Freescale Semiconductor
Application Note

Document Number: AN3974
Rev. 0, 01/2010

Different Display Configurations on i.MX35 Linux PDK

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

This application note provides information, considerations and steps to add or adapt a new display panel to the BSP distribution for i.MX35 PDK. It provides details about general panel and generalities of display controller from the module. It also describes the development process to adapt a new panel to the Board Support Package (BSP) taking into consideration the framework driver structure provided by the operating system. This application note assumes that the reader is familiar with LTIB packages and Linux device driver concepts.

The i.MX35 multimedia processor supports many display types. The Image Processing Unit (IPU) handles the display devices. This module also controls graphic interfaces such as cameras and 2D graphics acceleration. All IPU sub-modules are connected using a private DMA interface (IDMA). The IDMA is used only for the IPU to transfer data between sub-modules and also between IPU and the external memory.

Contents

freescale™ semiconductor

Figure 1 shows a functional diagram of IPU.

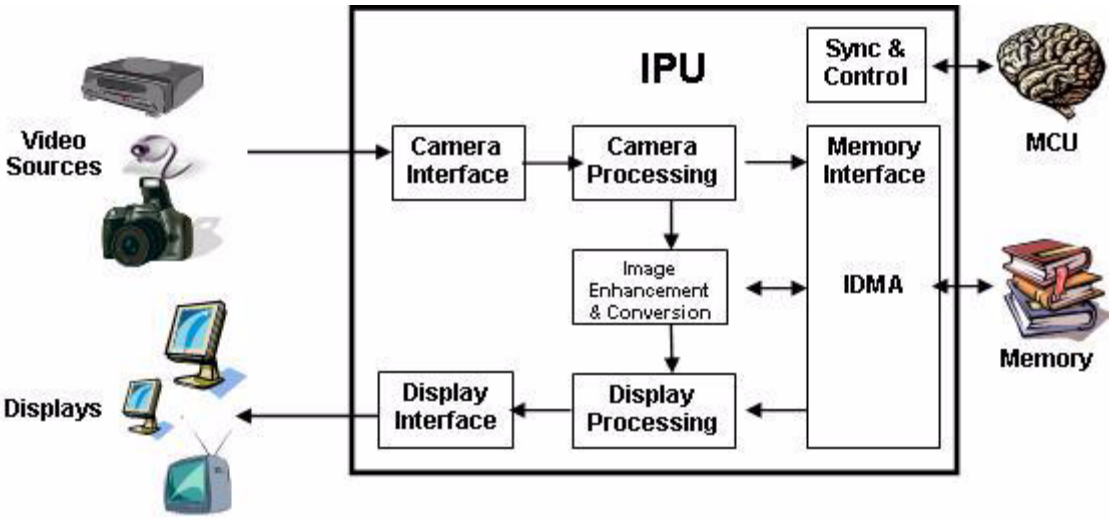

Figure 1. IPU Functional Diagram

The selection of proper Liquid Crystal Display (LCD) in a mobile device is a challenge and involves conflicts in the requirements. The following are examples of such conflicts:

- Large amount of data implying high rate of data transfer and processing and requiring significant resources.
- Flexibility to support a variety of use cases.
- Size, cost and power consumption.

Freescale's reference designs for the i.MX family demonstrate the functionality of the LCD, however developers can 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. Another important fact regarding LCD panels is that many displays become obsolete quickly. For this reason, it may be hard to find the same LCD panel included in the reference design when the users are creating their own product.

While some information would be useful for Smart Displays, this application note is intended only for dumb displays and especially those which do not have a SHARP synchronous interface. Do not confuse the SHARP LCD panels with the SHARP interface. Many SHARP LCD panels do not use the SHARP interface.

1 LCD Principles

This section describes the principles of a LCD.

1.1 LCD Basics

A LCD is an electronic device which consists of an array of pixels of color or monochrome units. Every element in the array consist of a special material, which allows them to change the characteristics of the

light that passes through them. These devices cannot emit light and because of this reason another element called backlight is usually shipped with the panel to create a fully functional display device.

1.1.1 Resolution

Resolution is the number of pixels contained in the LCD array. It has two dimensions: horizontal and vertical. There are some standard resolutions available in the market that must be followed.

Some of the most common video resolution standards are shown in Table 1.

Table 1. Common Video Resolution Standards

Video Name	Description	Width	Height	Aspect Ratio
CGA	Color Graphics Adaptor	320	200	8:5
QVGA	Quarter VGA	320	240	4:3
VGA	Video Graphics Array	640	480	4:3
NTSC	National Television System Committee	720	480	4:3
PAL	Phase Alternating Line (TV)	720	576	4:3
WVGA	Wide VGA	800	480	5:3
SVGA	Super VGA	800	600	4:3
WSVGA	Wide Super VGA	1024	600	—
XGA	Extended Graphic Array	1024	768	4:3

There are many resolution standards, but as SVGA is the maximum standard resolution supported by the i.MX35 processor, larger resolutions are not included in this table.

All resolutions mentioned in Table 1 refer to a landscape orientation, which means that there are more pixels in the horizontal axis than the vertical. However, Portrait LCD panels are also available in the market with the same standard resolution but the horizontal and vertical dimensions are inverted. In that case, Portrait LCD panels have more vertical pixels than horizontal pixels.

Figure 2 shows the Portrait and Landscape orientation of the LCD panels.

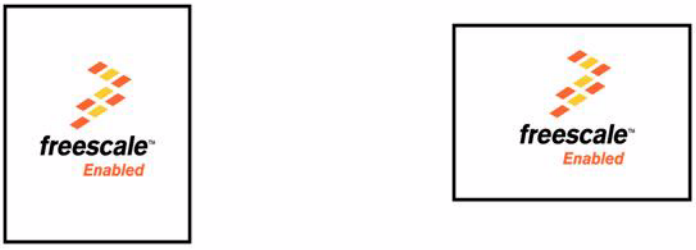

Figure 2. Portrait and Landscape Orientation

Selecting a proper LCD orientation is important, because both electronic and optical features are optimized for applications that use the native orientation of the panel. Besides the optical characteristics, dumb displays include an embedded LCD controller, and the chip draws the pixels from left to right and also

from top to bottom. In some scenarios, images or LCDs are shown using a nonnative orientation. In that case, display contents should be preprocessed to create a buffer where the image is written with the purpose of matching the way (order) in which the LCD controller expects to receive the pixel information. This operation is called rotation and i.MX35 includes hardware to perform it. It is recommended to select an LCD panel which can work using the native orientation, to avoid image processing overhead.

Figure 3 shows both portrait and Landscape LCD panels displaying images in a nonnative orientation.

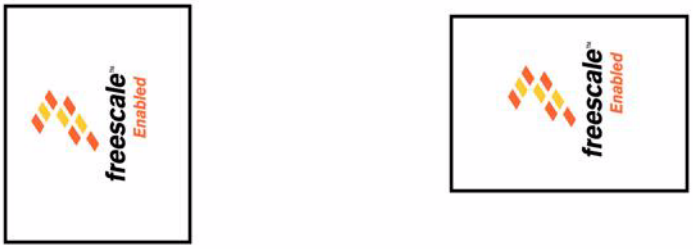

Figure 3. Rotated Frames on Portrait and Landscape Orientation

It is important to mention that rotation could be 90°, 180° or 270° (that is, it is not limited only to 90°). Note that if such rotations are chosen, then each frame should be rotated before sending them for the display.

1.1.1.1 Size

The size of an LCD panel is usually measured diagonally in inches, from corner to corner. When choosing between a VGA (640 x 480) panel and a QVGA (320 x 240) panel, one could expect the VGA panel to be of a larger size as it has more pixels (four time as many), but this is not always the case. LCD manufacturing processes allow size and resolution to be independent variables; it is hard to determine the size of a panel only from its resolution. Screens that are lager in size consume more power than smaller ones and they also impact the size and weight of the final product. On the other hand, high resolutions on small LCD screens can complicate the visibility of on-screen objects for the final user. Sometimes it is hard to determine how well a particular LCD panel l fits in an application based only on the information from the datasheet.

1.1.1.2 Color Spaces

A color space is used to represent colors. There are two main color spaces, RGB (that is, RGB565, RGB888, RGBA8888) and YUV (that is, YUV 4:4:4, YUV 4:2:2, YUV 4:2:0). The i.MX35 supports both the color spaces.

NOTE

Display panels can receive only data using the RGB color space.

1.2 LCD Types

This section describes the various LCD types.

1.2.1 Synchronous Panel (Dumb Display)

Dumb display or synchronous display stands for panels which require the system to send continuous frame data. The refresh is done by sending all pixels that compose a full frame, for every single frame. In general, smart displays are more expensive than dumb displays, and that is one of the reasons why it is more common to use synchronous (dumb) panels in a final product. This application note focusses on Thin Film Transistor Liquid Crystal Display (TFT) which is a particular group of synchronous panels.

1.2.1.1 Asynchronous Panel (Smart Displays)

The advantage of Smart displays is that the i.MX35 only has to send display data when the image has changed, and most of the times, it can send only the data for the portion of the image that has changed. Images can be sent at any time and the embedded Smart LCD display controller handles the screen refresh. The i.MX35 processor can handle up to three asynchronous displays simultaneously and can handle synchronous interface at the same time. This means that if the application requires two LCD panels, then one of them must use an asynchronous interface.

2 Synchronous Display Interface

The i.MX35 Synchronous Display controller can be configured to handle the following four different types of devices:

- TFT monochrome
- TFT color
- YUV progressive
- YUV interlaced

This document focuses only on the Synchronous TFT Color interface. For these cases, i.MX35 provides a 28-line interface which is described in Table 2.

Table 2. 28-Line Interface

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 [7:0]	DISPB_DATA[23:16]	Pixel Red component
Green Data[7:0]	DISPB_DATA[15:8]	Pixel Green Component
Blue Data [7:0]	DISPB_DATA[7:0]	Pixel Blue Component

The definitions of the signals referred in Table 2 are as follows:

HSYNC — Horizontal synchronization signal, also known as FPLINE or LP, indicates to the LCD that a line has ended and the following valid pixels are part of the next line.

VSYNC	Vertical synchronization signal, also known as FPFRAME, FLM, SPS or TV. When this signal is active, it indicates to the LCD that the current frame has ended. The LCD display must then restart the line index to zero to draw the next valid data in the first line of the panel.
DRDY	When data ready (DRDY) or data enable (DE) is active, it indicates to the LCD that the data in the RGB bus is valid and must be latched (latching happens using the PIXCLK signal). While data enable is active, every PIXCLK pulse makes the LCD draw a pixel using the color described in the RGB bus. The width of this signal should store all pixels (that is, as long as the sum of all pixel clock cycles are in a single line).
PIXCLK	Pixel clock signal specifies when RGB data is placed on the bus. There are two possibilities. The first option is when data is written by i.MX35 to the RGB bus on falling edges. In this case, data is stable and ready to be latched by the LCD panel on the rising edges, as long as data enable is active. This behavior corresponds to a high PIXCLK polarity. The second option is called low PIXCLK polarity and it means that i.MX35 writes RGB data in rising edges, and the data are latched by the LCD panel on the falling edges.
RGB Data	The i.MX35 can internally use different bit depths per pixel, such as RGB565, RGB666, RGB888, RGBA8888 and so on. In the same manner, the display interface could be configured to support more than one color depth. The i.MX35 processors can use up to 24 data lines (RGB888) as display interface bus. If the color depth used internally is bigger than the display interface, then a RGB to RGB conversion is performed where least significant bits are removed from the pixel, and the remaining bits are sent directly to the display interface. Dithering or filter actions are not performed during this process. The remaining RGB data lines can be used for other purposes including GPIO.

2.1 Extra Signals

There are also some other lines that are usually included on the panel interface. These signals are not part of the 28-line display interface, but they are required for a fully functional module. For example, it is common for some panels to have a reset signal, as well as initialization commands. These commands are usually sent through a serial interface such as I^2C or SPI. Display panels sometimes have touch panels embedded in them and have a backlight source, which requires additional signals.

2.2 SPI Interface

Some LCD displays require an initialization routine through a serial interface, 3-wire, 4-wire, or 5-wire. Even when the i.MX35 IPU has a serial interface (SD_D_CLK, SD_D_IO and SD_D_I), this interface should not be used to send serial commands to the LCDs. This interface is not intended for general purpose usage, instead, it is used only by the IPU when the Asynchronous Display1 or Display2 are configured to use serial interface.

2.3 Synchronous Display Interface Examples

Some of the examples of a synchronous display interfaces are described in the following sections.

2.3.1 i.MX35 PDK Chunghwa 5.7" VGA LCD Interface

Figure 4 shows an interface between i.MX35 and Chunghwa CLAA057VA01CTVGA Panel.

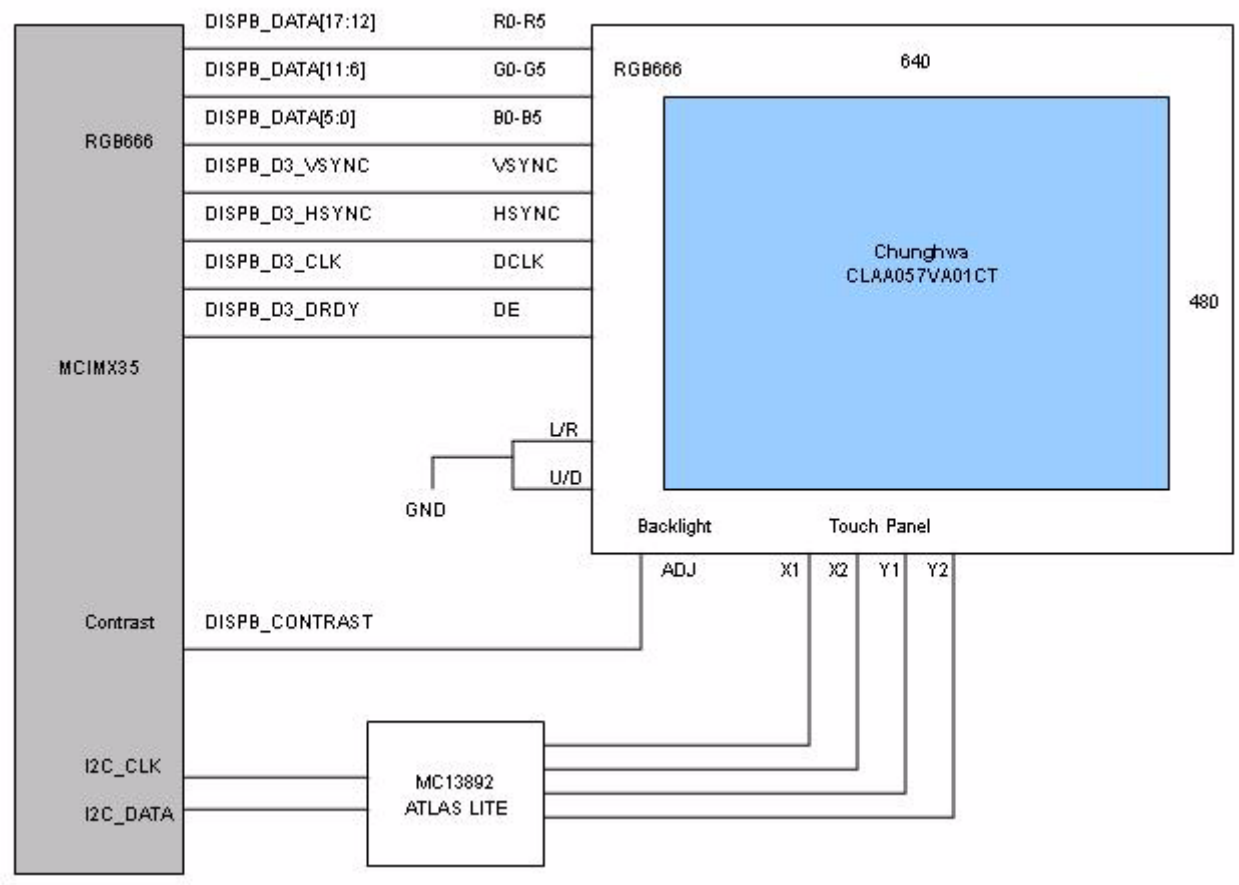

Figure 4. LCD Interface Between i.MX35 and Chunghwa CLAA057VA01CTVGA Panel

As shown Figure 4, the LCD panel requires HSYNC, VSYNC, DE, PIXCLK and the complete RGB data interface (DISPB_DATA[17:0]). Additional signals such as RESET signal or serial interface initialization routine commands (SPI or I^2C) are not required. The backlight unit is controlled by using a PWM signal generated by the i.MX35 (Contrast). The touch panel interface is handled by the MC13892 ATLAS LITE chip.

NOTE

Touch Panel functionality is not covered in this application note.

Each panel has its own interface and requirements, but in general terms, Figure 4 illustrates a typical synchronous panel interface. Determining the typicality of LCD panels is difficult, but the idea shown in Figure 4 can be used as a base for panel interface. The base interface idea is useful when there are many panels which do not use the complete interface. For example, some panels do not require HSYNC or VSYNC signal, or neither of them. Only DRDY, PIXCLK and RGB data are used. Also, many panels do not require a RESET signal or a serial initialization routine to handle display signals. When there is no serial interface, the timing, signals, porches and polarities are specified (fixed). The panels expect that the microprocessors complies with these waveforms, as such panels cannot handle a different interface.

2.3.2 i.MX35 PDK Chunghwa CLAA070VC01 7" WVGA LCD Interface

Figure 5 shows a LCD interface between i.MX35 and Chunghwa CLAA070VC01CWVGA Panel.

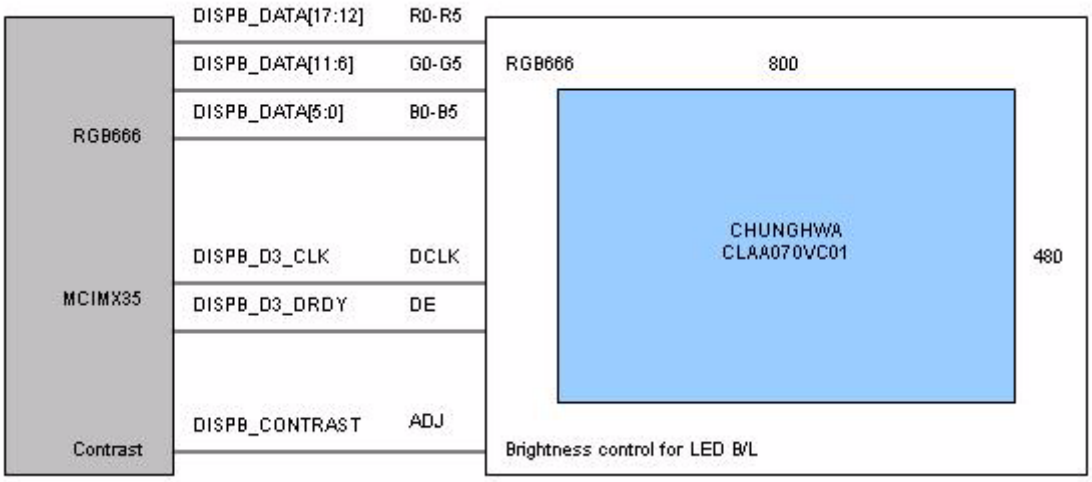

Figure 5. LCD Interface between i.MX35 and Chunghwa CLAA070VC01 WVGA Panel

This LCD is shipped with the i.MX35 PDK, and it shows a very simple display interface where HSYNC and VSYNC signals are not used. Both HSYNC and VSYNC can be used for other purposes, including GPIO.

The SPI interface is not required as there is also a Chip Select (CSPI1_SS2) available for other devices. Additionally, the power booster for the backlight unit is included in the module, which means that Contrast signal is directly connected to the Display connector. This display module does not include a touch panel, so it is necessary to add an external touch screen to the LCD panel.

In Figure 5 as in the previous 5.7" VGA LCD interface, only 18 RGB data lines are required, DISPB_DATA[17:0]. The remaining RGB data lines DISPB_DATA[23:18] can be used for other purposes such as GPIO or any other alternate function that each pin can perform.

Another drawback of this module is, the display cannot be turned off, since it does not have a RESET signal or SPI interface. This feature is particularly important for mobile devices where power consumption is an issue. If the energy used by the LCD has to be controlled, then external circuits should be used to control the energy. In case of PDK, power ON/OFF settings are handled by an external 8-bit Microprocessor driven by the i.MX35 through an I^2C bus.

Based on the above observations, the complete LCD circuit for the i.MX35 PDK system looks as shown in Figure 6.

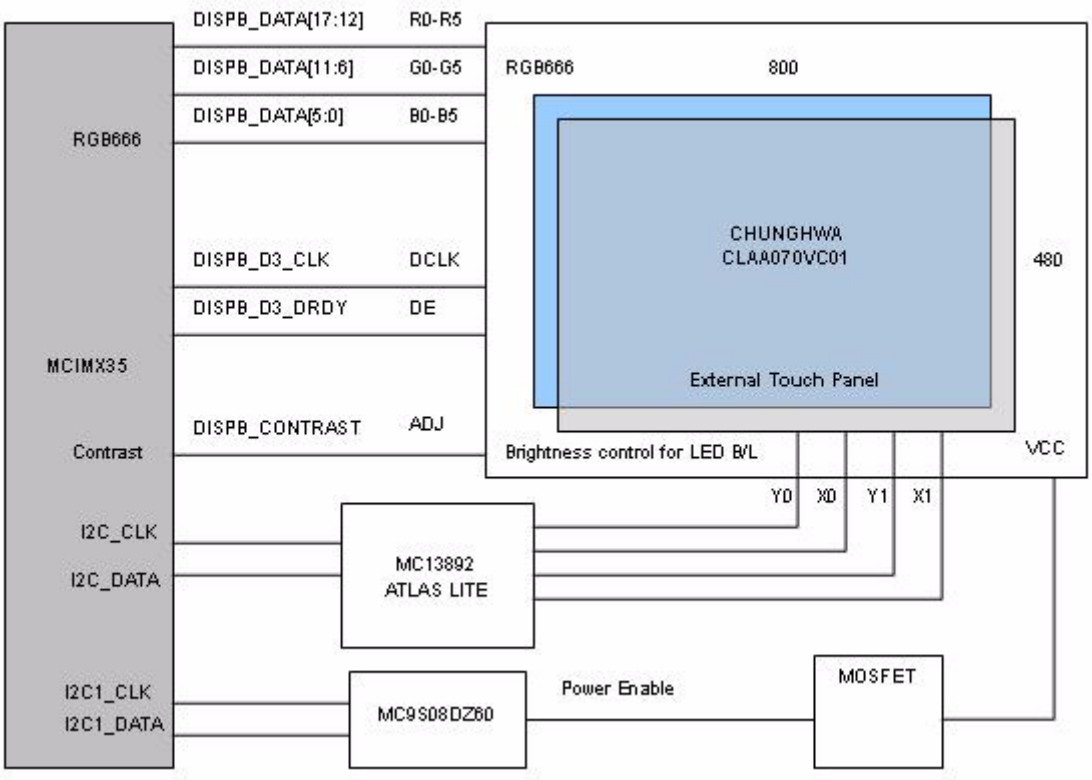

Figure 6. LCD Interface Between i.MX35 and Chunghwa CLAA070VC01VWGA Panel + Touch Panel

The previous examples can be helpful for selecting an LCD display, to determine the advantages and disadvantages of a particular panel.

2.4 Synchronous Display Timing and Signals

This section describes the timing and signal waveforms for configuring them in the LCD panel and i.MX35 Display interface. Before selecting a LCD panel, refer the datasheet. This datasheet must show the pin interface, the initialization routine, and the timing charts for the RGB interface as well as the serial interface, if needed. Sometimes, a shorter version of the datasheet is also received, where not all information is given. In such situations, request the full documentation from the supplier. Many times, the document is received with a big watermark in every page telling that it is a preliminary datasheet, and having the final version is recommended.

NOTE

There are not many modifications between a preliminary version of LCD datasheet and the final version.

2.4.1 Timing Concepts

The timing concepts that form the basis for LCD timing are explained below:

Horizontal Back Porch (HBP) Number of PIXCLK pulses between HSYNC signal and the first valid pixel data.

Horizontal Front Porch (HFP) Number of PIXCLK between last valid pixel data in the line and the next HSYNC pulse.

Vertical Back Porch (VBP) Number of lines (HSYNC pulses) from when a VSYNC signal is asserted and the first valid line.

Vertical Front Porch (VFP) Number of lines (HSYNC pulses) between last valid line of the frame and the next VSYNC Pulse.

VSYNC pulse width Number of HSYNC pulses when a VSYNC signal is active.

HSYNC pulse width Number of PIXCLK pulses when a HSYNC signal is active.

Active Frame width The Horizontal Resolution, which means the number of pixels in one line. For example, for a WVGA display (800H × 480V), the frame width is equal to 800 pixels.

Active Frame Height The Vertical Resolution of the LCD, using the same WVGA (800H × 480V) for example, the value of the frame height is 480 lines.

Screen Width Number of pixel clock periods between the last HSYNC and the new HSYNC. So, this value includes the valid pixels and also the Horizontal Back and Front porches.

$$SCREEN\_WIDTH = ACTIVE\_FRAME\_WIDTH + HBP + HFP \quad \textbf{Eqn. 1}$$

Screen Height Number of lines between VSYNC pulses. It includes all valid lines and also the Vertical, Back and Front porch.

$$SCREEN\_HEIGHT = ACTIVE\_FRAME\_HEIGHT + VBP + VFP \quad \textbf{Eqn. 2}$$

VSYNC polarity It is the value that VSYNC takes to indicate the starting of a new frame. It can be ACTIVE LOW when value is 0 or ACTIVE HIGH when it is 1.

HSYNC polarity It is the value which the HSYNC takes to indicate the starting of a new line. It can be ACTIVE LOW when value is 0 or ACTIVE HIGH when it is 1

2.4.2 Timing Charts

This section reviews the following charts to clarify the timing issues in a LCD interface. Any datasheet should include this kind of information. In general, three different charts are available as given below:

- The first one should cover the vertical timing.
- The second one specifies the horizontal timing.
- A third one with the pixel clock characteristics.

Additionally, if the display uses a serial interface, then a chart describing the serial interface and RESET is available. This information must be extracted from the datasheet when support for a new LCD panel should be added to a BSP. For example, when a VGA (640H x 480V) LCD panel is used instead of a Chunghwa CLAA057VA01CT, see Figure 5, then the display uses complete RGB interface: RGB666, VSYNC, HSYNC, Data Enable and Pixel clock.

2.4.2.1 Vertical Timing

Vertical Timing can be categorized as VGA vertical timing and WVGA vertical timing. These vertical timings are discussed in the following sections.

3.4.2.1.1 VGA Vertical Timing

Figure 7 shows vertical timing chart for a hypothetic synchronous display VGA (640H x 480V).

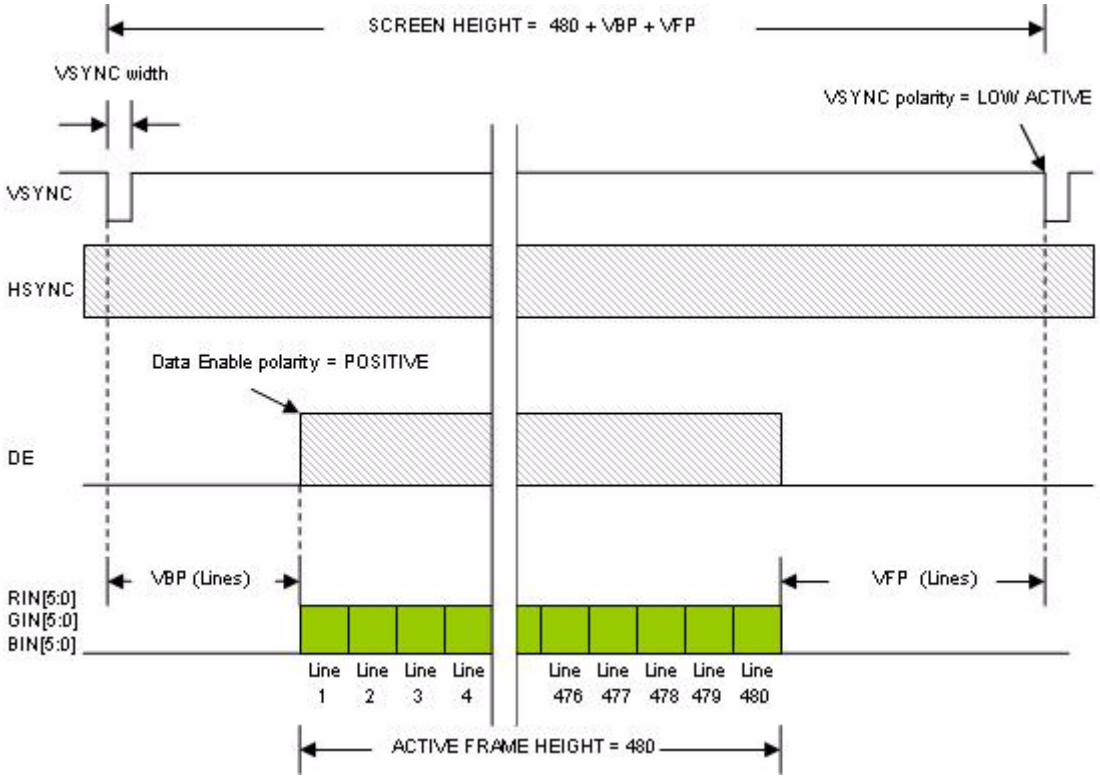

Figure 7. VGA Vertical Timing Example

It is important to mention how signals appear during a VSYNC period. The VSYNC period involves a complete frame cycle. Every pixel of each line in the frame is sent to the panel during this period. The beginning of the frame is asserted by the VSYNC signal (in this case, when the signal goes low). Then, HSYNC immediately marks the beginning of the first line (in this example, it is when HSYNC goes low). But, in order to meet the LCD timing requirements, the first lines are designated for the VBP. During VBP, data enable signal is not present, and the pixel data on the bus during these lines is ignored by the panel. After VBP, DE signal appears inside the boundaries of the HSYNC period. Details about DE during a line cycle are described in the next section. DE appears consequently during all valid lines (Vertical resolution

= 480V). During this time (ACTIVE FRAME HEIGHT), the LCD panel latches the RGB data on all lines and draws it on the screen. The final stage in the frame cycle is the VFP, where extra lines (HSYNC cycles) appear. During this time, DE remains inactive and again the panel discards any information on the RGB bus. The frame ends when VSYNC signal is asserted again (goes low).

Along with the chart, a table showing the range of the timing parameters can be seen as shown in Table 3.

Table 3. Timing Parameters Range

Parameter	Symbol	Min	Type	Max	Unit
Screen Height or Vertical period	VP	515	525	560	Line
VSYNC pulse width	VSW	1	1	1	Line
Vertical Back Porch+VSYNC	VBP	34	34	34	Line
Vertical Front Porch	VFP	1	11	46	Line
Active Frame Height	VDISP	—	480	—	Line
Vertical refresh rate	FV	55	60	65	Hz

From the above table, the timing features can be verified. In the first waveform, it is shown that VSYNC polarity is ACTIVE LOW, which means that vertical synchronization is normally high, but goes low to indicate the beginning of the new frame. Another feature that can be checked is the VSYNC width (VSW). Timing has certain flexibilities, so timing could be set using more than one value. It is highly recommended to use the typical values, or any values close to them, so use one line as VSYNC width.

VBP and VFP are also shown. Note that these values are measured in Lines or which translates in HSYNC pulses. In this example, VBP could be 34 lines, and VFP is 11 lines wide. Also note that VSYNC width is included into the VBP stage. This means that VBP starts when VSYNC is asserted, and not when the VSYNC returns to the normal state. Using the values described above, the Screen height or Vertical cycle is 525 lines. In some cases, the value of the VBP and VFP is not given in lines; instead it is expressed in nanoseconds or milliseconds. In such cases, additional calculations have to be done to find the number of lines needed to meet those timings.

3.4.2.1.2 WVGA Vertical Timing

If an LCD panel like the hypothetical WVGA (800H x 480V) shown in Figure 6 and Figure 7, where HSYNC and VSYNC signals are not used, then the waveforms should be analyzed using another perspective.

Figure 8 shows an example of WVGA Vertical timing.

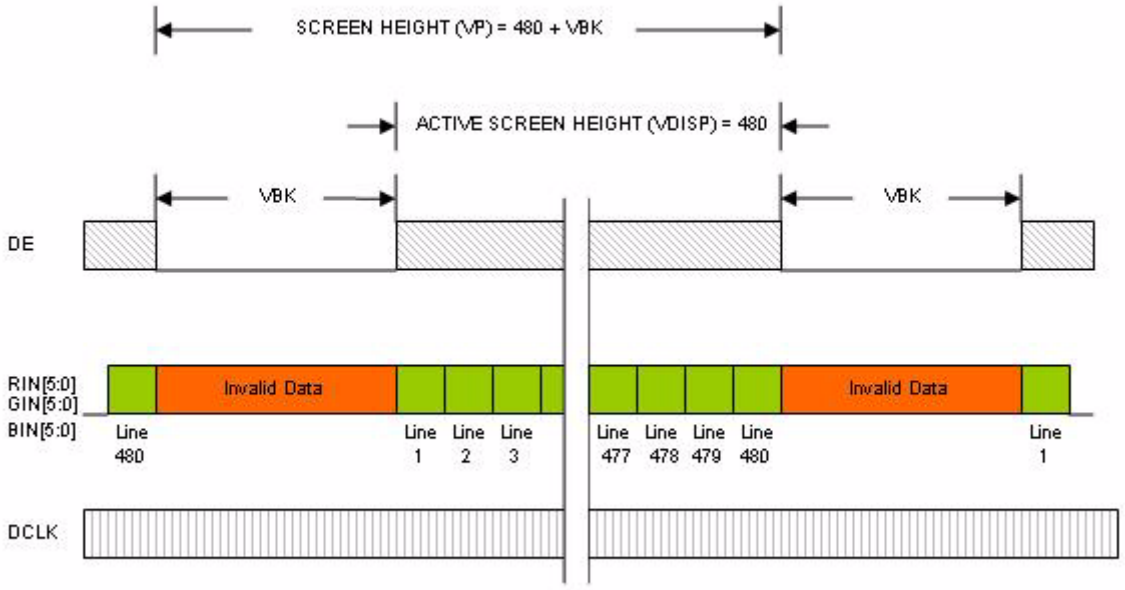

Figure 8. WVGA Vertical Timing Example

Table 4 shows the range of timing parameters.

Table 4. Timing Parameters Range

Parameter	Symbol	Min	Type	Max	Unit
Screen Height or Vertical Cycle	VP	90	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

In such cases, VSYNC width, VSYNC polarity, VBP and VFP are not given in the chart. Even when VSYNC is not used, the values are required to be configured the i.MX35 Display Interface. This waveform can be used to understand vertical cycle behavior. For the i.MX35, the sequence remains the same: vertical cycle starts with the VSYNC signal, and then the rest of the VBP follows, then the active frame area, and finally the VFP appears, until the next VSYNC is asserted.

The VSYNC width, VBP and VFP can be found out based on the events occurring during the Vertical Blank Period.

Figure 9 shows an example of WVGA vertical timing.

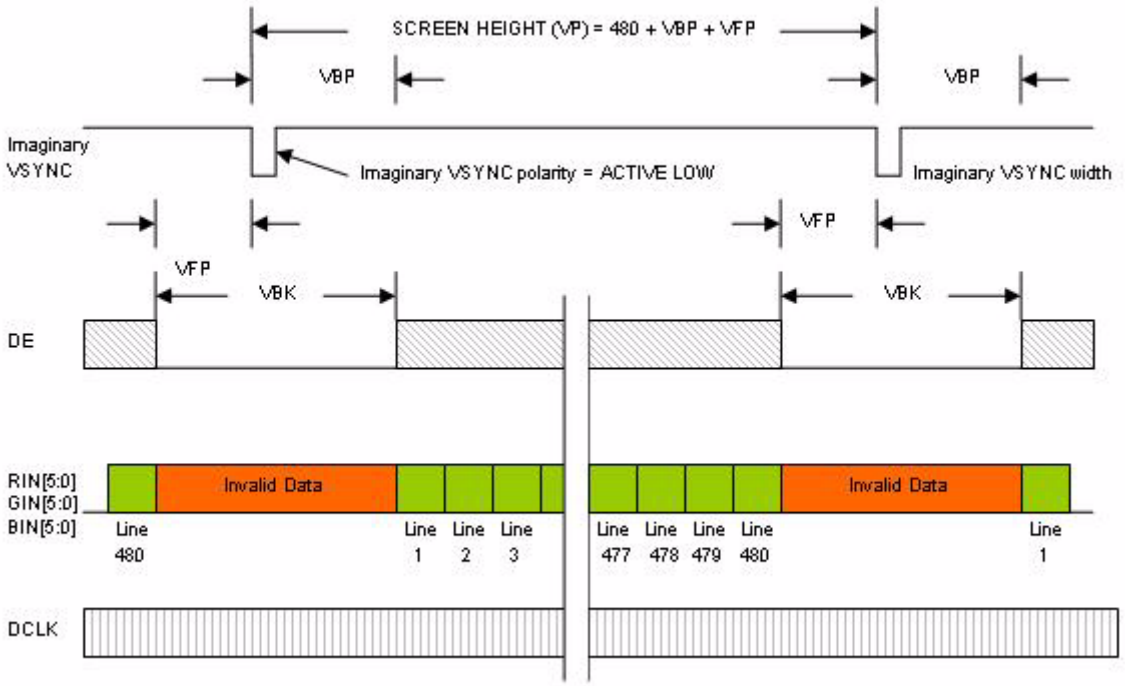

Figure 9. WVGA Vertical Timing Example with VSYNC Signals

The VSYNC signal can be used as a base to calculate the VFP and VBP. Since it is not used, any polarity can be set. However, setting VSYNC as ACTIVE LOW is recommended. VSYNC width is not strict either, but in these cases VSYNC is usually one line long (VSW=1). Now, to find VBP and VFP, it is important to not that the goal is to meet the Vertical Blank period, so the VBK period can be split into two parts. The first part is for the VFP, before VSYNC is asserted and the other part is for the VBP, including VSYNC. The sum of these values must be equal to the VBK period. Here any division works, but, leave an imaginary VSYNC in the middle of the Blank Period. This means that both VBP and VFP should be equal or almost equal.

Using this example and considering that VBK is 20 lines (typical), VBP+VSYNC could be 10 lines (and equal to VFP). Based on the information described above, vertical timing table is created as shown in Table 5.

Table 5. Vertical Timing Table

Parameter	Symbol	Min	Type	Max	Unit
Screen Height or Vertical cycle	VP	490	500	520	Line
VSYNC Pulse Width	VSW	1	1	1	Line
Vertical Back Porch+VSYNC	VBP	1	10	40	Line
Vertical Front Porch	VFP	0	10	39	Line
Vertical Blank	VBK	10	20	40	Line

Table 5. Vertical Timing Table (continued)

Parameter	Symbol	Min	Type	Max	Unit
Active Frame Height	VDISP	480	480	480	Line
Vertical refresh rate	FV	55	60	65	Hz

2.4.2.2 Horizontal Timing

Horizontal Timing can be categorized as VGA horizontal timing and WVGA horizontal timing. These horizontal timing are discussed in the following sections.

3.4.2.2.1 VGA Horizontal Timing

Figure 10 shows an example of line period datasheet chart.

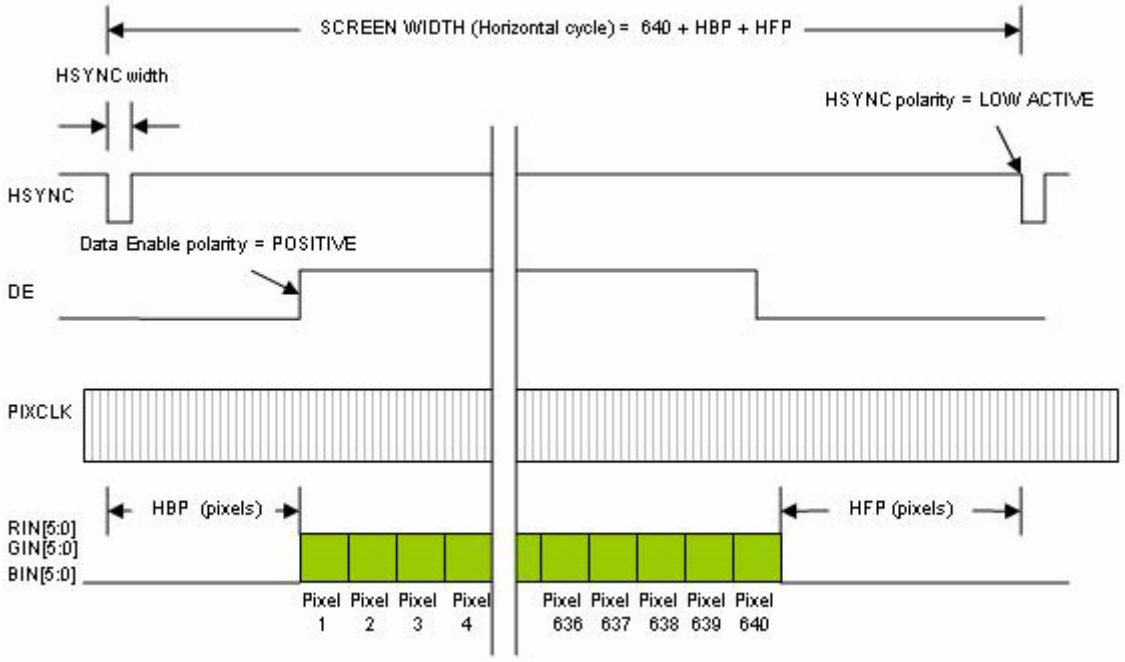

Figure 10. VGA Horizontal Timing Example

The line cycle begins when HSYNC is asserted, in this case when the signal goes low, then the HBP stage continues. During this time, the Data Enable signal remains inactive. After that the horizontal active area (ACTIVE FRAME WIDTH) begins. This stage starts when data enable is asserted. Because in this case, DE is active high, it starts when DE signal goes high. While DE is active, the panel latches the RGB data placed on the bus and draw a new pixel on the screen -in the current line-, for every pixel clock pulse. Data enable width is always equal to the horizontal resolution of the panel. In this example DE is 640 pixels long. After the active area, the HFP occurs, by this time, DE is inactive again and all the pixels in the line have been drawn. The line cycle ends when the new HSYNC pulse is asserted.

Similar to vertical timing, Table 6 shows horizontal timing characteristics (PIXCLK refers to pixel clock pulses).

Table 6. Horizontal Timing Characteristics

Parameter	Symbol	Min	Type	Max	Unit
Screen Width or Horizontal cycle	HP	750	800	900	PIXCLK
HSYNC Pulse width	HSW	1	1	1	PIXCLK
Horizontal Back Porch+HSYNC	HBP	46	46	46	PIXCLK
Horizontal Front Porch	HFP	64	114	214	PIXCLK
Active Frame Width	HDISP	—	640	—	PIXCLK

3.4.2.2.2 WVGA Horizontal Timing

The chart and table available for the WVGA (800H x 480V) datasheet is similar to the one shown in Figure 11.

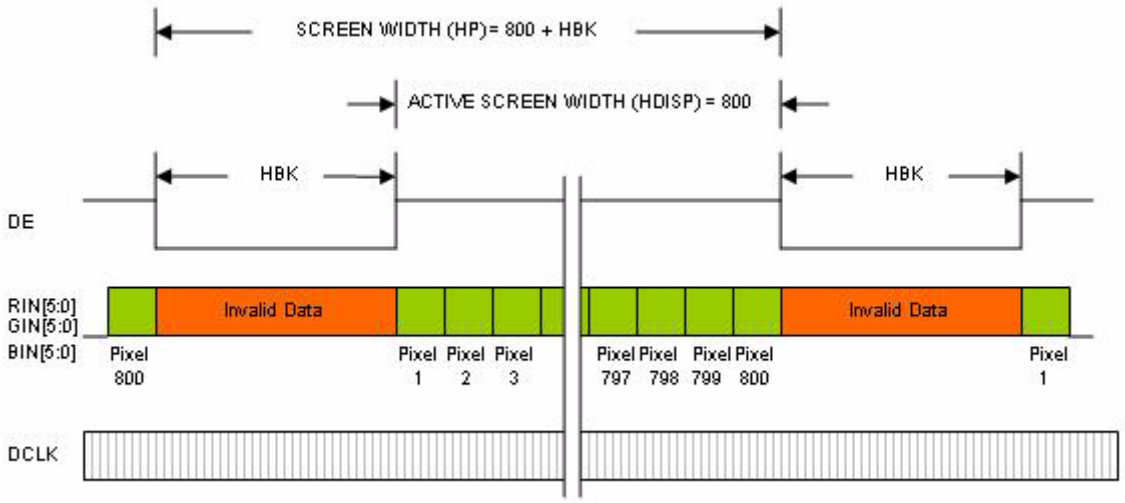

Figure 11. WVGA Horizontal Timing Example

Table 7 shows the WVGA horizontal timing.

Table 7. WVGA Horizontal Timing

Parameter	Symbol	Min	Type	Max	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 approach followed in WVGA vertical timing section should be followed here to calculate HYNC width, HBP and HFP. See Section , “3.4.2.1.2 WVGA Vertical Timing,” for information on WVGA Vertical timing.

The imaginary HSYNC signal is similar as shown in Figure 12.

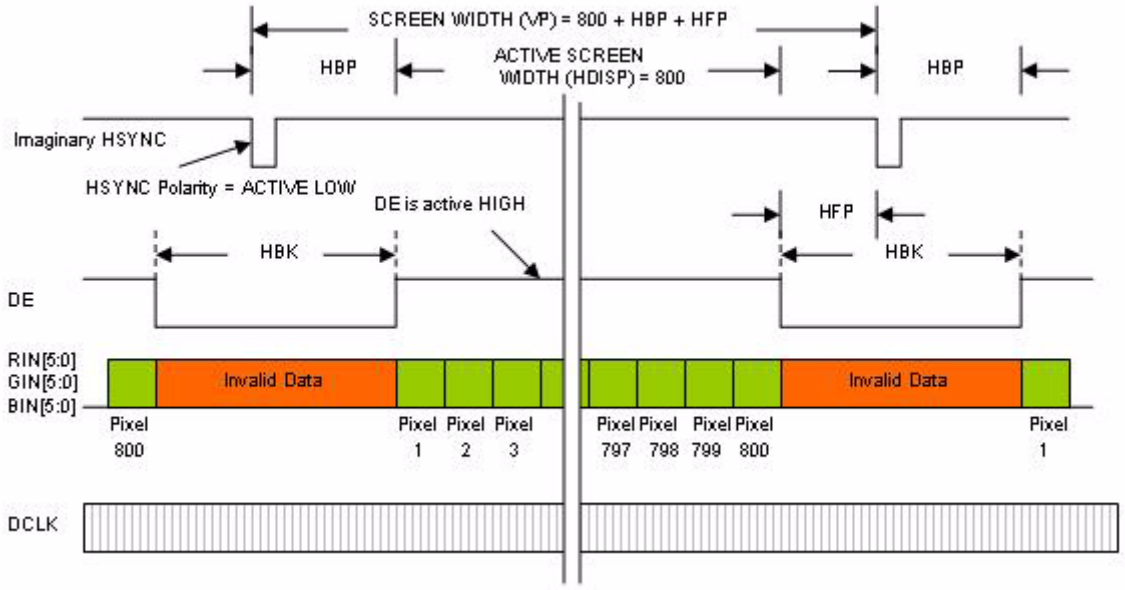

Figure 12. WVGA Horizontal Timing Example with Imaginary HSYNC Signals

Table 8 shows the WVGA horizontal timing.

Table 8. WVGA Horizontal Timing

Parameter	Symbol	Min	Type	Max	Unit
Screen Width or Horizontal cycle	HP	850	900	950	PIXCLK
HSYNC Width	HSW	1	1	1	PIXCLK
Horizontal Back Porch+HSYNC	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

2.4.2.3 Pixel Clock Timing

The VGA, WVGA pixel clock timings and the data polarity are explained in the following sections.

3.4.2.3.1 VGA Pixel Clock Timing

Pixel clock waveform characteristics in the datasheet is similar to the one shown in Figure 13.

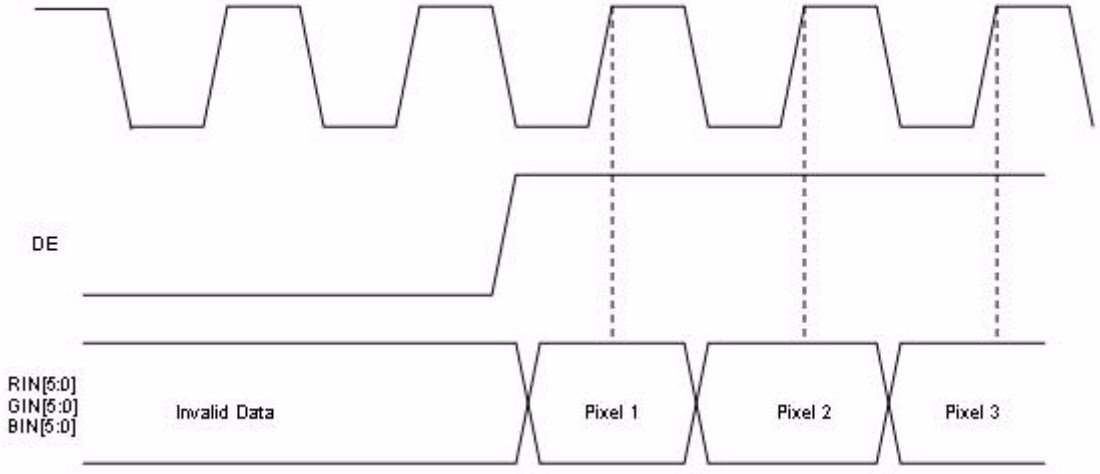

Figure 13. VGA Pixel Clock Timing Example

Table 9 shows the pixel clock frequency parameter.

Table 9. Pixel Clock Frequency Parameter

Parameter	Symbol	Min	Type	Max	Unit
Pixel Clock Frequency	PCLK	23	25	30	MHz

Pixel clock frequency is directly related with the frame refresh rate. Also, it is important as it determines when RGB data is latched by the panel. This is significant because i.MX35 prepares the data one edge before the LCD panel latches the data from the bus. A similar chart is usually included in the datasheet. In this case, data is latched by the LCD panel on DCLK rising edges, so i.MX35 should be configured to write the RGB data to the bus on the (previous) falling edge. In this manner, the data is ready and stable when the LCD panel reads it. This waveform in Figure 13 shows the typical inverse clock polarity. Clock polarity is set in the DI_DISP_SIG_POL i.MX35 register, under the D3_CLK_POL bit-field.

NOTE

The maximum display clock rate cannot be greater than a quarter of the high speed processing clock rate.

The HSP_CLK in the i.MX35 PDK BSP is 133 MHz, so the maximum pixel clock is 133 MHz / 4 = 33.25 MHz. However, most LCD displays can work with lower frequencies than the typical values.

3.4.2.3.2 WVGA Pixel Clock Timing

The pixel clock chart is similar to the one shown in Figure 14.

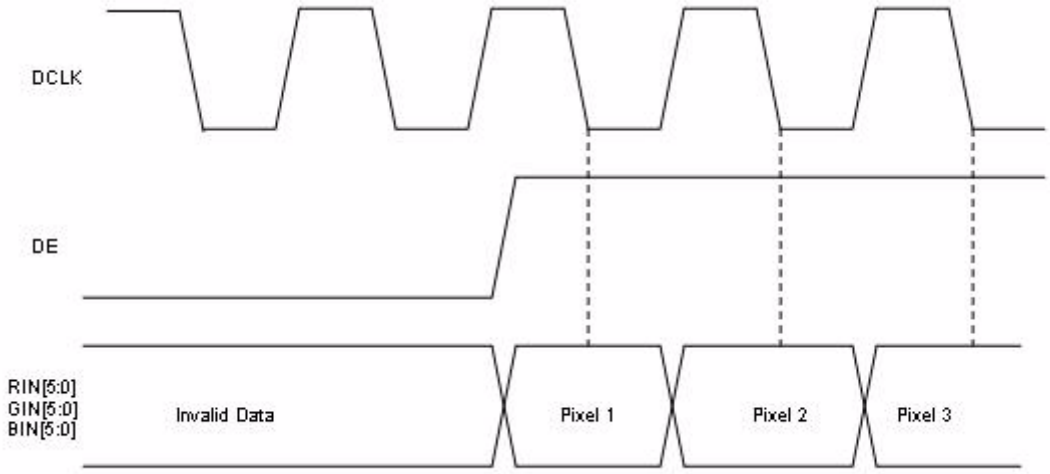

Figure 14. WVGA Pixel Clock Timing Example

Table 10 shows Pixel Clock Frequency Parameter.

Table 10. Pixel Clock Frequency Parameter

Parameter	Symbol	Min	Type	Max	Unit
Pixel Clock Frequency	PCLK	25	27	32	MHz

In contrast to the VGA panel, the WVGA display latches RGB data on DCLK falling edges, so i.MX35 should be configured to write the RGB data to the bus on the rising edge. The data is ready and stable when the panel reads it. The waveform in Figure 14 shows the straight clock polarity.

3.4.2.3.3 Data Polarity

The Data Polarity feature is the value of the active signals in the RGB bus which the LCD recognizes. For example, consider that i.MX35 is trying to draw a red pixel (only red component), using an RGB565 interface and the LCD uses straight polarity of the value in the bus would be 0xF800. Then it means that all Red bits are in high and other bits are in low. However, if the LCD utilizes inverse Data Polarity, the value would be 0x07FF, which means Red bits are in low and other bits are in high. Both values represent the red color, and the difference in the value is caused by the data polarity on the LCD panel. This feature is configured using the D3_DATA_POL bit-field on the DI_DISP_SIG_POL i.MX35 register.

2.4.3 Custom LCD Timing

The examples given in this application note does not require extra signals for LCD functionality. But if the LCD requires a RESET signal or initialization routine through a synchronous serial interface, then a similar chart as shown in Figure 17 can be found.

2.4.3.1 Reset

The Reset and Serial Command interface are explained in the following sections.

3.4.3.1.1 Reset

Many LCD panels include an LCD controller which requires an external system reset. If the LCD mentions the signal usage, then the timing required for this pulse should be found out.

Figure 15 shows an example of Reset signal.

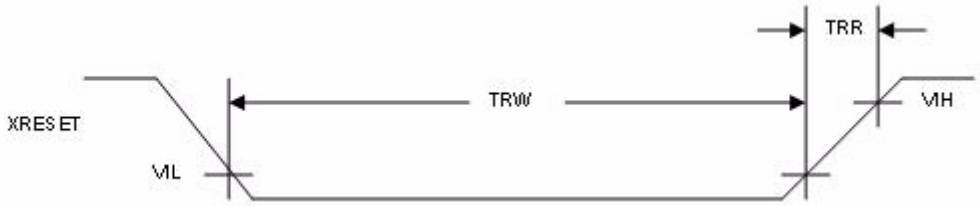

Figure 15. Reset Signal Example

Table 11 shows the Reset parameters.

Table 11. Reset Parameters

Parameter	Symbol	Min	Type	Max	Unit
Reset Width	TRW	15	—	—	ns
Reset Rising time	TRR	—	—	10	ns

Based on the above chart, the important fact that can be observed is that the RESET signal is ACTIVE LOW. It means RESET signal must be in high during normal operation, and also that it must be LOW for at least 15 ns to ensure that it was a valid reset. This waveform restricts the rising time of the signal to 10 ns. For this reason, it is recommended not to use a RC circuit to provide this signal. Generally this pin would be driven by an i.MX35 GPIO.

3.4.3.1.2 Serial Command Interface

When a LCD panel has a serial command interface, then a chart similar to the one shown in Figure 16 has to be included in the datasheet.

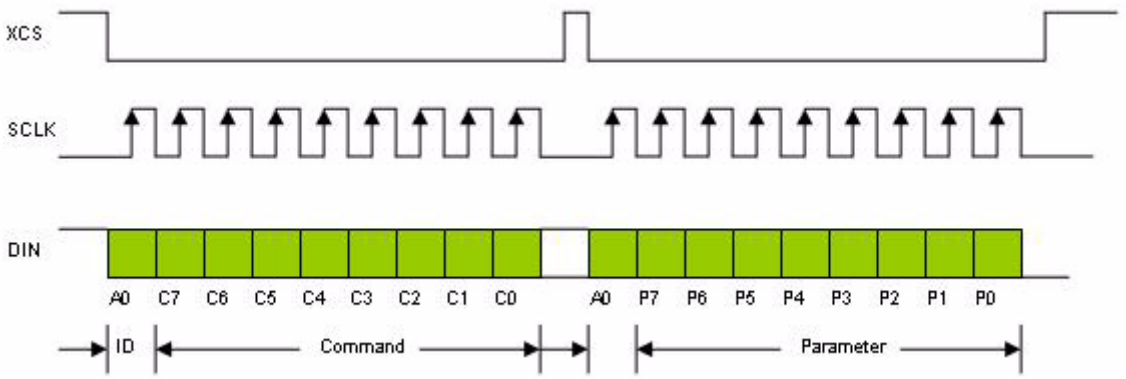

Figure 16. SPI Command Interface Signal Example

Although it is important to understand how to initialize LCD, this application note does not review all serial interfaces of the LCD.

The protocols and data format are described in the datasheet. The user should have prior knowledge in using synchronous serial interfaces to program these settings. See Chapter 16: Configurable Serial Peripheral Interface (CSPI) of the *MCIMX35 Multimedia Applications Processor Reference Manual* for more information.

2.5 LCD Panels Supported by the i.MX35

The i.MX35 processor supports up to four simultaneous displays handled by the Display controllers DISP0, DISP1, DISP2 and DISP3.

Table 12 shows the details of the display controllers.

Table 12. Display Controllers

Display	Display Type	Interface
DISP0	Asynchronous	Parallel Interface Only
DISP1	Asynchronous	Serial and Parallel Interface
DISP2	Asynchronous	Serial and Parallel Interface
DISP3	Asynchronous	RGB Interface (HSYNC, VSYNC, PIXCLK, up to RGB888)

Only one of the LCD display controllers on the i.MX35 is synchronous (dumb display). This document focusses on the DISP3 controller. Note that the DISP3 RGB interface is multiplexed with other asynchronous parallel interfaces. It means that the data can be sent to a synchronous display (DISP3) and also to another parallel display device at the same time. But, the i.MX35 sends data to the asynchronous panel (smart display) while the synchronous interface is inactive (during Horizontal and vertical back and

front porches). As a result, the frame rate of smart displays can be affected when multiple displays are attached to the i.MX35.

The synchronous LCD interface on the i.MX35 is very flexible. It can handle many types of LCD devices, as long as these devices have the following characteristics.

- Synchronous Display (Dumb Display).
- RGB interface (RGB888 maximum).
- Resolution up to SVGA.
- Utilization of at least data enable and pixel clock to latch RGB Data (some LCDs need HSYNC and VSYNC signals also, which are also supported by the i.MX35).
- Maximum pixel clock frequency of 33.25 MHz.

In addition, the i.MX35 can handle dumb displays with a SHARP interface, but its support is limited to certain models. See Section 4.7.13.2 Interface to Sharp HR-TFT Panels in *i.MX35 (MCIMX35) Multimedia Applications Processors datasheet* for information regarding the timing restrictions.

As this application note is only intended for nonsharp dumb displays, smart displays and SHARP displays interfaces are not covered.

3 Display Configuration in Linux

This section describes how to add a new panel to Linux. It also describes the general display infrastructure in Linux and analyzes the i.MX family implementation.

3.1 Linux Framebuffer

The framebuffer device provides an abstraction for the graphics driver. It represents the frame buffer of some video hardware, and allows application software to access the graphic hardware through a well-defined interface. The advantage is that the software does not need to know anything about the low-level interface.

The framebuffer is a concept related to the video controller for a graphics display. The framebuffer is a memory buffer for the video controller that contains a data frame. This data frame is shown as an information on the display. The information provided to the frame is basically, color values for each pixel.

Some of the advantages of the Framebuffer infrastructure are its ease-of-use and the user applications can access video memory directly (using `mmap`). As Framebuffer is implemented as a character device, user applications can interface with the device using common system calls such as `open()`, `read()`, `write()` and `ioctl()`. All these functions are part of the file operations interface that every character device should contain.

However, the framebuffer has the advantage of `mmap()`. This function, by definition, maps files or devices to program memory. In this case, the video buffer area is the resource that is mapped. So, the user can apply `mmap()` to get the user space memory equivalent of the hardware video frame buffer.

The result is that the user gets a pointer to the framebuffer memory and the changes made to this memory is reflected on the display.

A similar procedure can be done using `write()` and `seek()` operations. However, this procedure is time consuming, as these functions may need to be called several times to cover a particular area of the framebuffer, or the whole display area.

3.1.1 Linux Framebuffer Structures

The Framebuffer in Linux provides a set of structures (some of which are used for user-space applications) that are important elements to be taken into consideration while developing a new panel driver.

The important data structures are:

- fb_fix_screeninfo
- fb_bitfield
- fb_ops
- fb_info
- fb_videomode

These definitions are available in the file `.../include/linux/fb.h`. In the following paragraphs, these structures are briefly described. See `.../include/linux/fb.h` file for more information and definition of each structure.

struct fb_fix_screeninfo — This structure contains the fixed parameters for the graphics card/controller. One of these fixed parameters is the start of the framebuffer memory (unsigned long smem_start). This structure can be used in user applications. Other important elements are:

```
__u32 smem_len: Length of the framebuffer memory.
__u32 type: Pixel format
```

struct fb_var_screeninfo — This structure contains the parameters for the graphics card/controller that can be modified. These are the features that the user configures such as resolution, number of bits per pixel (__u32 bits_per_pixel). It also contains a structure that defines the length and bit offset for each color (struct fb_bitfield). This structure can be used in user applications. Other important elements are:

```
__u32 xres: Visible resolution in x
__u32 yres: Visible resolution in y
__u32 xoffset: Offset from virtual to visible in x axis.
__u32 yoffset: Offset from virtual to visible in y axis.
struct fb_bitfield <color>: Color bitfields for red, green, blue and
transparency (four of these are declared).
```

struct fb_bitfield — This structures contain the details of each color in a pixel. The fields are the offset (beginning of a bitfield __u32 offset) the length (__u32 length) and the most significant bit flag. One of these structures is declared for each color (red, green, blue and transparency) inside fb_var_screeninfo.

```
struct fb_bitfield <color> {
__u32 offset; /* beginning of bitfield*/
__u32 length;/* length of bitfield*/
__u32 msb_right;/* != 0 : Most significant bit is */
```

```
/* right */
};
```

For example, in a configuration where the mode is RGB888, the pixel width is 3 bytes. This means:

```
red.legth= 8 → red.offset= 24
green.length= 8 → green.offset = 16
blue.length= 8 → blue.offset = 8
```

struct fb_ops — This structure contains function pointers to framebuffer operations. The operations range from basic or common functions for a driver, such as open and release, to functions oriented to the parameter settings or ioctl calls. Some of the functions are:

int (*fb_open)(struct fb_info *info, int user)

The operation for opening a fb device that is passed as an argument to a fb_info structure pointer.

int (*fb_set_par)(struct fb_info *info)

The operation for setting the video mode and other parameters, according to the content of the var (fb_var_screeninfo) element from the fb_info structure pointer that is passed as an argument.

int (*fb_blank)(int blank, struct fb_info *info)

The operation to blank the display, passed as an argument to an fb_info structure pointer.

int (*fb_ioctl)(struct fb_info *info, unsigned int cmd,unsigned long arg)

The function for input output operations such as performing request of structure values, or configuring structure values passed as an argument to a fb_info structure pointer and a command to perform.

int (*fb_mmap)(struct fb_info *info, struct vm_area_struct *vma)

The operation for executing mmap instruction passed as an argument to a fb_info structure pointer and a pointer to a virtual memory area struct.

struct fb_videomode — This structure is used when the user wants to add support for a new panel. This structure gives specific information about the new panel such as name, resolution, pixel clock, timings for synchronization, margins and other variables that were described in previous sections, regarding LCD timings. The structure filling is described in the panel sections.

```
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 — This is an important structure in the framebuffer framework. It is the place where all the previously mentioned structures are declared. Other structures and elements are also included, such as pointers to a device, to an event queue, or current display device specifications. It also contains structures that are enabled depending on conditional building. For example, when the support for backlight is enabled.

When a framebuffer driver gets registered with the kernel, it makes use of a pointer to this type of structure, containing the information (in several different structures) for the current specific hardware panel. This structure is visible only to the kernel.

```
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 */
…
};
```

Framebuffer implementation is available in the following file:

`…/drivers/video/fbmem.c`

The following section, see Section 3.2, “Linux Framebuffer for i.MX,” describes this file and all the i.MX framebuffer implementation sources. It also describes the important functions and structures and a general flow chart for the initialization process.

Summarizing the framebuffer framework:

- For using the framework in user-space, the framebuffer device is like a `/dev/mem (/dev/fb* char device)`, so file operations used for any character devices can be used (open, read, write, mmap).
- A good example of a framebuffer driver is the virtual framebuffer that is located at `…/drivers/video/vfb.c`. This implementation follows some actions that should be followed for the development of any framebuffer driver like: filling the fix and var structures contained in the fb_info structure targeted to the current panel; filling the file operations structure and then the driver information for the fb_info structure. The following step is to initialize hardware and the memory area, and finally the registration of the framebuffer driver using a pointer to the current fb_info structure

3.2 Linux Framebuffer for i.MX

The LCD driver and the framebuffer implementation for the i.MX family requires the framebuffer framework. This framework is used to have hardware access ability and to create an abstraction layer for the software, so it does not need to know about the low level.

Once the driver is loaded (if it was marked on the kernel configuration screen, under device drivers >> graphics support), the hardware can be accessed using special nodes (like any other character device) that are located in the `/dev` directory of the root. As mentioned earlier, the nodes are available in `/dev/fb*`.

The usage of the nodes `(/dev/fb*)` as an access mechanism allows some ioctls to interact, set, or get information from the device. Some of the ioctl functions are as follows:

- Request information such as name, organization, addresses, length.
- Request and change variable information about the hardware such as geometry, depth, color or timing.
- Get and set parts.

The following section describes the i.MX framebuffer implementation that interacts with the generic framebuffer driver. A general description of the initialization process is also provided and a flow chart that shows the different stages of the framebuffer implementation is also provided.

3.2.1 Initialization Process

As any other driver initialization process, the framebuffer contains several sections. In each section, specific functions to initialize resources and probe or testing hardware are used. All files that are important part of the framebuffer startup should initialize and pass through the binding process. Therefore, functions like init and probe are described constantly, because these functions consume most of the time during the fb startup.

Kernel Start up (mx35_3Stack.c)

The initialization process of the framebuffer starts early, when the kernel calls the functions associated with the configuration of the board (in this case, the i.MX35 3-Stack Board). This function, called mxc_board_init, is basically the initialization of the important subsystems on the board. Among these systems, the framebuffer is started by registering a platform device containing some data relevant to the driver such as name, dma mask, and so on `(mxc_init_fb)`.

Framebuffer Initialization (fbmem.c)

This is the initialization of the framebuffer driver but it is not attached to any specific platform. By default, it starts the resources and important structures (most of the structures are described in the previous section) using the fbmem_init function (it registers the fb char device).

IPU initialization (ipu_common.c)

The next step is setting up the IPU module from the IPU generic parameters, to its registration as a device on the system, and the initial configuration of

processes and associated modules, such as Synchronous Display Controller (SDC).

i.MX Framebuffer Initialization (mxcfb.c)

After the IPU gets registered as a device, initialized, and probed, the next layer of software related to the implementation of the frame buffer for the i.MX family is initialized and tested with its components. The framebuffer carries out the initialization and other activities, and calls the IPU functions. The initialization process covers the registration of the framebuffer for i.MX as a platform driver. Most of the process is covered by the probe function, where (among other resources settings) the framebuffer gets registered. The probe function for the framebuffer, in general, executes tests related to the IPU such as initializing modes for the SDC module, tests for IPU_IRQ, enabling channels, disabling channels, and registering the framebuffer.

Specific Panel Driver (mxcfb_claa_wvga.c, mxcfb_epson_vga or similar)

The driver for the i.MX platform can be termed as a generic driver that can work for several panels (see mxcfb_modedb.c file for information on the available panel configurations). However, a new panel requires a specific driver, as it can have a different interface. In such cases, the driver is similar to the mxcfb.c; but, is more specific. The usage of the initialization and probe functions are also similar. This driver does not replace the mxcfb.c, but it is complementary to it. So, to enable proper functionality of the panel, both files should be built into the system.

Video 4 Linux Initialization (mxc_v4l2_output.c)

All elements to set up the framebuffer and to get the display working have been set. If video usage is required, then an extra step is needed. The BSP contains generic drivers that follow the v4l2 specification, for capture and output. These drivers are loaded after all the framebuffer and IPU setup is complete. The output driver makes use of IPU post-processing functions for its operation. It also contains an initialization routine and a probe routine.

3.2.2 Initialization Flow Chart

Figure 17 shows the initialization flow chart.

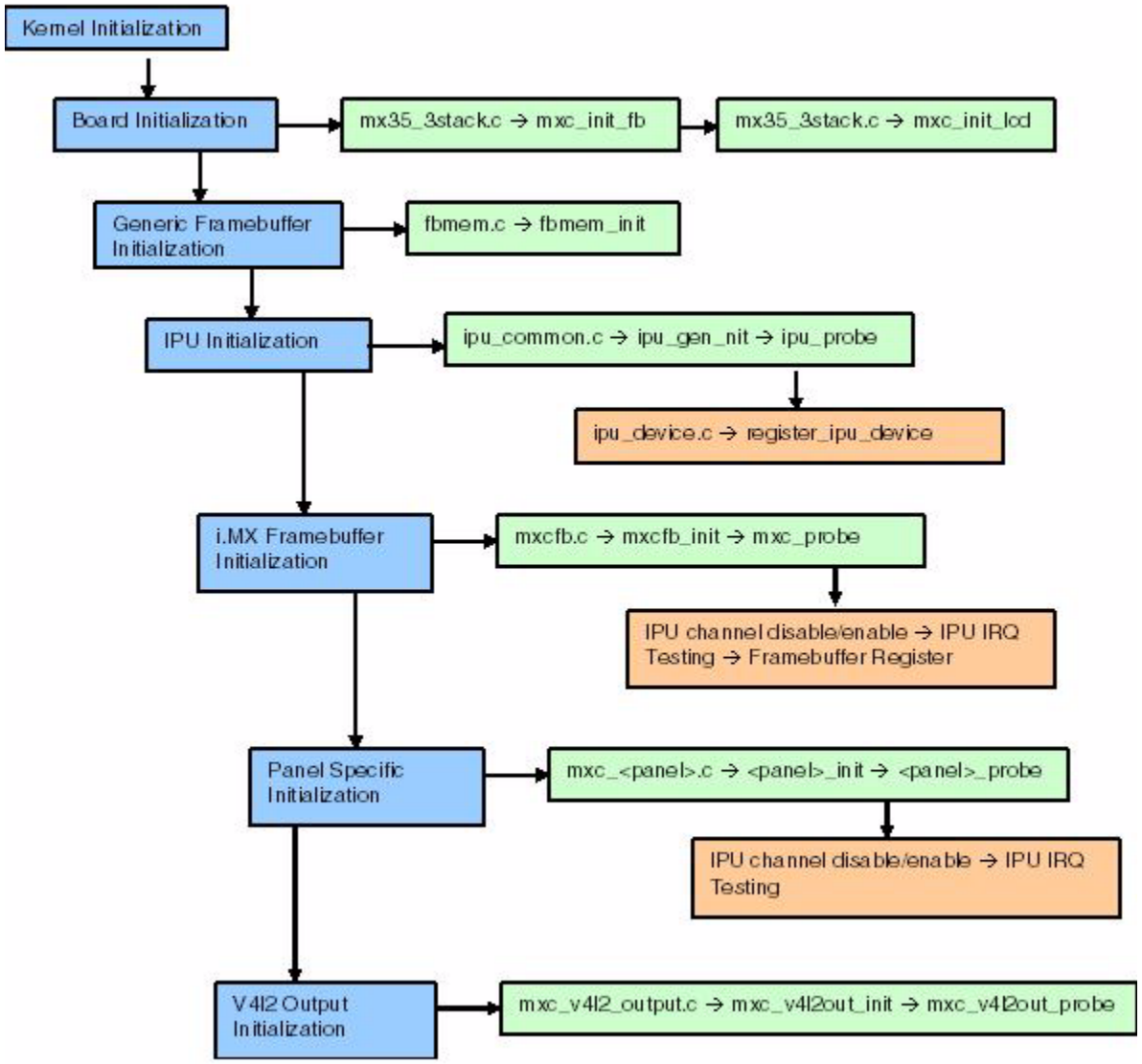

Figure 17. Initialization Flow Chart

3.2.3 Files and Important elements

This section provides more details on the files mentioned earlier in Section 3.2, “Linux Framebuffer for i.MX,” and other files.

3.2.3.1 mx35_3stack.c

Location: `/arch/arm/mach-mx35/mx35_3stack.c`

Generalities: This file contains the initialization and set up routines for the i.MX35 3-Stack Board. It is called during kernel start up. The most important function is mxc_board_init, as it calls several different module specific routines, mxc_init_fb among them, which registers the framebuffer as a platform device. Not a lot of information is provided. It gives a name, DMA mask, and some platform data (a char with the type of panel). The mxc_board_init sets the clocks, GPIO modules, the PMIC, and so on. Also, there is a small initialization of a LCD structure in the function mxc_init_lcd. This registers a platform device structure for the LCD.

```
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);
}
#if defined(CONFIG_FB_MXC_SYNC_PANEL) || defined(CONFIG_FB_MXC_SYNC_PANEL_MODULE)
/* mxc lcd driver */
static struct platform_device mxc_fb_device = {
.name = "mxc_sdc_fb",
.id = 0,
.dev = {
         .release = mxc_nop_release,
         .coherent_dma_mask = 0xFFFFFFFF,
         },
};
static void mxc_init_fb(void)
{
(void)platform_device_register(&mxc_fb_device);
}
#else
static inline void mxc_init_fb(void)
{
}
#endif
```

mx35_3stack Flow Chart:

Figure 18 shows the mx35_3stack flow chart.

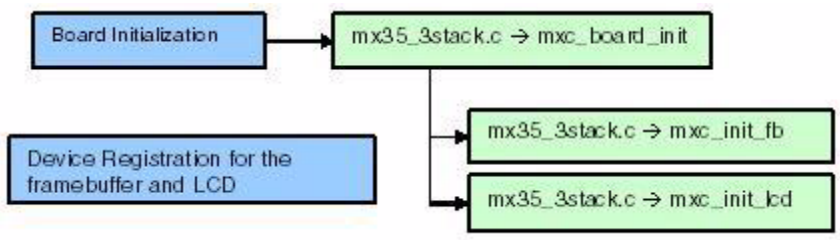

Figure 18. mx35_3stack Flow Chart

3.2.3.2 fbmem.c

Location: `/drivers/video/fbmem.c`

Generalities: This file contains initialization of the framebuffer subsystem for Linux. Apart from the initialization, it also includes the file operations (fops) common for a char device plus the usage of the `mmap()` function, which is widely used in this case (and not always used in other char devices). There are also general routines for the panel framebuffer drivers like register and unregister framebuffer devices. Functions that display the logo during the booting process is also available in this file.

Important Functions or Structures: The module definition and registering functions are explained below**:**

struct fb_info *registered_fb[FB_MAX] __read_mostly

This can be seen as the global fb_info structure for the framebuffer system.

int num_registered_fb __read_mostly

int is used to hold the frambuffers registered in the system.

static int __init fbmem_init(void)

Initialization of the framebuffer subsystem as a char driver, passing the fops table as a parameter (with the pointers to each one of the file operation functions). It also creates a class for graphics.

static void __exit fbmem_exit(void)R

Releases the char driver and deletes the graphics class.

int register_framebuffer(struct fb_info *fb_info)

Registers a framebuffer device taking the fb info structure as an argument.

int unregister_framebuffer(struct fb_info *fb_info)

Unregisters a framebuffer device taking the fb info structure as an argument.

file_operations fb_fops — This structure contains the file operations for the frame buffer subsystem implemented as a char driver. The common read, write, open and ioctl are available, but mmap is also an important member.

```
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)

A regular ioctl function for a character device, where the command to execute is passed as an argument and acts depending on the case selected. Some of the cases are: FBIOGET_VSCREENINFO and FBIOGET_FSCREENINFO that collects information about the variable or fixed values in the fb_information structure.

static int fb_mmap(struct file *file, struct vm_area_struct * vma)

mmap implementation for the framebuffer char driver.

fbmem.c Flow Chart:

Figure 19 shows the fbmem.c flow chart.

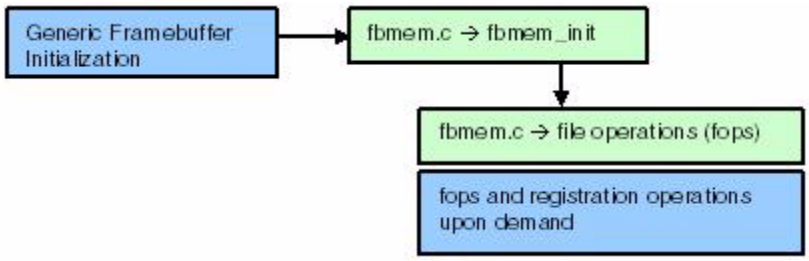

Figure 19. fbmem.c Flow Chart

Some of the IPU files are discussed in the following sections.

3.2.3.3 ipu_common.c

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

Generalities: This file contains the common software routines for IPU functionality such as channel, buffer, and IRQ management. It also contains the platform_driver structure implemented for the IPU, init, and exit functions for the module

Important Functions or Structures: The module definition and registering functions are explained below:

platform_driver mxcipu_driverThis structure contains the power management pointers to the routines for testing the platform driver and behavior in low power modes.

```
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)

This is an initialization routine for the IPU platform driver. It registers the mxcipu_driver.

static void __exit ipu_gen_uninit(void)

This exit routine is used for the IPU platform driver. It releases the IPU IRQs and unregisters the mxcipu_driver.

static int ipu_probe(struct platform_device *pdev)

This is Probe function for the mxcipu_driver and it is called when the registration is made. It sets IRQ request and clocks for the IPU. It also registers the IPU device (see ipu_device.c).

Common IPU functions: The common IPU functions are explained below:

ipu_request_irq	This function registers an interrupt handler for the specified interrupt lines that are defined in the enumeration ipu_irq_line (located in the ipu.h file).
ipu_disable_irq	This function disables the interrupt for the specified interrupt line.
ipu_enable_irq	This function enables the interrupt for the specified interrupt line.
ipu_init_channel_buffer	This function is called to initialize a buffer for a logical IPU channel. The parameters entered as inputs are physical addresses for the buffers, type of the buffers, logical channel ID, width and height in pixels and so on.
ipu_select_buffer	This function is used to mark a channel's buffers as ready. The channel ID and buffer type are passed as parameters.
ipu_init_channel	This function is called to initialize a logical IPU channel. It uses a logical channel ID and a union with channel initialization parameters (ipu_channel_params_t included in ipu.h).
ipu_uninit_channel	This function is used to uninitialize a logical IPU channel.
ipu_link_channels	This function links 2 channels together for automatic frame synchronization. There are 2 parameters, one is the logical channel ID for the source and the other is the logical channel ID for the destination, the output of the source channel is linked to the input of the destination channel.
ipu_unlink_channels	This function unlinks two channels and disables the automatic frame synchronization.

ipu_enable_channel This function enables a logical channel taking the channel ID as input.

ipu_disable_channel This function disables a logical channel taking the channel ID as input.

ipu_common.c Flow Chart:

Figure 20 shows the ipu_common.c flow chart.

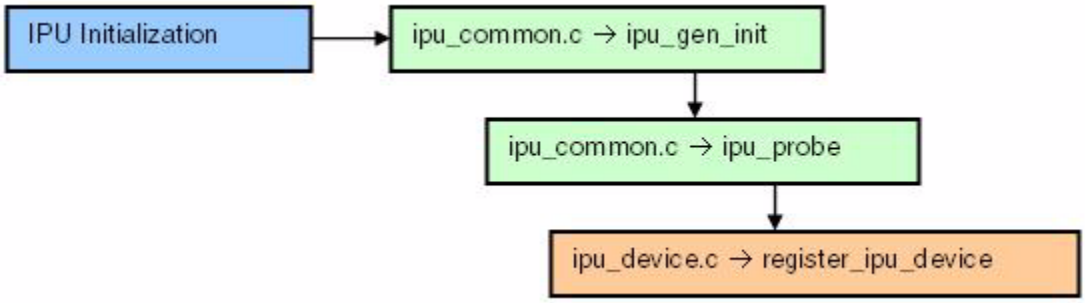

Figure 20. ipu_common.c Flow Chart

NOTE

Many IPU functions that are widely called from framebuffer and v4l2 drivers (such as ipu_init_channel) are defined in ipu_common.c.

3.2.3.4 ipu_device.c

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

Generalities: This file contains the structure and functions for the fops operations related to the mxc_ipu device. It also contains a generic interrupt handler for the IPU related IRQs

Important Functions or Structures: The registering and other important functions areexplained below:

int register_ipu_device() This function registers the mxc_ipu as a char device, providing fops table and name. It also creates a class in the device model structure. This function is called from the `probe()` function in `ipu_common.c`

static irqreturn_t mxc_ipu_generic_handler(int irq, void *dev_id)

This function is a generic handler for any IRQ that the IPU should process.

file_operations mxc_ipu_fopsThis structure contains the file operations for the mxc_ipu device.

```
static struct file_operations mxc_ipu_fops = {
    .owner = THIS_MODULE,
    .open = mxc_ipu_open,
    .mmap = mxc_ipu_mmap,
    .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)

ioctl function for the mxc_ipu device. This function Processes commands passed as arguments. Most commands are specific functions described

earlier in the section ipu_common.c such as IPU_INIT_CHANNEL, IPU_LINK_CHANNELSand so on.

ipu_device.c Flow Chart:

Figure 21 shows the ipu_device.c flow chart.

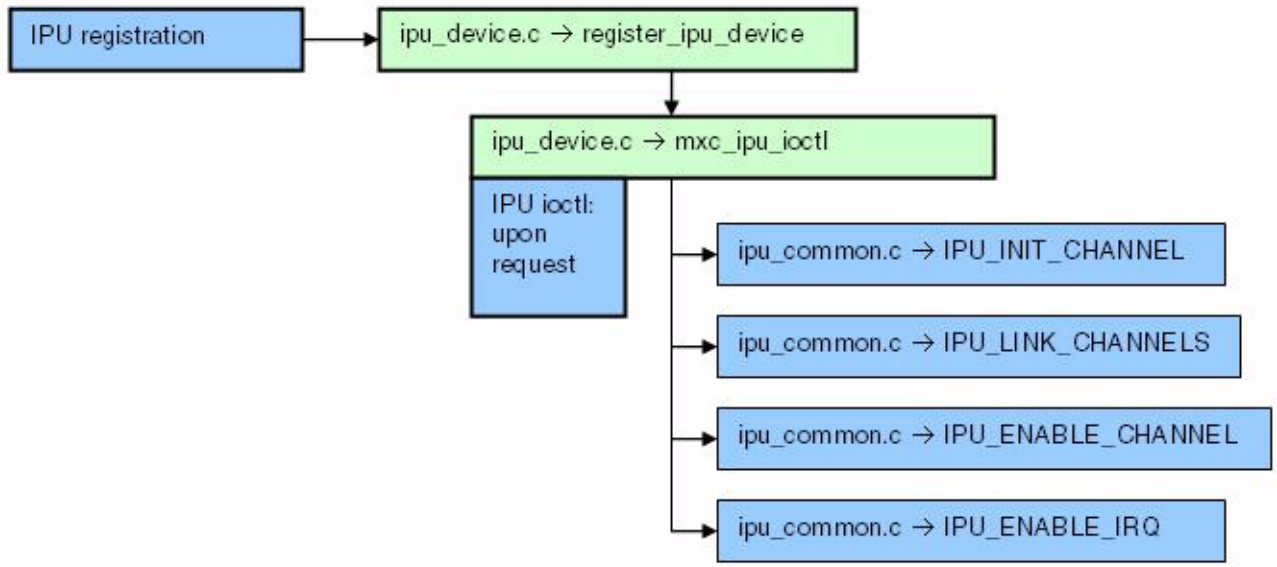

Figure 21. ipu_device.c Flow Chart

Other important files for the IPU structure are explained below:

3.2.3.5 ipu_sdc.c

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

Generalities: This file contains routines related to the SDC module on the IPU. The examples range from the sdc_init to routines for setting alpha blending modes or color keys for the SDC plane.

3.2.3.6 ipu_ic.c

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

Generalities: This file contains routines for color conversion and resizing the sub modules of the IPU.

3.2.3.7 mxcfb_modedb.c

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

Generalities: This file contains the declaration of an array of fb_videomode structures related to the mxc framebuffer implementation. The elements are Sharp-VGA, NEC-VGA, TV out modes and so on. The data available on each structure refers to the important elements that describes each panel such as

resolution, timing and so on. See include/linux/fb.h for more information for the parameters contained in the structure.

3.2.3.8 mxcfb.c

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

Generalities: This driver contains registering and initialization routines for the implementation of the framebuffer oriented to the i.MX Family. The structure related to the framebuffer, functions for registering the driver and setting up the frame buffer system for the normal framebuffer data structures for the overlay are initialized if they are enabled.

Important Functions or Structures: The module definition and registering functions are explained below:

platform_driver mxcfb_driver The Structure created for the framebuffer driver implementation contains the pointers to functions related to power management callbacks such as probe or suspend.

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

mxcfb_data The Structure that encapsulates two fb_info structures. The first structure acts for the normal framebuffer and the second one for the overlay. This structure also contains other flags.

```
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) The entry function for the framebuffer. It registers the platform driver structure containing callback functions for power management and shutdown conditions.

void mxcfb_exit(void) This is an framebuffer exit function. Its functions are: unmaps the video memory of the framebuffer structures, unregisters the fb_info structure of the framebuffer. These activities are carried by the fb_info structure for the framebuffer and the overlay. It also unregisters the platform driver structure of the framebuffer.

static int mxcfb_probe(struct platform_device *pdev)

This is the function member of the platform driver structure pointers. The probe function should verify if the specified device hardware exists. Several

processes are executed in the probe function such as framebuffer initialization, memory allocation, framebuffer registration (fb_info structures for normal and overlay structures) and also IPU initialization that involves setting the transparent color key for SDC graphic plane and the foreground/background alpha blending modes.

static struct fb_info *mxcfb_init_fbinfo(struct device *dev, struct fb_ops *ops)

This function is called by mxc_probe. It allocates memory for the fb_info structure and fills information such as color maps and so on in the fields that are related to this structure.

platform_set_drvdata(pdev, &mxcfb_drv_data)

This function is used to pass the address and information available in the mxcfb_drv_data structure to the platform device.

Some of the file operations are discussed below.

fb_ops mxcfb_ops — This structure contains the pointers to the functions that can be used by the framebuffer driver to perform functions such as rectangle filling or cursor definitions. This structure is used for the normal framebuffer implementation.

```
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 — This structure contains the pointers to the functions that can be used by the framebuffer driver to perform functions such as rectangle filling or cursor definitions. This structure is used for the overlay framebuffer implementation.

```
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)

This function used to handle the mmap function for the mxc framebuffer.

static int mxcfb_ioctl_ovl(struct fb_info *fbi, unsigned int cmd, unsigned long arg)

static int mxcfb_ioctl(struct fb_info *fbi, unsigned int cmd, unsigned long arg)

These functions are used to handle ioctl commands for the framebuffer. The first function is used for the normal framebuffer structure and the other is for the overlay structure.

static int mxcfb_set_par(struct fb_info *fbi)

This function is used to set the parameters (most of them from the videomode structure) to the processor's registers and calls ipu_sdc_init_panel. This is also used to change the operating mode.

mxcfb.c Flow Chart:

Figure 22 shows the mxcfb.c flow chart.

Figure 22. mxcfb.c Flow Chart

3.2.3.9 mxc_v4l2_output.c

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

Generalities: This file contains the implementation of the v4l2 standard for output devices targeted to the i.MX family. The file contains the common char driver infrastructure elements such as an initialization routine for registering the driver, a probe function and the standard file operations (fops) related to the v4l2 implementation. It also includes the functions that enable or disable the playback of video.

Important Functions or Structures: The module definition and registering functions are explained below:

platform_driver mxc_v4l2out_driver

This is the platform_driver structure for the v4l2 output driver containing pointers to functions related to power management such as probe and remove.

```
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

This is the platform_device structure for the v4l2 output driver containing the name and id of the device.

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

static int mxc_v4l2out_init(void)

This is used for initializing the driver where the registration of the platform driver and platform device is made.

static void mxc_v4l2out_clean(void)

This is an exit function from the driver where the platform device and platform driver are unregistered. A video device is also unregistered.

static int mxc_v4l2out_probe(struct platform_device *pdev)

This is a probe function for the v4l2 driver. It contains the register of the video device with video_register_device, setup of the outputs and cropping commands.

The Fops operations are discussed below.

file_operations mxc_v4l2out_fops

This is the file operations structure for the mxc_v4l2_output. It contains the pointers to common functions such as open, close, ioctl. It also contains the mmap implementation for this driver.

```
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)

This function is used for mmap implementation for the v4l2 driver and it is part of the file operations.

static int mxc_v4l2out_do_ioctl(struct inode *inode, struct file *file, unsigned int ioctlnr, void *arg)

This function is called by video_usercopy that in turn is called by mxc_v4l2out_ioctl. This function performs the commands sent by the application when trying to perform some specific ioctls.

Other operations are as follows:

static int mxc_v4l2out_streamon(vout_data * vout)

This function is used to start the playback to the framebuffer/display. It uses many calls and modifications directly to the IPU. These calls refer to the usage of the IPU channels such as selecting buffers, initializing channels, enabling IRQs and so on.

static int mxc_v4l2out_streamoff(vout_data * vout)

This function is used to stop the playback. It uses the implemented functions in the lower layer of the IPU configuration such as uninitializing and unlinking channels and disabling IPU IRQs.

mxc_v4l2_output.c Flow Chart:

Figure 23 shows the mxc_v412 output.c flow chart.

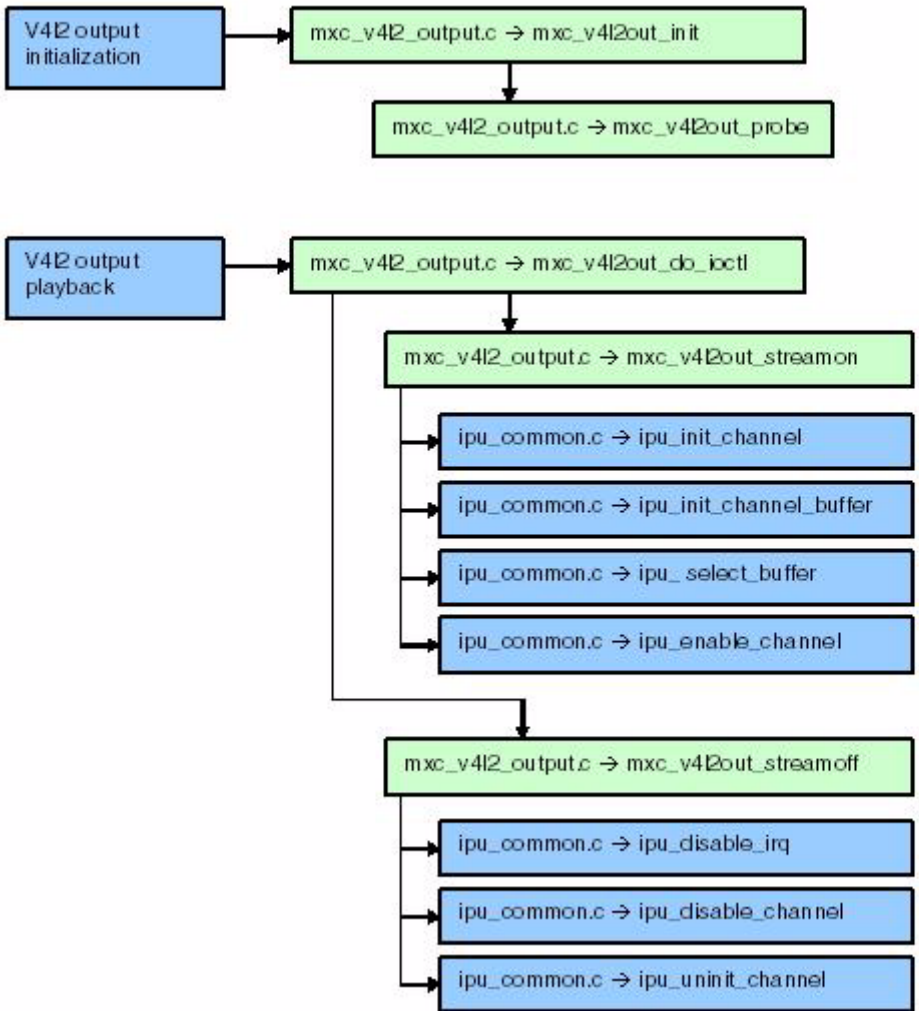

Figure 23. mxc_v412 output.c Flow Chart

3.3 Panel Configurations

Implementation of a new driver into the Linux kernel is important for a system with a new LCD panel, which does not have a driver. To enhance the performance of the application, hardware such as i.MX's IPU can be used.

3.3.1 Case: VGA

The LCD panel configurations are explained in the following sections.

3.3.1.1 Panel Generalities

CLAA057VA01CT is 5.7" color TFT-LCD (Thin Film Transistor Liquid Crystal Display) module composed of LCD panel, driver ICs, control circuit, and LED backlight.

The 5.7"screen produces a high resolution image that is composed of 640×480 pixel elements in a stripe arrangement. It can display 262K colors by means of a 6 bit per channel RGB signal input.

Horizontal Timing Parameters

Table 13 provides horizontal timing parameters.

Table 13. Horizontal Timing Parameters

Parameter	Symbol	Min	Type	Max	Unit
Screen Width or Horizontal Cycle	HP	750	800	900	PIXCLK
Horizontal Blank Period	HBK	110	160	260	PIXCLK
Active Frame Width	HDISP	640	640	640	PIXCLK

Vertical Timing Parameters

Table 14 provides vertical timing parameters.

Table 14. Vertical Timing Parameters

Parameter	Symbol	Min	Type	Max	Unit
Screen Height or Vertical Cycle	VP	515	525	560	Line
Vertical Blank	VBK	35	45	80	Line
Active Frame Height	VDISP	480	480	480	Line
Vertical Refresh rate	FV	55	60	65	Hz

Pin Connections

Table 15 provides the pin connection details of LCD panel.

Table 15. LCD Panel Pin Connection

Pin Number	Symbol	Description
1	U/D	Up or Down Display Control
2	NC	Customer non-connect ; initial pull high =DE mod
3	Hsync	Horizontal SYNC
4	VLED	Power Supply for LED
5	VLED	Power Supply for LED
6	VLED	Power Supply for LED
7	Vcc	Power Supply for LCD
8	Vsync	Vertical SYNC
9	DE	Data Enable
10	X2	TSP Control (Left)
11	Y1	TSP Control (Up)
12	ADJ	Adjust for LED brightness
13	B5	Blue Data 5 (MSB)
14	B4	Blue Dat a4
15	B3	Blue Data 3
16	Vss	Power Ground
17	B2	Blue Data 2
18	B1	Blue Data 1
19	B0	Blue Data 0
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

Table 15. LCD Panel Pin Connection (continued)

Pin Number	Symbol	Description
32	Vss	Power Ground
33	R2	Red Data 2
34	R1	Red Data 1
35	R0	Red Data 0 (LSB)
36	X1	TSP Control (Right)
37	Y2	TSP Control (Down)
38	DCLK	Clock Signals
39	Vss	Power Ground
40	L/R	Left / Right Display Control

3.3.1.2 Panel Configuration

Configuring `fb_videomode` structure of a new panel is required to create a new driver. This structure contains information about the timings, resolution, name and configurations for the panel to work properly. The code below shows the parameters to be filled.

```
struct fb_videomode {
        const char *name; → = CLAA-VGA
        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
};
```

The parameters are available in the section 3.2 of *panel datasheet*. Take a quick look at the timing parameters shown in panel generalities and use the typical values (recommended). Some LCD datasheets only provide Blank Periods instead of Porches.

The following section discusses what to do in such cases. However, as this datasheet provides both Blank Periods and Porches, use the values available in the Porches section of the timing specification table.

const char *name — This is just a name, so any parameter with a descriptive name between " " suffice.

u32 refresh — This is the refresh rate. The value is given in Hz. Generally, it provides refresh rate or vertical refresh rate. In this case, 60Hz is used.

u32 xres	This is the resolution in the x axis. This value can be got easily as it is one of the most important and descriptive parameters in the LCD. It is also available as number of horizontal pixels (H).
u32 yres	This is the resolution in the y axis. It is provided in the data sheet as the vertical resolution or number of lines (V).
u32 pixclock	This is the pixel clock. It can be found in the datasheet as dot clock or just clock. The value is usually given in MHz. However, sometimes the inverse value is required so that it can be entered in the structure in picoseconds. In this case it is 25 MHz, so 1/25M= 40000 ps.
u32 left_margin	The left margin is equivalent to the Horizontal Back Porch (HBP) described. Some LCD datasheets provide this parameter. However, some provide a total that is the sum of the HBP, HFP and hsync pulse width termed as Horizontal Blank Period. When this happens, the fill this value using the procedure described in Section , “3.4.2.2.2 WVGA Horizontal Timing,” alternatively, HBP and the hsync pulse width can be used (and leave HFP at 0). The sum of both should add up to the Horizontal Blank Period (with typical value of 160 in this case). There could be some variation in the values of hsync pulse width and HBP, but both of them should add to Horizontal Blank Period (160). This value is provided in pixel clock units. In this case the value is 45.
u32 right_margin	This right margin is equivalent to the Horizontal Front Porch. In this case, if the procedure to avoid this parameter as described in the u32 left_margin description is followed, then the value would be 0 (and left margin would be around 159). However, if the data from the datasheet is used, then the typical value of 114 is used. This value is provided in pixel clock units.
u32 upper_margin	This upper margin is equivalent to the VBP. It is a case similar to the left margin. Some datasheets show the VFP, VBP and the vsync pulse width. Others provide only a value that covers the sum of the 3 periods. The name of the sum is Vertical Blank Period (in this case with a typical value of 45). Avoiding one of the porches is possible, so only the VBP and the vsync pulse would be used. The division of the value between both parameters could vary, but they must add up to the Vertical Blank Period (in that case, a value of 44 is selected). However, the typical value of 33 is used. The value is provided in horizontal lines.
u32 lower_margin	This lower margin refers to the Vertical Front Porch. As with u32 right_margin and for similar reasons (that is,. if the datasheet only provides Blank Periods), the value for this parameter can be selected as 0. In this case, the typical value of 11 is used.
u32 hsync_len	This is the hsync pulse width. Some LCD datasheets provide this value but others only provide a Horizontal Blank Period, as described above. The Horizontal Blank Period on this LCD is 160. So if the left margin is set to 159, then the value is set to 1 pixel clock pulse. Using the selected value (the typical ones from the data sheet) value 1 is set.

u32 vsync_len	This is the vsync pulse width. As in the hsync case, some datasheets provide the value, but others only provide a Vertical Blank Period. The sum of the upper margin and this value should be equal to the Blank Period (45 lines in this case). If the upper margin value is selected as 44, this parameter would have a value of 1. Using the selected values (typical) we value 1 is set.
u32 sync	This value refers to the display interface clock polarity for display 3. In this case the value is 0 (as opposed to FB_SYNC_CLK_LAT_FALL), so the clock polarity is inverse. Data is considered as valid after data enable goes HIGH, and each pixel is read on the rising edges of the clock.
u32 vmode	This value determines the video mode. The Linux kernel defines more modes such as FB_VMODE_INTERLACED or FB_VMODE_DOUBLE, among others. But, for i.MX using TFT panels, set this to FB_VMODE_NONINTERLACED.
u32 flag	This value is not used, usually left in 0.

The structure looks as shown in the following code:

```
static struct fb_videomode video_modes[] = {
{
        /* 640x480 @ 60 Hz , pixel clk @ 25MHz */
        const char *name; → "CLAA-VGA",
        u32 refresh; → 60,
        u32 xres; → 640,
        u32 yres; → 480,
        u32 pixclock; →  40000,
        u32 left_margin;→ 45,
        u32 right_margin; →  114,
        u32 upper_margin; → 33,
        u32 lower_margin; → 11,
        u32 hsync_len; → 1,
        u32 vsync_len; → 1,
        u32 sync; → 0,
        u32 vmode; → FB_VMODE_NONINTERLACED,
        u32 flag; →  0,
}
```

The fb_videomode structure for the panel is declared in the VGA panel driver. The information is passed to a fb_var_screeninfo structure inside the lcd_init_fb function (which is called by probe). In the fb.h definition, the pixclock parameter should be passed in pico seconds. The data in that var structure is passed to the general info structure via the fb_set_var function, also called in lcd_init_fb.

Finally when the function ipu_sdc_init is called, many parameters are finally set to configure at the register level (in the last stage of that process).

A particular setup is made when passing the values for the timing and porches to the registers of the i.MX processor. i.MX35 manages the porches, active dimensions and sync signals as a single parameter for each dimension. So, in the ipu_sdc_init function, these values are added and then passed to the SDC_HOR_CONF register for the horizontal case (SCREEN_WIDTH field) and SDC_VER_CONF for the vertical case (SCREEN_HEIGHT field).

HSYNC+Horizontal Back Porch and VSYNC+Vertical Back Porch are also added into g_h_start_width and g_v_start_width, respectively, and then they are written into SDC_BG_POS/SDC_FG_POS by the ipu_disp_set_window_pos function

Figure 24 shows th fb_videomode flow chart.

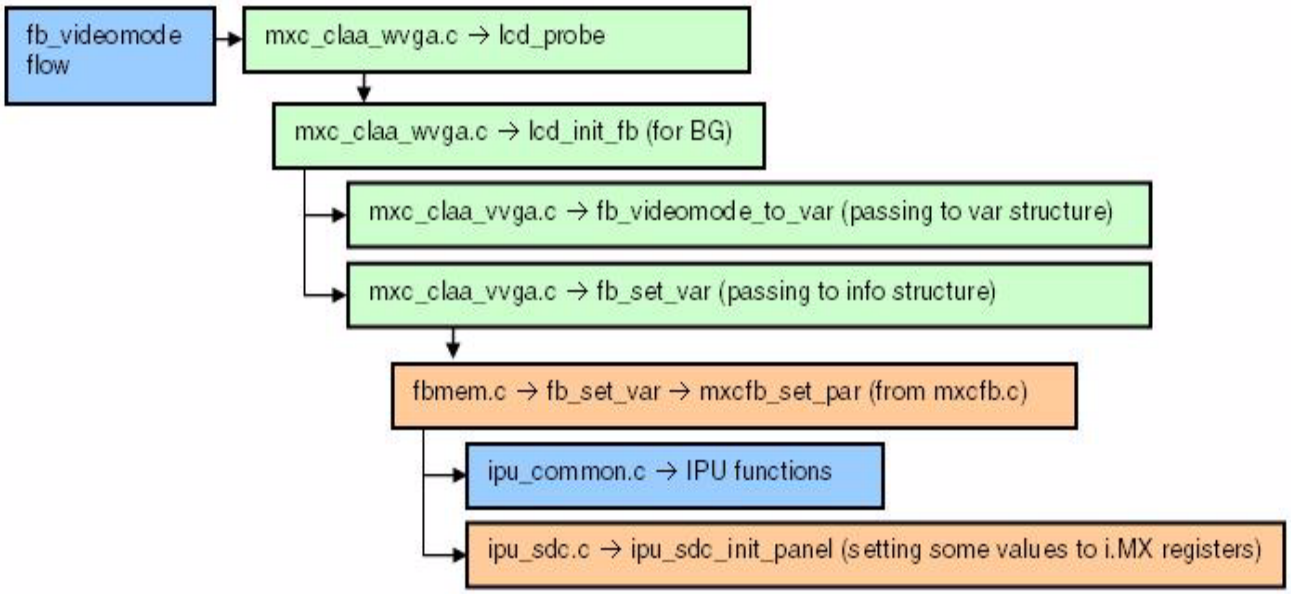

Figure 24. fb_videomode Flow Chart

3.3.1.3 Driver Development Process

This section describes how a new driver is made. Important functions and main differences that make this driver unique and where these differences should be addressed are also discussed. The general procedure and elements to be considered when creating a panel driver are also discussed.

A new panel may have differences such as interface and connection, voltage level management and so on (apart from the default difference in resolution, when that is the case). So, a new file creation to cover the specific needs of this panel may be required. Some of these changes are addressed in this file, and other changes such as pin setup and registration of the LCD panel device structure are addressed in other files.

A new panel driver is similar in terms of structure to the mxcfb.c file. In many ways, we could say that the new file could be seen as a subset of mxcfb.c. The panel driver in general terms should contain:

- An fb_videomode structure with the timing and configuration data for the panel.
- Char driver standard functions for initialization and cleanup (usually for registering/unregistering a platform driver structure).
- Platform driver structure with standard functions (probe, suspend, resume) pointers and these functions' implementations.
- If using notifier chain facility for event processing, function and structure related to this topic.
- Functions for communicating and enabling the flow of information between panel structures and framebuffer generic structures.

- Interface and voltage related functions (called by other functions) for the LCD panel power management.

Many elements that are discussed earlier can be found in the following file.

3.3.1.4 mxcfb_claa_wvga.c

Location: `drivers/video/mxc/mxcfb_claa_wvga.c`

From the above file, the following file can be created:

3.3.1.5 mxcfb_claa_vga.c

Location: drivers/video/mxc/mxcfb_claa_vga.c

Voltage levels and RGB bus width are same for the VGA CLAA057VA01CT and the WVGA CPT_CLAA070VC01 that comes with the PDK. So, minor modification is required. A hypothetical mxcfb_claa_vga.c file would be virtually identical to mxcfb_claa_wvga.c, with the exception of the fb_videomode and the FB_EVENT_BLANK case. Note that gpio_lcd_active already has entries for VSYNC and HSYNC signals, even though the 7" screen does not require them.

NOTE

5.7" connector has to be populated to connect the panel to the 3 stack board

Generalities: This file is used to register the driver and to test the probe functionality. It also provides the timings in the `fb_videomode` structure and contains some functions for interaction with the panel related to events and ON/OFF functionality.

Important Functions or Structures: The module definition and registering functions are explained below:

platform_driver lcd_driver — This Structure contains the pointers to power management and binding functions such as probe, suspend or resume targeted to this panel.

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

static int init claa_lcd_init(void)

This function registers the LCD_driver structure of type platform driver.

static void exit claa_lcd_exit(void)

This function unregisters the LCD_driver structure of platform driver type.

static int devinit lcd_probe(struct platform_device *pdev)

This function is called when a device is installed. This function takes care of setting voltage levels, calls functions to initialize the LCD, turns it on and

finally calls a function to notify the kernel that a new event happened (in this case the installation of the LCD driver).

static void lcd_init_fb(struct fb_info *info)

This function is called by LCD_probe. It fills the memory for the fb_var_screeninfo structure that contains specific details about the panel (fb_videomode structure --> resolution, size, timings), 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)This function turns on the panel.

static void lcd_poweroff(void)This function turns off the panel.

fb_register_client(&nb) This function is called from the LCD_probe function. It registers a client notifier that contains a notifier block structure. This structure contain details such as a pointer to the function called when a LCD driver event occurs.

static int lcd_fb_event(struct notifier_block *nb, unsigned long val, void *v)

This function is called when a LCD driver event occurs. It registers the event occurred.

Some critical elements for the panel that are not discussed are located in other files. These elements are related to pin configuration (for the LCD connection) and the initialization of an LCD device structure. The following section describes the LCD flow to be initialized. Also, some comment on the elements that are part of the panel driver are given.

Similar to the general framebuffer initialization, the driver for the LCD also starts at a very early point during the kernel loading process. As mentioned earlier in another section, the registration of the LCD panel as a platform device structure is made in the mx35_3stack.c file. From there, both the LCD panel and the framebuffer are registered as platform device elements.

The next stage is to cover the processor pin setup, so the LCD can have a proper connection. All these tasks are developed in the mx35_3stack_gpio.c located in the `…linux-2.6.26/arch/arm/mach-mx35/` folder. Among other features, there is a function called `gpio_lcd_active` that sets all the pins used by the panel with the help of the function mxc_request_iomux and passed as a parameter (pin by pin) the characteristics that the user wants to get from every specific signal.

```
void gpio_lcd_active(void)
{
        mxc_request_iomux(MX35_PIN_LD0, MUX_CONFIG_FUNC);
        mxc_request_iomux(MX35_PIN_LD1, MUX_CONFIG_FUNC);
        mxc_request_iomux(MX35_PIN_LD2, MUX_CONFIG_FUNC);
        mxc_request_iomux(MX35_PIN_LD3, MUX_CONFIG_FUNC);
        mxc_request_iomux(MX35_PIN_LD4, MUX_CONFIG_FUNC);
        mxc_request_iomux(MX35_PIN_LD5, MUX_CONFIG_FUNC);
        mxc_request_iomux(MX35_PIN_LD6, MUX_CONFIG_FUNC);
        mxc_request_iomux(MX35_PIN_LD7, MUX_CONFIG_FUNC);
        mxc_request_iomux(MX35_PIN_LD8, MUX_CONFIG_FUNC);
        mxc_request_iomux(MX35_PIN_LD9, MUX_CONFIG_FUNC);
        mxc_request_iomux(MX35_PIN_LD10, MUX_CONFIG_FUNC);
        mxc_request_iomux(MX35_PIN_LD11, MUX_CONFIG_FUNC);
```

```
                mxc_request_iomux(MX35_PIN_LD12, MUX_CONFIG_FUNC);
                mxc_request_iomux(MX35_PIN_LD13, MUX_CONFIG_FUNC);
                mxc_request_iomux(MX35_PIN_LD14, MUX_CONFIG_FUNC);
                mxc_request_iomux(MX35_PIN_LD15, MUX_CONFIG_FUNC);
                mxc_request_iomux(MX35_PIN_LD16, MUX_CONFIG_FUNC);
                mxc_request_iomux(MX35_PIN_LD17, MUX_CONFIG_FUNC);
                mxc_request_iomux(MX35_PIN_D3_VSYNC, MUX_CONFIG_FUNC);
                mxc_request_iomux(MX35_PIN_D3_HSYNC, MUX_CONFIG_FUNC);
                mxc_request_iomux(MX35_PIN_D3_FPSHIFT, MUX_CONFIG_FUNC);
                mxc_request_iomux(MX35_PIN_D3_DRDY, MUX_CONFIG_FUNC);
                mxc_request_iomux(MX35_PIN_CONTRAST, MUX_CONFIG_FUNC);
        }
```

Once the pins are set properly, the following stages occur from the loading of the generic framebuffer infrastructure (fbmem.c) to the loading and registering of the structures and functions related to this panel. Most of the process has already been described. First, it goes to the fbmem.c file. Then it loads the IPU module. Next the framebuffer implementation for i.MX family by registering the framebuffers (normal and overlay) begins and concludes with the panel driver.

The panel driver also has initialization routines for registering the platform driver, but most of the initialization procedure is made by the probe function. The actions that occur are: allocation of memory for the structures related to this panel, passing of the `fb_videmode` structure with this panel's data to the var_screen_info structure of the `fb_info` structure. The panel driver also has functions for turning the panel ON/OFF, for regulating voltage levels using functions from the regulation framework, and registering an LCD event for the kernel notifier chain register.

The function `lcd_init_fb` in the panel driver can be seen as the function where all the flow of information (of `fb_videomode` structure) and turning on events happen. In `fbmem.c`, a function called `fb_set_var` is a link to the `mxcfb_set_par` function in `mxcfb.c` (`mxcfb_set_par` is actually part of the file operations structure of the `fb_info` structure), that is, it's the implementation of the `fb_set_par` for the i.MX framebuffer. In `mxcfb.c`, the function, `ipu_sdc_init_panel` is called having as arguments with many elements of the `fb_videmode` structure. This function is the place were all these elements are eventually passed to the processor registers for the SDC module.

The `lcd_init_fb` is called twice in the probe function of the panel driver and the `ipu_sdc_init_panel` function is accessed only once. This is because the function detects when the configuration performed should be made for the background or foreground. In this case, it only accesses the SDC init panel function when the configuration is made for the background.

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

The above code checks if the logical IPU channel that is used at that time is employed for the background case. If that is true, then the `ipu_sdc_init_panel` function is called, else it skips that section.

After that section, the panel's probe function calls `lcd_poweron` (when background only), using functions from voltage and current regulator frameworks (such as regulator_enable).

The panel's probe function uses the `fb_register_client` function. A notifier block structure with a pointer is passed as an argument to a `lcd_fb_event` function. This function is called whenever an LCD event happens (such as when Qtopia starts).

mxcfb_claa_vga Flow Chart:

Figure 25 shows the mxcfb_claa_vga flow chart.

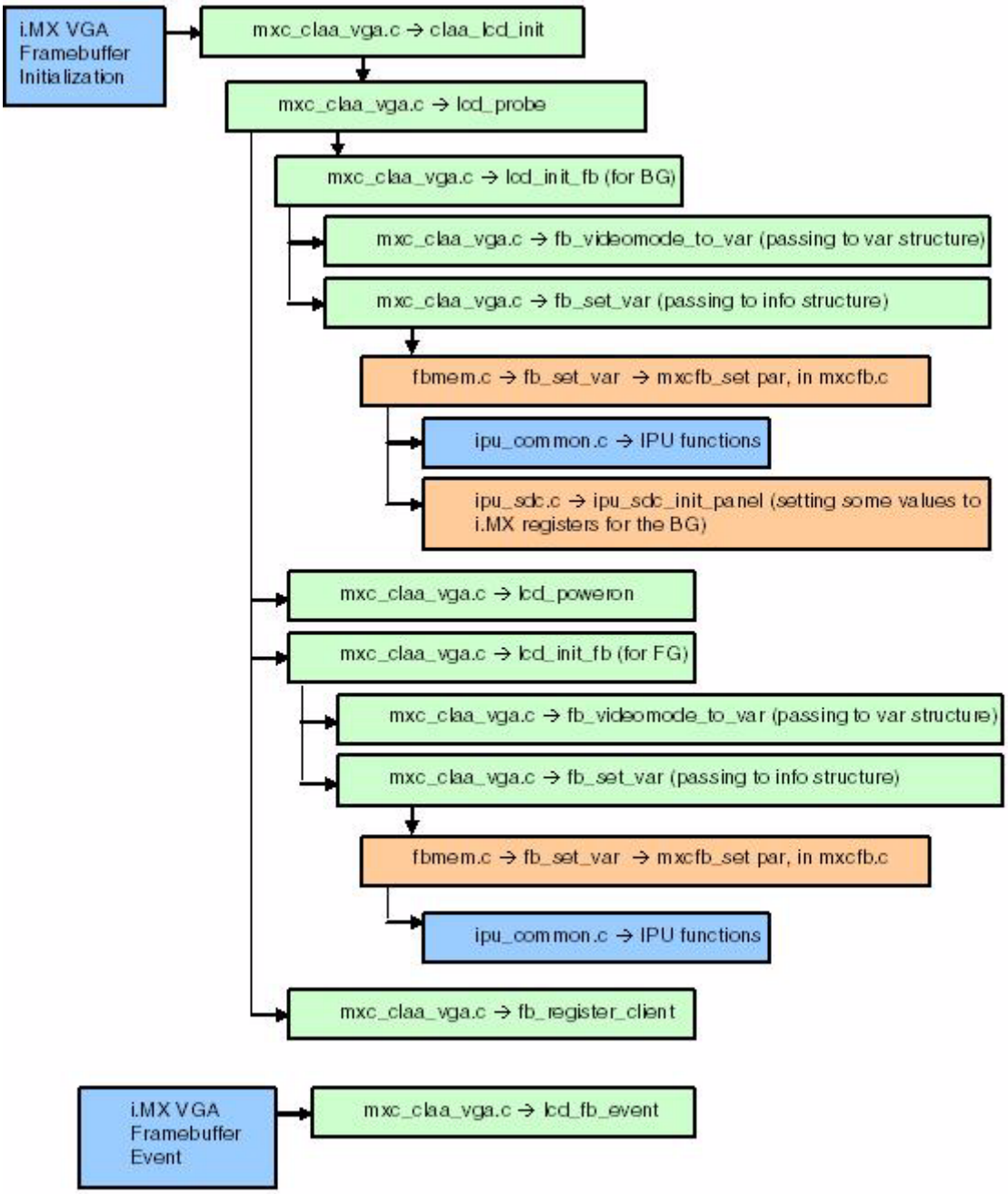

Figure 25. mxcfb_claa_vga Flow Chart

Linux voltage and current regulator Framework is used to have an interface that can work in the Linux Kernel 2.6 for controlling voltage and current levels. It also provides information to the user through a sysfs interface. The framework is made by Liam Girdwood from Wolfson Microelectronics.

See http://opensource.wolfsonmicro.com/node/15 for more information. The framework works with some power management ICs (PMIC) such as MC13787 from Freescale or the Wolfson WM8350.

The functions provided in the panel driver come from the …linux-2.6.26/drivers/regulator/reg_core.c file. Inside the …linux-2.6.26/drivers/regulator/ directory there is a subdirectory containing the code for the PMIC used in the i.MX35 3-Stack board (MC13892). Some of the functions used in the panel driver are: regulator_get, regulator_set_voltage, regulator_enable.

Another topic that is also present on the panel driver is the notifier chain or notifier block. The notifier chain is an information mechanism where different elements notify asynchronous events to the kernel.

The basic element is the notifier block structure (the definition can be found in …linux-2.6.26/include/linux/notifier.h file). Among the elements on the notifier block structure is a pointer to the event. In the panel driver there is a declaration of a notifier block having as an event argument the function lcd_fb_event. At the end of the probe function, is where the registration of the event is made by the fb_register_client that is holding as an argument the notifier block, containing as an element the lcd_fb_event pointer function. The fb_register_client function is inside the …linux-2.6.26/drivers/video/fb_notify.c file. It calls a function in charge or assembling the notifier block passed as an argument to the notifier chain. In other words, with the registration function, a callback is registered when a change happens in the LCD.

4 References

This application note has been written using the following references:

- i.MX35 (MCIMX35) Multimedia Applications Processor Reference Manual - IMX35RM- Rev. 2, 3/2009.
- i.MX35 PDK 1.0 Linux Reference Manual - Rev. 2, 3/2009.
- Embedded Linux Systems Design and Development, Raghavan P., Lad Amol, Neelakandan Sririam, Auerbach Publications, 2006, Chapter 9: Embedded Graphics 309-340 for Section 4.1.
- Different Display Configurations on i.MX31 Linux PDK, Olmedo, Lech; Corona, Ernesto, Freescale Semiconductor, 2008

5 Revision History

Table 16 provides a revision history for this application note. Note that this revision history table reflects the changes to this application note template, but can also be used for the application note revision history.

Table 16. Document Revision History

Rev. Number	Date	Substantive Change(s)
0	01/2010	Initial release.

THIS PAGE INTENTIONALLY LEFT BLANK

THIS PAGE INTENTIONALLY LEFT BLANK

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: AN3974
Rev. 0
01/2010