the `fb_videomode` structure should be configured properly. This structure contains the information about the timings, resolution, name, and configuration for the panel to work properly. Table 15 gives the parameters of the `fb_videomode` structure and the structure is as follows:

```
struct fb_videomode {
const char *name;           →        = CLAA-WVGA
u32 refresh;                →        = Refresh rate in Hz
u32 xres;                   →        = resolution in x
u32 yres;                   →        = resolution in y
u32 pixclock;               →        = Pixel clock in picoseconds
u32 left_margin;            →        = Horizontal Back Porch
u32 right_margin;           →        = Horizontal Front Porch
u32 upper_margin;           →        = Vertical Back Porch
u32 lower_margin;           →        = Vertical Front Porch
u32 hsync_len;              →        = Hsync pulse width
u32 vsync_len;              →        = Vsync pulse width
u32 sync;                   →        = Polarity on the Data Enable
u32 vmode;                  →        = Video Mode
u32 flag;                   →        = 0
};
```

Parameters of the `fb_videomode` structure are obtained from the panel data sheet. These are determined by using some of the timing concepts described in Section 3.2, “Synchronous Display Timing and Signals.” These parameters are as follows:

**NOTE**

Refer to the timing parameters in Table 14 and use the typical values (recommended).

- `const char` *name—is the name of a parameter and is enclosed in double chords
- `u32 refresh`—is the refresh rate and is given in Hz. Generally, the `u32 refresh` value is given directly as refresh rate or vertical refresh rate.
- `u32 xres`—is the resolution in the x axis or the number of pixels in horizontal position (H). It is easy to get this value as `u32 xres` is one of the most important and descriptive parameters in the LCD.
- `u32 yres`—is the parameter provided by the data sheet and is the resolution or number of pixels in the vertical position (V)
- `u32 pixclock`—is the pixel clock that is available in the data sheet as dot clock or clock and is generally given in MHz. However, these values should be provided in picoseconds (ps) for entering them into the `fb_videomode` structure. The clock frequency used in this case is 25 MHz.
- `u32 left_margin`—is parallel to the HBP. The value of this parameter is provided directly in some of the LCD data sheets. However, instead of providing the HBP, HFP, and HSYNC pulse width, some data sheets provide the total time, horizontal blank period, which is the sum of the HBP, HFP, and HSYNC pulse width. In this case, the value is calculated using the procedure described in Section 3.2.2.2, “Horizontal Timing Charts,” or by using the HBP and HSYNC pulse width. The HBP and HSYNC pulse width values should sum the horizontal blank period (the typical value). Though variations can exist between the two values, both of them should sum the horizontal blank period (100). This left margin is given in pixel clocks and the value in this case is 80 pixel clocks.

- `u32 right_margin`—is the HFP. The procedure adopted in this case avoids this parameter as the HFP is not considered in the calculation of the horizontal blank period. Therefore, the right margin in this case is 0 and is provided in pixel clocks.
- `u32 upper_margin`—is parallel to the VBP. The procedure to find the upper margin is similar to the left margin. Some data sheets give the VFP, VBP, and VSYNC pulse width, whereas some others provide a value, which is the sum of these three elements. This value is the vertical blank period. In this case, only the VBP (one of the porches) and VSYNC pulse width are taken to calculate the vertical blank period. Though variations can exist between the VBP and VSYNC pulse width, both of them should sum the vertical blank period. This upper margin is given in horizontal lines and the value in this case is 10.
- `u32 lower_margin`—refers to the VFP. The procedure to find the lower margin is similar to the right margin. As the vertical blank period provided by the LCD is only the sum of the VSYNC pulse width and the upper margin (VBP), the value of this parameter is 0.
- `u32 hsync_len`—is the HSYNC pulse width. Some LCD data sheets provide this value directly. However, some provide only the horizontal blank period, which is the sum of the right margin and this value. In this case, the `hsync_len` value is set to be 20 pixel clocks as the horizontal blank period and `left_margin` of this LCD are 100 and 80, respectively.
- `u32 vsync_len`—is the VSYNC pulse width. Similar to the case of the `hsync_len` parameter, some LCD data sheets directly provide this value. However, some provide only the vertical blank period, which is the sum of the `upper_margin` and this value. In this case, the `vsync_len` value is set to be 10 horizontal lines as the vertical blank period and `upper_margin` of this LCD are 20 and 10, respectively.
- `u32 sync`—refers to the signal polarity and determines the time for the valid data. Generally, this signals is DE or data ready. In this case, the value of `u32 sync` is same as `FB_SYNC_OE_ACT_HIGH`, which means that a high DE value marks the beginning of the valid data period.
- `u32 vmode`—is the video mode of the configuration. The i.MX offers two modes for the vmode—interlaced and non-interlaced. The video mode is stored in the `SDC_COM_CONF` register. In this case, the value is same as that of `FB_VMODE_NONINTERLACED` and is also named in the register options as TFT color.
- `u32 flag`—is not used and is generally left as 0.

The `fb_videomode` structure after calculating all the values is as follows:

```
static struct fb_videomode video_modes[] = {
{
        /* 800x480 @ 55 Hz , pixel clk @ 25MHz */
        const char *name;→  "CLAA-WVGA",
        u32 refresh; →  55,
        u32 xres; →  800,
        u32 yres; →  480,
        u32 pixclock; →  40000,
        u32 left_margin; →  80,
        u32 right_margin; →  0,
        u32 upper_margin; →  10,
        u32 lower_margin; →  0,
u32 hsync_len;→  20,
u32 vsync_len;→  10,
        u32 sync; →F B_SYNC_OE_ACT_HIGH,
        u32 vmode; →  FB_VMODE_NONINTERLACED,
        u32 flag; →  0,
}
```

### Procedure to Transfer Data into Framebuffer Configuration

The `fb_videomode` structure for the panel is declared in the WVGA panel driver. The information in the `fb_videomode` structure is passed to the `fb_var_screeninfo` structure (from the definition in `fb.h`, it is observed that the pixel clock parameter should be passed in picoseconds) that is defined in the `lcd_init_fb` function (which is called by the probe function). The data in the `fb_var_screeninfo` structure is passed to the general info structure in the `fb_set_var` function, which is also called by the `lcd_init_fb()` function.

When the `ipu_sdc_init()` function is called, many of the parameters are set to be configured in the register level or in the last stage of this transfer procedure. The processor considers the back porches and pulse width as a single parameter when the timing and back and front porch values are transferred to the i.MX register. Therefore, these values are added in the `ipu_sdc_init()` function and then passed to `SDC_HOR_CONF` and `SDC_VER_CONF` for the horizontal and vertical cases, respectively.

Figure 24 shows the flow chart that describes the procedure to transfer data into the framebuffer configuration.

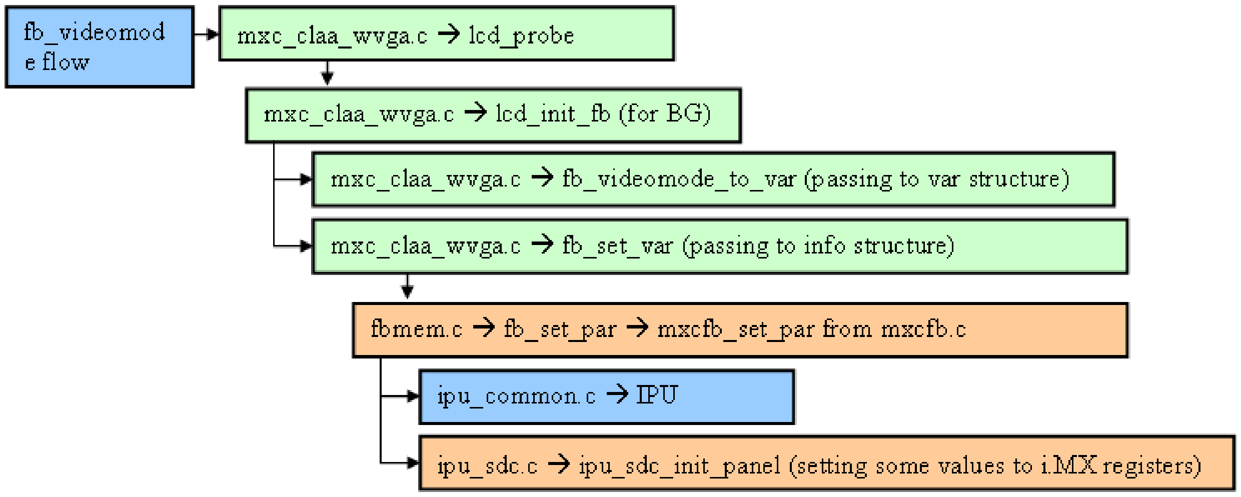

fb_videomode flow
mxc_claa_wvga.c → lcd_probe
mxc_claa_wvga.c → lcd_init_fb (for BG)
mxc_claa_wvga.c → fb_videomode_to_var (passing to var structure)
mxc_claa_wvga.c → fb_set_var (passing to info structure)
fbmem.c → fb_set_par → mxcfb_set_par from mxcfb.c
ipu_common.c → IPU
ipu_sdc.c → ipu_sdc_init_panel (setting some values to i.MX registers)

**Figure 24. Flow Chart to Transfer Data into Framebuffer**

## 4.3.1.3 Driver Development Process

This section describes the driver development process. This includes the description of the most important functions, main differences that make the new driver unique, and files where these differences should be addressed. The section also describes the general procedure and elements that should be considered when a panel driver is created.

### Panel Driver File

The CLAA070VC01 panel offers many advantages with respect to the interface and connection, voltage level management, and signals (besides the obvious fact of the difference in resolution) compared to other panels. Therefore, a new driver file should be created to cover the specific requirements of this panel.

Some of these changes are addressed in the new file while the changes that include the pin setup and LCD panel device structure registration are addressed in different files. However, the new panel driver has a structure similar to that of the `mxcfb.c` file and can be considered as a subset of the `mxcfb.c` file.

The panel driver file should contain the following structures and functions:

- `fb_videomode` structure with the timing and configuration data for the panel
- Char driver standard functions for the initialization and cleanup (generally, a platform driver structure is used for registering/unregistering)
- Platform driver structure with the standard pointers and functions (`probe`, `suspend`, `resume`, and so on) and their function implementations
- Function and structure related to the notifier chain facility for event processing, if this facility is used
- Functions to communicate and pass the flow of information between the panel and framebuffer generic structures
- Interface and voltage related functions for the LCD panel power management

The panel driver is the `mxcfb_claa_wvga.c` file and is located in the `drivers/video/mxc` folder. This file registers the driver and tests the driver functionality (using the probe function). The file also provides timings in the `fb_videomode` structure and contains some functions to interact with the panel for instances related to the events and ON/OFF functionality.

Important functions and structures in the `mxcfb_claa_wvga.c` file are as follows:

- Module definition and registering functions:
  - `platform_driver lcd_driver`—contains pointers to the power management and binding functions such as `probe`, `suspend`, and `resume` that are targeted to this panel. The `platform_driver lcd_driver` structure is as follows:

```
static struct platform_driver lcd_driver = {
         .driver = {
                                                   .name = "lcd_claa"},
         .probe = lcd_probe,
         .remove = __devexit_p(lcd_remove),
         .suspend = lcd_suspend,
         .resume = lcd_resume,
     };
```

  - `static int __init claa_lcd_init(void)`—function registers the `lcd_driver` structure of the type, `platform_driver`
  - `static void __exit claa_lcd_exit(void)`—function unregisters the `lcd_driver` structure of the type, `platform_driver`
  - `static int __devinit lcd_probe(struct platform_device *pdev)`—function is called when a device is installed and performs some functions. These functions include setting the voltage level, calling the function to initialize the LCD, switching ON the LCD, and calling a function to notify the kernel that a new event has happened (the LCD driver installation).
  - `static void lcd_init_fb(struct fb_info *info)`—function is called by the `lcd_probe` function. This function gives specific information about the panel (information about the parameters—resolution, size, and timings—of the `fb_videomode` structure) to the

fb_var_screeninfo structure and converts the information from the fb_videomode structure to parameters for the fb_var_screeninfo structure.

- Power and event related functions:
  - static void lcd_poweron(void)—function turns the panel ON
  - static void lcd_poweroff(void)—function turns the panel OFF
  - fb_register_client(&nb)—function is called by the lcd_probe function. This function registers a client notifier when an LCD driver event occurs. The client notifier contains a notifier block structure that contains information such as information about the pointer to the called function.
  - static int lcd_fb_event(struct notifier_block *nb, unsigned long val, void *v)—function is called when an LCD driver event occurs. The occurred event is then registered by this function.

**NOTE**

Some other critical panel elements that are not mentioned in this section are located in other files. These elements are related to the pin configuration (for the LCD connection) and LCD device structure initialization.

### LCD Initialization and Working

Similar to the general framebuffer initialization, the LCD driver also starts at an early stage of the kernel loading. As mentioned in Section 4.2.2, "Files and Important Elements," the registration of the LCD panel as a platform device structure is made in the mx3_3stack.c file. At this point, both the LCD panel and framebuffer are registered as the platform device elements.

In this stage, the processor pins are setup so that the LCD has a proper connection. All these tasks are developed in the mx3_3stack_gpio.c file that is located in the …linux-2.6.24/arch/arm/mach-mx3/ folder. Among the other features, the gpio_lcd_active function in the mx3_3stack_gpio.c file sets all the pins that are used by the panel with the help of the mxc_request_iomux function. The gpio_lcd_active function passes parameters (pins) to this function and the desired characteristics are obtained from each signal.

The gpio_lcd_active function is as follows:

```
void gpio_lcd_active(void)
{
         mxc_request_iomux(MX31_PIN_LD0, OUTPUTCONFIG_FUNC, INPUTCONFIG_FUNC);
         mxc_request_iomux(MX31_PIN_LD1, OUTPUTCONFIG_FUNC, INPUTCONFIG_FUNC);
         mxc_request_iomux(MX31_PIN_LD2, OUTPUTCONFIG_FUNC, INPUTCONFIG_FUNC);
         mxc_request_iomux(MX31_PIN_LD3, OUTPUTCONFIG_FUNC, INPUTCONFIG_FUNC);
         mxc_request_iomux(MX31_PIN_LD4, OUTPUTCONFIG_FUNC, INPUTCONFIG_FUNC);
         mxc_request_iomux(MX31_PIN_LD5, OUTPUTCONFIG_FUNC, INPUTCONFIG_FUNC);
         mxc_request_iomux(MX31_PIN_LD6, OUTPUTCONFIG_FUNC, INPUTCONFIG_FUNC);
         mxc_request_iomux(MX31_PIN_LD7, OUTPUTCONFIG_FUNC, INPUTCONFIG_FUNC);
         mxc_request_iomux(MX31_PIN_LD8, OUTPUTCONFIG_FUNC, INPUTCONFIG_FUNC);
         mxc_request_iomux(MX31_PIN_LD9, OUTPUTCONFIG_FUNC, INPUTCONFIG_FUNC);
         mxc_request_iomux(MX31_PIN_LD10, OUTPUTCONFIG_FUNC, INPUTCONFIG_FUNC);
         mxc_request_iomux(MX31_PIN_LD11, OUTPUTCONFIG_FUNC, INPUTCONFIG_FUNC);
         mxc_request_iomux(MX31_PIN_LD12, OUTPUTCONFIG_FUNC, INPUTCONFIG_FUNC);
         mxc_request_iomux(MX31_PIN_LD13, OUTPUTCONFIG_FUNC, INPUTCONFIG_FUNC);
         mxc_request_iomux(MX31_PIN_LD14, OUTPUTCONFIG_FUNC, INPUTCONFIG_FUNC);
```

```
        mxc_request_iomux(MX31_PIN_LD15, OUTPUTCONFIG_FUNC, INPUTCONFIG_FUNC);
        mxc_request_iomux(MX31_PIN_LD16, OUTPUTCONFIG_FUNC, INPUTCONFIG_FUNC); //LD16
mxc_request_iomux(MX31_PIN_LD17, OUTPUTCONFIG_FUNC, INPUTCONFIG_FUNC); //LD17
        mxc_request_iomux(MX31_PIN_VSYNC3, OUTPUTCONFIG_FUNC, INPUTCONFIG_FUNC); //VSYNC
        mxc_request_iomux(MX31_PIN_HSYNC, OUTPUTCONFIG_FUNC, INPUTCONFIG_FUNC); //HSYNC
        mxc_request_iomux(MX31_PIN_FPSHIFT, OUTPUTCONFIG_FUNC, INPUTCONFIG_FUNC); //CLK
        mxc_request_iomux(MX31_PIN_DRDY0, OUTPUTCONFIG_FUNC, INPUTCONFIG_FUNC); //DRDY
        mxc_request_iomux(MX31_PIN_D3_REV, OUTPUTCONFIG_FUNC, INPUTCONFIG_FUNC); //REV
        mxc_request_iomux(MX31_PIN_CONTRAST,OUTPUTCONFIG_FUNC,INPUTCONFIG_FUNC); //CONTR
        mxc_request_iomux(MX31_PIN_D3_SPL, OUTPUTCONFIG_FUNC, INPUTCONFIG_FUNC); //SPL
        mxc_request_iomux(MX31_PIN_D3_CLS, OUTPUTCONFIG_FUNC, INPUTCONFIG_FUNC); //CLS
}
```

At this point, all the pins are set properly. The following stages are from loading the generic framebuffer infrastructure (in the `fbmem.c` file) to loading and registering the structures and functions related to this panel. The process starts with the `fbmem.c` file execution and then, the IPU module is loaded. The framebuffer implementation for the i.MX family is then started by registering the framebuffers (normal and overlay) and the procedure completes with the panel driver.

The panel driver execution is also initiated with the initialization routines that registers the platform driver. However, most of the initialization procedure is performed under the probe function. Various actions that are performed by the `probe` function are as follows:

- Allocation of memory for the structures related to the panel
- Passing the `fb_videmode` structure, which is declared in the `fb_info` structure, with the data from the panel to the `var_screen_info` structure
- Functions to switch ON/OFF the panel and regulating the voltage levels using the functions from the regulation framework
- Registration of an LCD event for the kernel notifier chain register

All the information flow (from the `fb_videomode` structure) and important events happen in the `lcd_init_fb` function. The `fb_set_var` function, which is the part of the fops structure in the `fb_info` structure, is a link to the `mxcfb_set_par` function that is located in the `mxcfb.c` file. The `mxcfb.c` file is the implementation of `fb_set_pat` for the i.MX framebuffer. Here, among the most important things, the `ipu_sdc_init_panel` function is called with the arguments as the most of the `fb_videmode` structure elements. This function is the place where all these elements are eventually passed to the processor registers for the SDC module. Though the `lcd_init_fb` function is called twice in the probe function, only one time the function accesses the `ipu_sdc_init_panel` function. This is because, the function detects when the configuration should be made for the background or foreground. In this case, the function accesses only the SDC function when the configuration is made for the background.

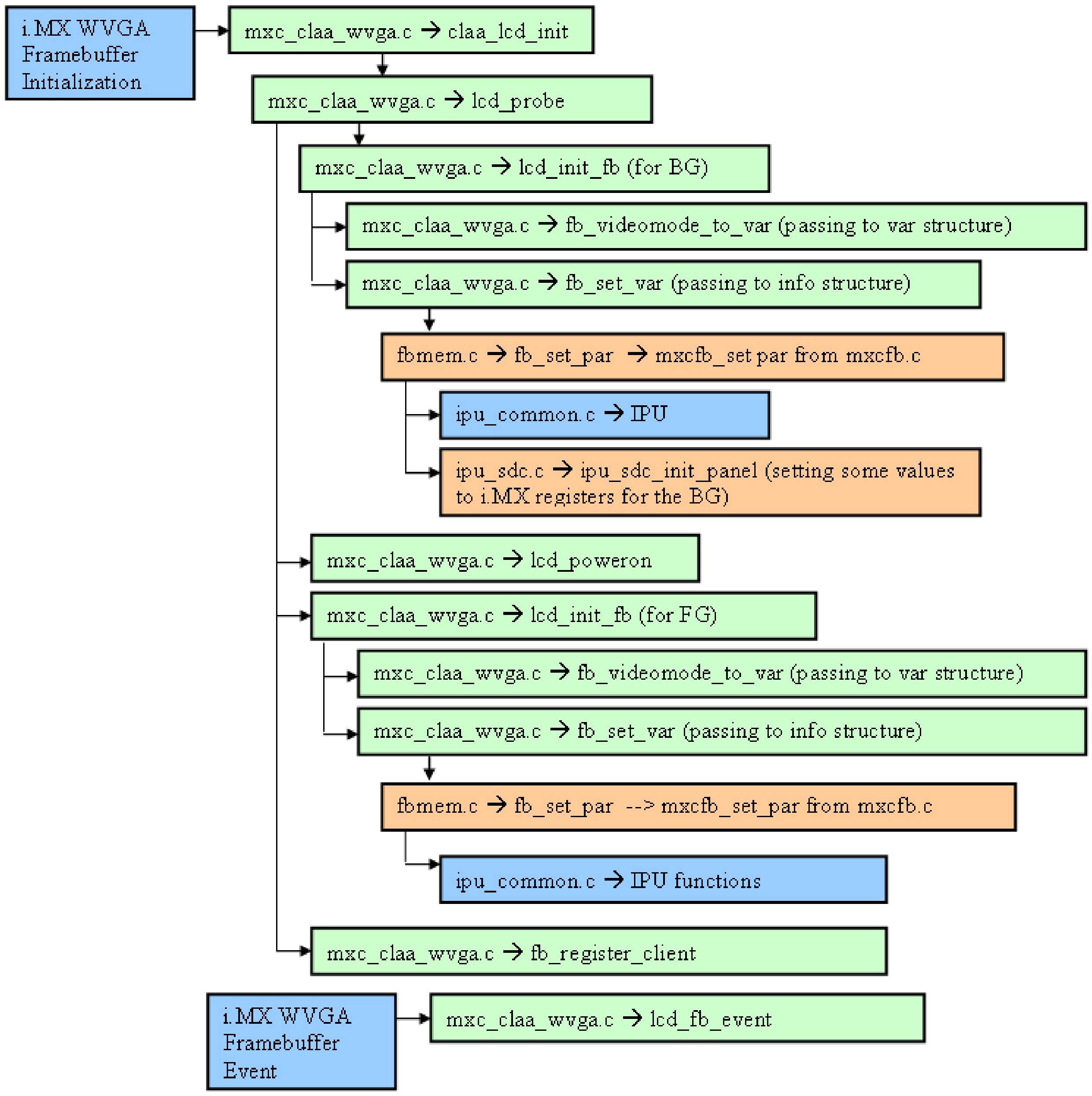
This condition is coded as follows:

```
if (mxc_fbi→ipu_ch == MEM_SDC_BG) {
                  memset(&sig_cfg, 0, sizeof(sig_cfg));
```

The condition asks if the logical IPU channel (that is used at that time) is employed for the background case. If this is correct the `ipu_sdc_init_panel` function is called. If not, that section is skipped.

The `probe` function calls the `lcd_poweron` function (when background only) using the functions in the voltage and current regulator frameworks (such as `regulator_enable`). The `probe` function then uses the `fb_register_client` function by passing a notifier block structure as an argument that contains a pointer to the `lcd_fb_event` function. This function is called whenever an LCD events happens (when Qtopia starts).

Figure 25 shows the flowchart for the `mxcfb_claa_wvga.c` file.



i.MX WVGA Framebuffer Initialization
mxc_claa_wvga.c → claa_lcd_init
mxc_claa_wvga.c → lcd_probe
mxc_claa_wvga.c → lcd_init_fb (for BG)
mxc_claa_wvga.c → fb_videomode_to_var (passing to var structure)
mxc_claa_wvga.c → fb_set_var (passing to info structure)
fbmem.c → fb_set_par → mxcfb_set par from mxcfb.c
ipu_common.c → IPU
ipu_sdc.c → ipu_sdc_init_panel (setting some values to i.MX registers for the BG)
mxc_claa_wvga.c → lcd_poweron
mxc_claa_wvga.c → lcd_init_fb (for FG)
mxc_claa_wvga.c → fb_videomode_to_var (passing to var structure)
mxc_claa_wvga.c → fb_set_var (passing to info structure)
fbmem.c → fb_set_par --> mxcfb_set_par from mxcfb.c
ipu_common.c → IPU functions
mxc_claa_wvga.c → fb_register_client
i.MX WVGA Framebuffer Event
mxc_claa_wvga.c → lcd_fb_event

**Figure 25. mxcfb_claa_wvga Flow Chart**

## Linux Voltage and Current Regulator Framework

Linux voltage and current regulator framework is a part of the panel driver and is used as an interface for controlling the voltage and current levels in the Linux Kernel 2.6. This framework also provides the information to the user with the help of sysfs interface. The framework works with PMICs such as the MC13787 from Freescale and Wolfson WM8350.

The functions provided in the panel driver comes from the …`linux-2.6.24/drivers/regulator/reg_core.c` file. This file is also located in the folder that contains the code for the PMIC, which is used in the 3-Stack board (MC13783). Some of the functions that are used in the panel driver are the `regulator_get`, `regulator_set_voltage`, and `regulator_enable` functions.

The Linux voltage and current regulator framework is made by Liam Girdwood from the Wolfson Microelectronics. More information about this framework can be found in the link, http://opensource.wolfsonmicro.com/node/15.

### Notifier Chain

The notifier chain or notifier block is a part of the panel driver and is an information mechanism where different elements are notified to the asynchronous events of the kernel. The basic element of the notifier chain is the notifier block structure (the definition is available in the …`linux-2.6.24/include/linux/notifier.h` file). The notifier block structure (`notifier_block`) includes a reference event function in a struct member. In the panel driver, the notifier block function declaration receives the function (`lcd_fb_event` function) event reference as the argument. At the end of the probe function, the registration of the event is made by the `fb_register_client` function. The `fb_register_client` function, which resides inside the …`linux-2.6.24/drivers/video/fb_notify.c` file, calls the function in charge or assembles the notifier block that is passed as an argument to the notifier chain. Therefore, with the registration function, a callback is registered whenever a change in the LCD occurs.

## 5 References

- *MCIMX31 and MCIMX31L Applications Processors Reference Manual* (MCIX31RM)
- *i.MX31 PDK 1.5 Linux Reference Manual* (926-77210)
- *LTM08C351S Product Information* (Toshiba Matsushita Display Technology Co. Ltd)
- *Embedded Linux Systems Design and Development* (Raghavan P, Lad Amol, Neelakandan Sririam, Auerbach Publications)

## 6 Revision History

Table 16 provides a revision history for this application note.

**Table 16. Document Revision History**

| Rev. Number | Date | Substantive Change(s) |
|---|---|---|
| 0 | 08/2010 | Initial release. |

THIS PAGE INTENTIONALLY LEFT BLANK

***How to Reach Us:***

**Home Page:**
www.freescale.com

**Web Support:**
http://www.freescale.com/support

**USA/Europe or Locations Not Listed:**
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
1-800-521-6274 or
+1-480-768-2130
www.freescale.com/support

**Europe, Middle East, and Africa:**
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

**Japan:**
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku
Tokyo 153-0064
Japan
0120 191014 or
+81 3 5437 9125
support.japan@freescale.com

**Asia/Pacific:**
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
support.asia@freescale.com

**For Literature Requests Only:**
Freescale Semiconductor
Literature Distribution Center
1-800 441-2447 or
+1-303-675-2140
Fax: +1-303-675-2150
LDCForFreescaleSemiconductor
@hibbertgroup.com

Document Number: AN4182
Rev. 0
08/2010