Freescale Semiconductor
Application Note

Document Number: AN4180
Rev. 0, 08/2010

System-80 Asynchronous Display on the i.MX31 WINCE 6.0 PDK

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

Contents

This application note provides the necessary information, considerations, and procedure to add or adapt a System-80 Type 2 (sampling with the read and write signals) asynchronous panel to the WINCE600 Board Support Package (BSP) for the i.MX31 Platform Development Kit (PDK). This application note describes the smart panel information and the generalities of the Asynchronous Display Controller (ADC). In addition, this application note also describes the development process to adapt a new panel to the BSP, considering that the framework driver structure is already provided by the operating system. This application note assumes that the reader is familiar with the Microsoft® Platform Builder packages and the WINCE Embedded 6.0 Device Driver concepts.

1 Overview of i.MX31 Display

As a multimedia processor, the i.MX31 supports several types of displays. The display devices are handled by a special module called the Image Processing Unit (IPU). The IPU module also handles other graphic interfaces such as the Camera interface and 2D graphics acceleration. All the IPU submodules are connected by a private

Direct Memory Access (DMA) Interface that is used for the IPU to transfer data between the submodules and also between the IPU and external memory.

Figure 1 shows the functional block diagram of the IPU module.

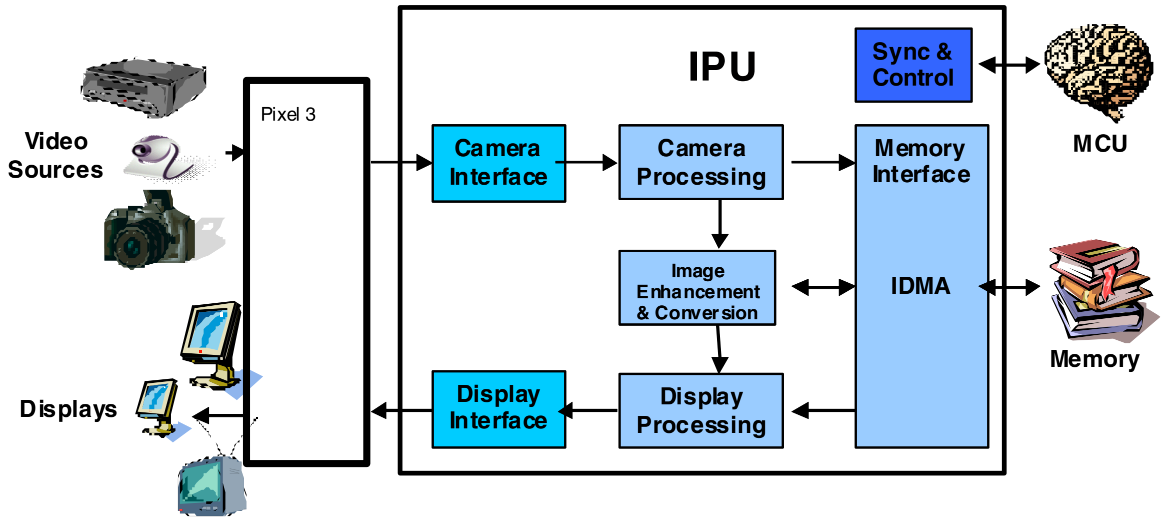


Figure 1. IPU Functional Block Diagram

Selecting an appropriate Liquid Crystal Display (LCD) for a mobile device involves several conflicts with respect to the requirements. Some of these conflicts are described as follows:

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

Freescale provides reference designs for the i.MX family where the functionality of LCD is demonstrated. However, according to the requirements, developers find many reasons to replace the display in their products. Features such as screen size, resolution, weight, power consumption, and price are very important in a commercial multimedia product. Another important fact about LCD panels is that many displays become obsolete quickly. Therefore, it is hard to find the same LCD panel included in the reference design while creating a product.

This application note is intended only for Smart Displays that include panels, which uses the System-80 Type 2 (sampling with the read and write signals) as display interface. However, this application note also provides some information about dumb displays.

2 LCD Generalities

This section describes the generalities of the LCD devices.

2.1 LCD Basics

LCD is an electronic device, which consists of an array of pixels, which can be either color or monochrome unit. Every element in the array is created with a special material that allows the LCD to change the characteristics of the light that passes through them. These devices do not emit light and therefore, another element named backlight is shipped along with the panel to create a fully functional display device.

2.1.1 Resolution

In this application note, the term resolution is used to refer to the number of pixels contained in an LCD array. It has two dimensions—horizontal and vertical.

Table 1 lists the most common video resolution standards.

Table 1. Video Resolution Standards

Video Name	Description	Width	Height	Aspect Ratio
CGA	Color Graphics Adapter	320	200	8:5
QVGA	Quarter VGA	320	240	4:3
VGA	Video Graphics Array	640	480	4:3
NTSC	National Television System Committee	720	480	3:2
PAL	Phase Alternating Line (TV)	768	576	4:3

The maximum standard resolution that the i.MX31 supports is SVGA and hence, resolutions greater than SVGA are not included in Table 1.

All resolutions mentioned in Table 1 show a landscape orientation of LCD panels, which means that there are more pixels in the horizontal axis than in the vertical axis. However, there are also portrait LCD panels available in the market with the same standard resolution but the horizontal and vertical size are inverted. These portrait LCD panels have more vertical pixels when compared to the horizontal pixels.

Figure 2 shows the portrait and landscape orientation of an LCD panel.

freescale™
Enabled
freescale™
Enabled

Figure 2. Portrait and Landscape Orientation of an LCD Panel

It is important to select an appropriate orientation of the LCD panel because both the electronic and optical features are optimized for applications that use the native orientation of the panel. Besides the optical characteristics, the dumb displays include an embedded LCD controller to draw the pixels from left to right and top to bottom. However, to show images or videos on the LCD panel using a non-native orientation, the display content is pre-processed. Therefore, the image is stored in a buffer in a way (order) the LCD controller expects the pixel information to be sent to it. This operation is called rotation and the i.MX31 includes hardware to perform this operation. It is recommended to select the LCD panel that mostly uses its native orientation to avoid additional image processing.

Figure 3 shows the portrait and landscape LCD panels displaying images in the non-native orientation.

freescale™
Enabled
freescale™
Enabled

Figure 3. Rotated Frame on Portrait and Landscape Orientation of an LCD Panel

The frames can be rotated in 90°, 180°, or 270°. Every frame has to be rotated before sending to the display.

2.1.2 Size

The size of an LCD panel is measured diagonally in inches, from top left corner to bottom right corner. Since the size directly impacts the pixel width, it is common to assume the size of a VGA (640 × 480) panel to be larger than a QVGA (320 × 240) panel because the number of pixels in VGA is four times greater when compared to QVGA. But, this is not true at all times. LCD manufacturing processes allow the size and resolution to be independent variables. It is difficult to determine the size of a panel from its resolution. Screens that are larger in size tend to consume more power than the smaller ones and also impact the size and weight of the final product. On the other hand, higher resolutions on smaller LCD panels can complicate the visibility for the final user. Based on the information available in the datasheet, it is difficult to determine if a particular LCD panel fits the application. Instead, it is recommended to view the LCD in other reference design or demo before making a final decision.

2.1.3 Color Spaces

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

2.2 LCD Types

The LCD panels are categorized as synchronous and asynchronous panels and their descriptions are described in the following sections.

2.2.1 Synchronous Panel (Dumb Display)

Dumb displays or synchronous displays are panels which require the microprocessor to send all pixels in the image every frame. In these panels, screen refresh is performed by driving the complete frame data continuously. In general, smart displays are more expensive than dumb displays and due to this reason, synchronous panels are more commonly used in the final product.

2.2.2 Asynchronous Panel (Smart Display)

The advantage of smart displays is that the i.MX31 has to send only the display data when the image has changed, and most of the times it sends only the portion that has changed. Images can be sent at any time and the screen refresh is handled by the embedded Smart Liquid Crystal Display Controller. Another advantage of smart displays is that the i.MX31 can handle three asynchronous displays and synchronous interface simultaneously. If an application requires two LCD panels, one of them must be an asynchronous interfaces.

3 Asynchronous Display Interfaces

There are two types of asynchronous display interfaces—System-80 LCD interface and 68 K interface.

3.1 System-80 (Type 2) LCD Interface

The System-80 LCD interface is one of the widely used interfaces for smart displays. It is also known as 80-series system bus interface or Intel 80 interface. This interface is composed of four control lines—CS, RS, WR, and RD, and the data bus whose width can vary. The i.MX31 can handle the Type 1 (sampling with the chip select signal) and Type 2 (sampling with the read and write signals) interfaces, but this application note focuses only on the Type 2 interface, which is the most commonly used interface.

Table 2 shows the System-80 (Type 2) interface signals.

Table 2. System-80 (Type 2) Interface Signals

Signal	IPU Signal	Description
CS0	IPP_DO_DISPB_D0_CS	Display 0 chip select
CS1	IPP_DO_DISPB_D1_CS	Display 1 chip select
CS2	IPP_DO_DISPB_D2_CS	Display 2 chip select
RS	IPP_DO_DISPB_PAR_RS	Data/command select for parallel interface
WR	IPP_DO_DISPB_WR	Write strobe signal
RD	IPP_DO_DISPB_RD	Read strobe signal
DB [15:0]	DISPB_DATA [15:0]	Data bus

The various signals that are available in the System-80 LCD interface are as follows:

- CS 0–2—Chip Select (CS) signal is used to communicate to the smart LCD panel that the commands and data used in the System-80 interface belongs to LCD. The i.MX31 can handle up to three asynchronous panels, and the method used to distinguish which LCD is addressed by the microprocessor by using a CS per panel. Most of the times, the CS0 signal is active low and the tag used for this line is `nCS`.

- RS—Register Select (data/command) is a control signal for the System-80 interface of the asynchronous display (0–2). Using the RS signal, the processor communicates to the LCD panel if the information in the bus has to be interpreted either as a command or data. In general, command mode is used to set the register address before sending the register's value using the data mode.
- WR—When the Write strobe signal is active, the processor is indicating that either the command or data is on the data bus and it must be read by the LCD panel.
- RD—The Read strobe signal is used by the i.MX31 to request the LCD panel to place the command or data in the bus when the processor is ready to read the information and this information is related with the previous command.
- DI—The Display Interface (DI) is used to send/receive data and command to/from the LCD panel. Depending on the LCD interface, the bus width can vary from 7 to 18 bits.

Figure 4 shows the LCD interface between the i.MX31 processor and the Giant plus GPM722A0 VGA panel.

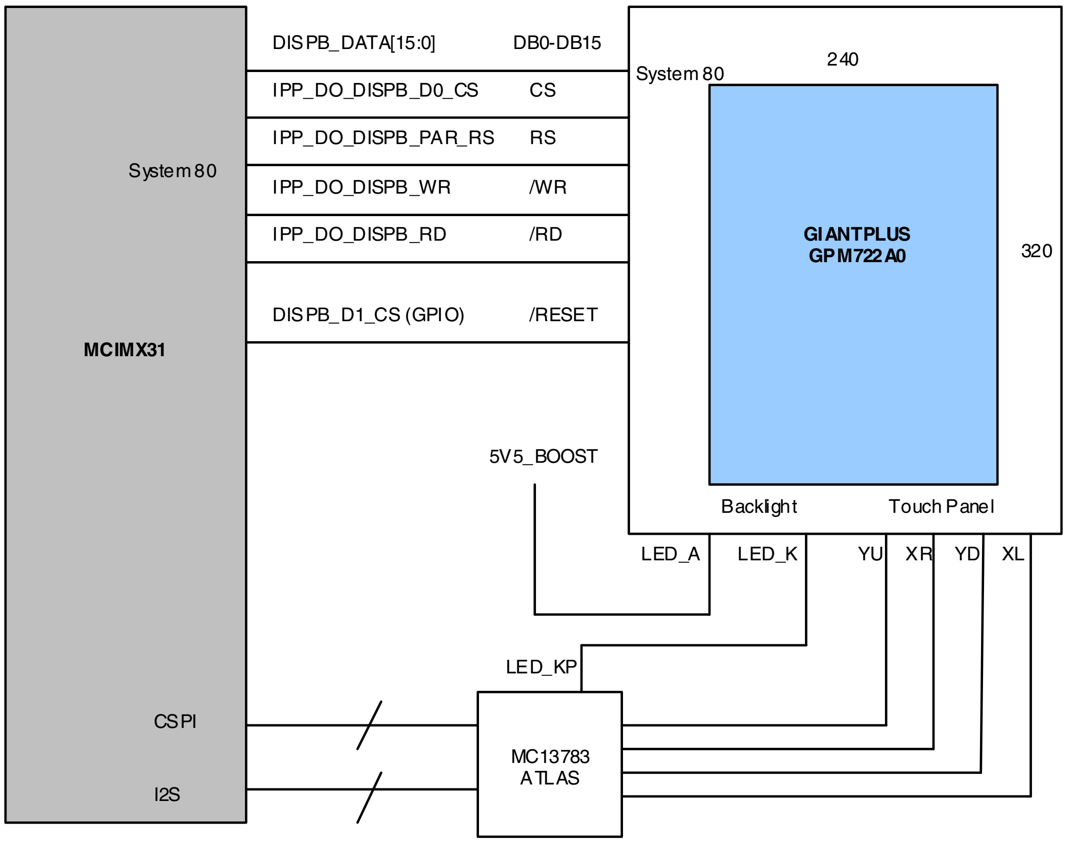


Figure 4. LCD Interface Between the i.MX31 and GiantPlus GPM722A0 VGA Panel

Figure 4 shows how i.MX31 is interfaced with a smart LCD panel such as the GPM722A0 GiantPlus. The System-80 (Type 2) interface is used for LCD initialization (command mode) and also for sending the display buffers (template data mode). Additionally, the LCD panel requires a reset signal, external backlight power booster circuit, and touch screen support. In PDK (only), the touch screen support feature is provided by PMIC ATLAS MC13783 by using the keypad LED circuit. Also, the PMIC settings are configured by the i.MX31 using an SPI interface.

3.2 68 K Interface

The 68 K interface is from Motorola and it is one of the widely used interfaces for smart displays. This interface is composed of four control lines—CS, RS, RW_WR, and E_RDB, and the data bus, whose width can vary. The difference between this interface and the System-80 interface is the inclusion of an Enable signal (E_RDB), which is used to indicate when the data needs to be latched for read and write operations. Also, unlike System-80 interface, this interface combines the read or write selection in a single line (RW_WR).

Table 3 shows the 68 K interface signals.

Table 3. 68 K Interface Signals

Signal	IPU Signal	Description
CS0	IPP_DO_DISPB_D0_CS	Display 0 chip select
CS1	IPP_DO_DISPB_D1_CS	Display 1 chip select
CS2	IPP_DO_DISPB_D2_CS	Display 2 chip select
RS	IPP_DO_DISPB_PAR_RS	Data/command select for parallel interface
RD_WR	IPP_DO_DISPB_RD_WR	Read/Write strobe signal
EN	IPP_DO_DISPB_EN	Enable
DB [15:0]	DISPB_DATA [15:0]	Data bus

The various signals that are available in the 68 K interface are as follows:

- CS 0–2—Chip Select (CS) signal is used to communicate to the smart LCD panel that the commands and data used in the 68 K interface belongs to LCD. The i.MX31 can handle up to three asynchronous panels, and the method used to distinguish which LCD is addressed by the microprocessor is by using one CS per panel. Most of the times, the CS signal is active low and the tag used for this line is `nCS`.
- RS—Register Select (data/command) is a control signal for the 68 K interface of the asynchronous display (0–2). Using the RS signal, the processor communicates to the LCD panel if the information in the bus has to be interpreted either as a command or data. In general, command mode is used to set the register address before sending the register’s value using the data mode.
- RD_WR—Read/Write control signal is used to distinguish whether the information on the data bus is being written to the LCD, or if the instruction is a reading request to the LCD.
- EN—Enable signal determines when the data in the bus is ready to perform both the read and write operations.
- DI—Depending on the LCD interface, the bus width can vary from 7 to 18 bits. This application note does not focus on the display interface. Therefore, there are no details available about this interface.

3.3 Asynchronous Serial Interfaces

Apart from the parallel interface, there are also serial interfaces that can be used to control the smart panels. The i.MX31 IPU can handle the following types of asynchronous serial interfaces:

- 3-wire (with bidirectional data line)
- 4-wire (with separate data input and output lines)
- 5-wire type 1 (with sampling RS by the serial clock)
- 5-wire type 2 (with sampling RS by the chip select signal)

As this application note do not focus on the above interfaces, there are no details available about all these interfaces.

3.4 System-80 (Type 2) Display Timing and Signals

This section focuses on the timing and signal waveforms and how to configure them in the LCD panel and the i.MX31 display interface. The first step in selecting an LCD module is to refer to its datasheet. The datasheet must contain the pin interface, system interface data format (that is, 16-bit or 18-bit interface, 1,2, or 3 transfer per pixel), initialization routine, and timing charts for the System-80 (Type 2) interface. Many times a shorter version of the datasheet is only available with no sufficient information. In this case, it is advisable to request for full documentation from the supplier. The document that is received contains a message (in every sheet) indicating that the document is only a preliminary version. Though there is not much difference between the preliminary and final versions, it is always better to have the final version.

3.4.1 Timing Charts

To understand the timing issues in a System-80 LCD interface, review the charts—CS, RS, RD, WR, and Data bus in the datasheet.

These charts can be used to verify the logical value of every signal while performing any operation. Most of the System-80 LCD datasheets contain diagrams to show how to read, write, and send commands to the LCD controller.

For example, if the LCD is using the System-80 (Type 2) interface with 18/16 bit data system interface, then the diagrams are similar to the following figures.

Figure 5 shows the write sequence of the System-80 (Type 2) 18/16 bit interface.

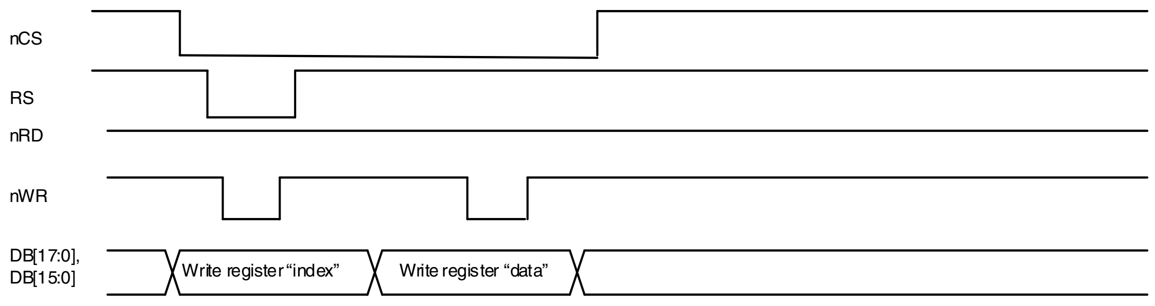


Figure 5. Write Sequence of the System-80 (Type 2) 18/16 Bit Interface

In this figure, the CS, RD, and WR signals are active low. These signals become active only when the signal goes low. From Figure 5, it is verified to be a System-80 (Type 2) interface, because the data is latched by using the RD and WR signals. When RS is the only signal used in conjunction with WR signal and also when RS is low, the information on the data bus must be treated as a command (register). On the other hand, when RS has a high value and WR is active, the information on the data bus is related with some data, which is typically a straight RS polarity.

3.4.1.1 Read from Register

Figure 6 shows the read sequence of the System-80 (Type 2) 18/16 bit interface.

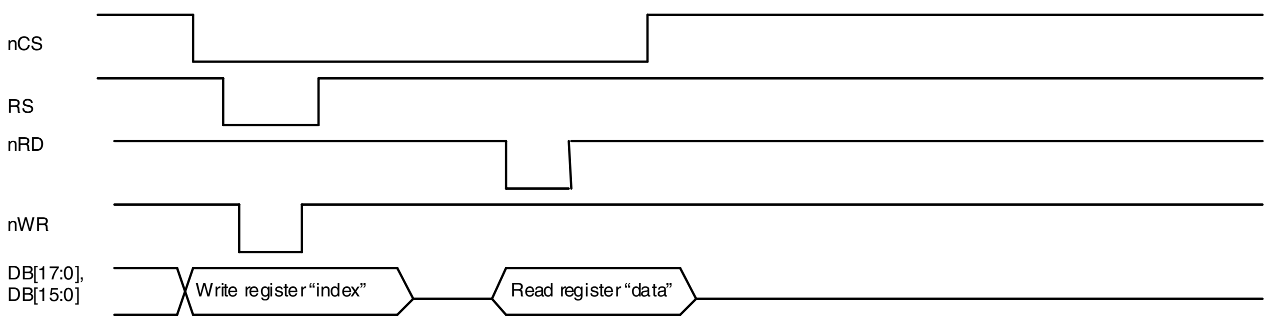


Figure 6. Read Sequence of the System-80 (Type 2) 18/16 Bit Interface

Figure 6 shows that the RD signal is used to indicate when the data is going to be read and this signal is used only after a command is issued.

It is observed that the data polarity is always straight for color LCDs and can be inverted for monochromatic LCDs.

3.4.2 Timing Concepts

Though the previous figures are helpful in understanding the logical flow of the interface, more details about the timing of each state (in ns) are required to configure the display interface.

Table 4 shows the timing concepts in the asynchronous System-80 (Type 2) LCD interface.

Table 4. Timing Concepts

Timing Concepts	Description
Write system cycle time (t_{CYCW})	Width of the write signal strobe cycle that is used to mark the writing of each data or command to the LCD. It is the combination of the write low pulse width and the write high pulse width and the transition between these states.
Read system cycle time (t_{CYCR})	Width of the read signal strobe cycle that is used to mark the reading of each data or command from the LCD panel. It is the combination of the read low pulse width and the read high pulse width and the transition between these states.
Write low pulse width (t_{WL})	Time for which the WR signal must be low for a valid write system cycle time.

Table 4. Timing Concepts (continued)

Write high pulse width (t_{WH})	Time for which the WR signal must be high for a valid write system cycle time.
Read low pulse width (t_{RL})	Time for which the RD signal must be low for a valid read system cycle time.
Read high pulse width (t_{RH})	Time for which the RD signal must be high for a valid read system cycle time
Setup time for write (t_{DCSW})	Time measured between the t_{CYCW} initialization (RS) until the WR signal goes low.
Setup time for read (t_{DCSR})	Time measured between the t_{CYCR} initialization (RS) until the RD signal goes low.
Write signal fall time (t_{Wf})	Maximum acceptable time that the WR signal must spend in the transition from high-level to low-level state. Most of the times, this value is ignored, and it is assumed that the i.MX LCD interface is able to switch the signal at high speed.
Write signal rise time (t_{Wr})	Maximum acceptable time that the WR signal must spend in the transition from low-level to high-level state. Most of the times, this value is ignored, and it is assumed that the i.MX LCD interface is able to switch the signal at high speed.
Read signal fall time (t_{Rf})	Maximum acceptable time that the RD signal must spend in the transition from high-level to low-level state. Most of the times, this value is ignored, and it is assumed that the i.MX LCD interface is able to switch the signal at high speed.
Read signal rise time (t_{Rr})	Maximum acceptable time that the RD signal must spend in the transition from low-level to high-level state. Most of the times, this value is ignored, and it is assumed that the i.MX LCD interface is able to switch the signal at high speed.
Hold time for write (t_{DCHW})	After the WR signal switches from low-level to high-level (write latch instruction), the address hold time is the time that the RS signal needs to maintain in its current value (0 = command and 1 = Data).
Hold time for read (t_{DCHR})	After the RD signal switches from low-level to high-level (write latch instruction), the address hold time is the time that the RS signal needs to maintain in its current value (0 = command and 1 = Data).
Write data set up time (t_{DS})	Time required to verify whether the data bus has the valid data before executing the write latch instruction.
Write data hold time (t_{DH})	Time required to verify whether the i.MX31 has the valid data on the bus after executing the write latch instruction.
Read data delay time (t_{RACC})	Time taken by the LCD to place the valid data on the bus after it receives the read start instruction (RD transition from high-level to low-level).
Read data hold time (t_{ROH})	Time taken by the LCD to place the valid data on the bus after it receives the read latch instruction (RD transition from low-level to high-level).

Figure 7 shows the write cycle timing of the System-80 (Type 2) interface.

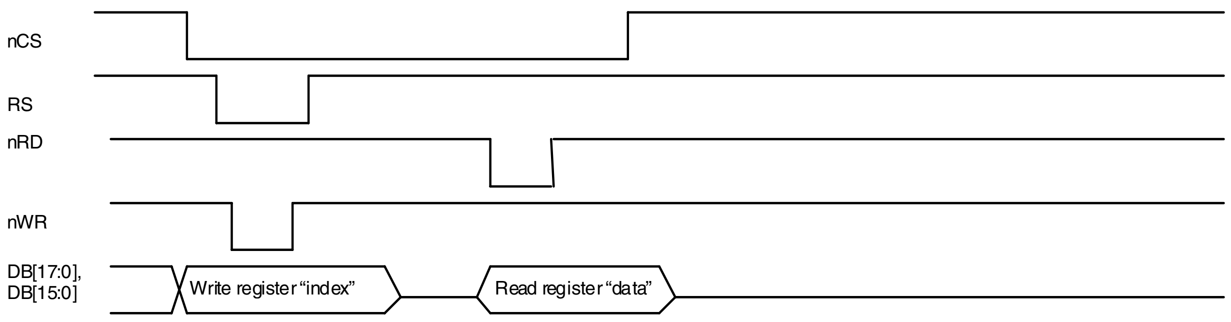


Figure 7. Write Cycle Timing of the System-80 (Type 2) Interface

Figure 8 shows the read cycle timing of the System-80 (Type 2) interface.

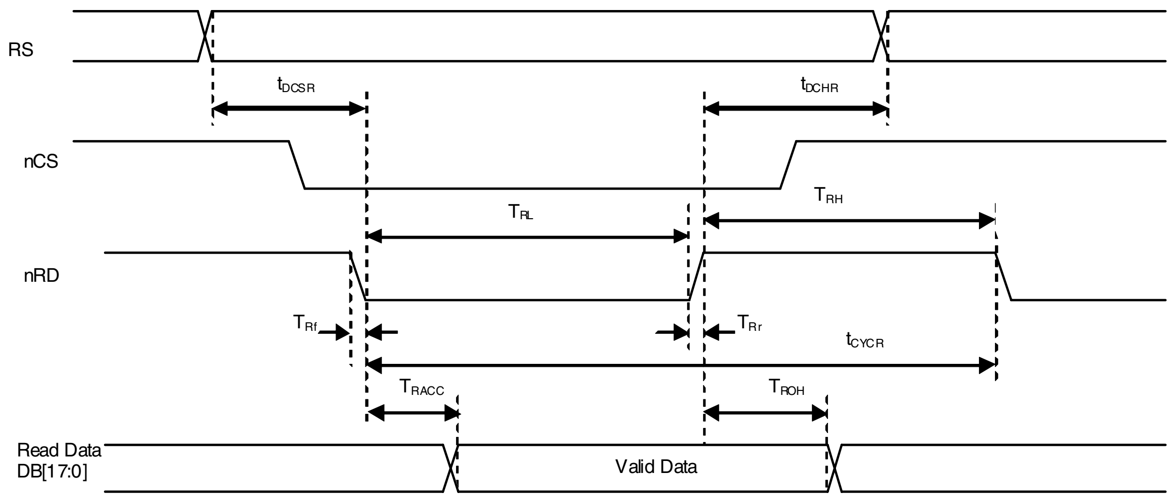


Figure 8. Read Cycle Timing of the System-80 (Type 2) Interface

3.4.3 Custom LCD Timing

This section describes the reset signal and its artifacts.

3.4.3.1 Reset

Many LCD panels include an LCD controller, which needs an external system reset. If the LCD uses the reset signal, the timing regarding the pulse has to be found.

Figure 9 shows the example of a reset signal.

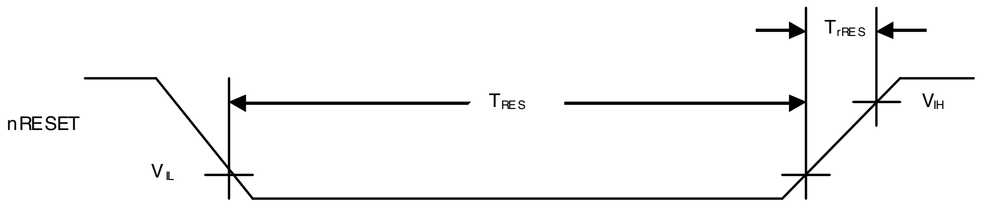


Figure 9. Reset Signal Example

Table 5 provides the artifacts of the reset signal.

Table 5. Artifacts of Reset Signal

Parameter	Symbol	Minimum	Type	Maximum	Unit
Reset low-level width	T_{RES}	1	—	—	ms
Reset rise time	T_{rRES}	—	—	10	µs

The important fact that can be observed is the reset signal is active low. This means that the reset signal must be high while performing normal operations. The reset signal must be low for at least 15 ns to ensure a valid reset. The reset pin is controlled by the i.MX31 General Purpose Input/Output (GPIO). It is recommended not to use the RC circuit to generate the reset signal as it restricts the rising time of the signal to 10 µs.

3.5 System Interface Data/Command Format

The data width is not the only characteristic of the data interface. Another important feature of the smart panel is the Interface Data Format.

To explain this, the following example needs to be reviewed:

Consider that an RGB565 LCD is available and a 16-bit data bus width is used. The first step is to send all the RGB data using one single transfer (16-bit data bus and one transfer per pixel) through the parallel data bus and this approach is an appropriate way to do it.

Figure 10 shows the data format of the RGB565 LCD system interface.

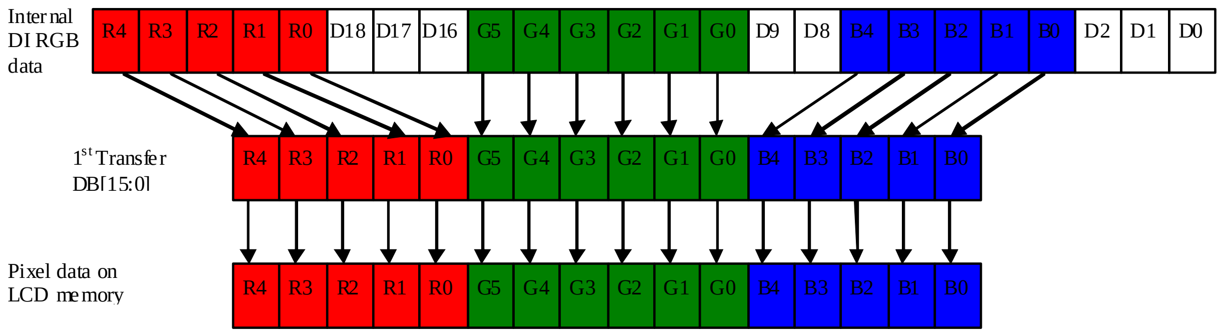


Figure 10. RGB565 16 Bit Data Bus, 1 Transfer Per Pixel System Interface Data Format

Now, suppose that this same LCD also supports RGB666 and an image is created based on this pixel depth. It is evident that all the RGB data cannot be sent through the 16-bit data bus by using only one transfer (i.MX31 data bus is 18-bit width maximum, but currently only 16 bits are used because LCD supports only 16-bit data transfer). For example, send all the red and green components in the first transfer and then send the remaining two blue bits in the following transfer. Then, in the next transfer start over with the next pixel.

Figure 11 shows the data format of the RGB666 LCD system interface.

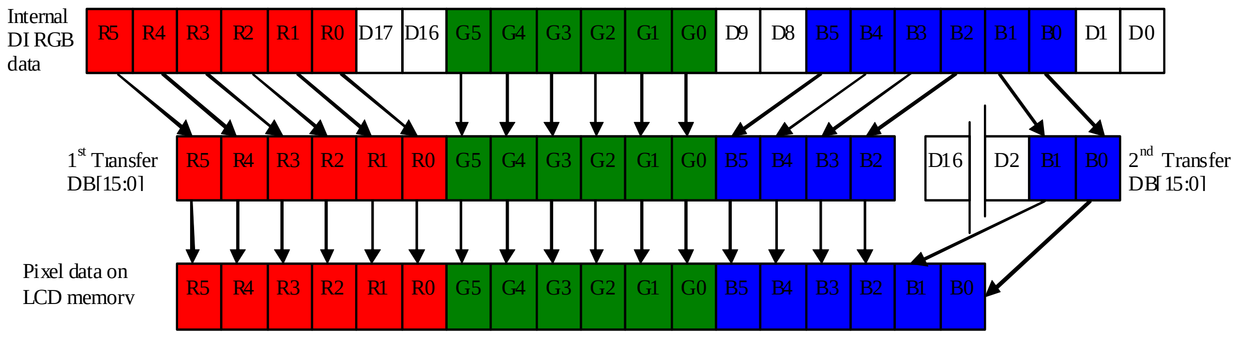


Figure 11. RGB666 16 Bit Data Bus, 2 Transfers Per Pixel System Interface Data Format

However, the LCD can request for a different system interface data format, where the dummy data has to be sent in the first transfer and not in the second. Also, R5 and R4 must be placed in the most significant bit (msb) of the data bus instead of the least significant bit (lsb).

Figure 12 shows the data format of the RGB666 LCD system interface.

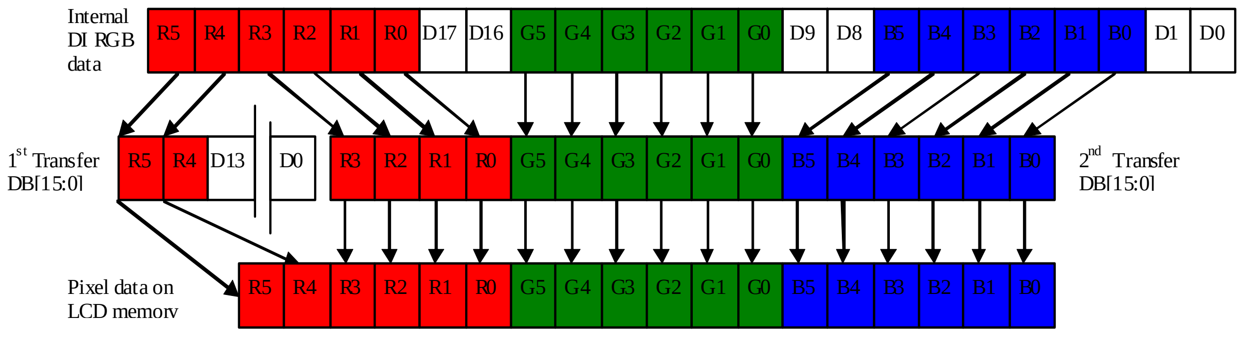

Internal DI RGB data
R5 R4 R3 R2 R1 R0 D17 D16 G5 G4 G3 G2 G1 G0 D9 D8 B5 B4 B3 B2 B1 B0 D1 D0
1st Transfer DB[15:0]
R5 R4 D13 D0
R3 R2 R1 R0 G5 G4 G3 G2 G1 G0 B5 B4 B3 B2 B1 B0
2nd Transfer DB[15:0]
Pixel data on LCD memory
R5 R4 R3 R2 R1 R0 G5 G4 G3 G2 G1 G0 B5 B4 B3 B2 B1 B0

Figure 12. RGB666 16 Bit Data Bus, 2 Transfers Per Pixel System Interface Data Format

Though Figure 11 and Figure 12 give the best performance regarding data transfers between the i.MX and LCD, there are also other means to send this data. For example, each color component can be sent in a single data transfer. First, send the six bits of the red component, then send the six green bits in the next transfer, and then in the third transfer, send the remaining five bits of the blue component. After this transfer, the process begins with the next pixel data.

Figure 13 shows the data format of the RGB666 LCD system interface.

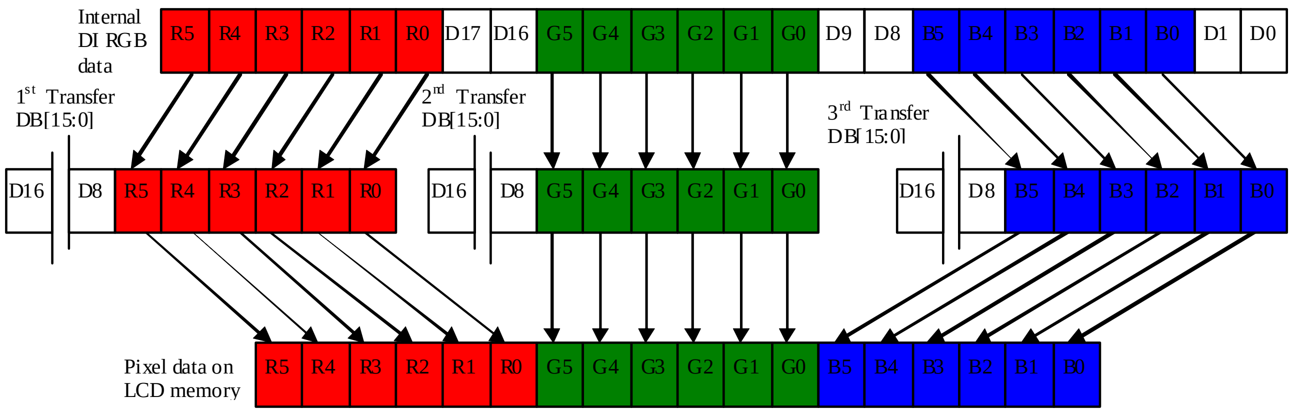


Figure 13. RGB666 16 Bit Data Bus, 3 Transfers Per Pixel System Interface Data Format

The same LCD supports more than one color depth (bits per pixel) and more than one system interface data format. The system interface is selected using a specific pin or register configuration. If the system interface is selected by a register, the question that arises is, how the data is sent before selecting the system interface. This is possible because command and pixel data transfer system interface are not always the same. The command interface can perform only one transfer per command, but the data transfer system interface can perform two or three transfers per pixel.

3.6 LCD Panels Supported by the i.MX31

The i.MX31 can handle up to four displays at the same time.

Table 6 lists the various types of displays that are handled by the display controllers.

Table 6. Display Controllers and their Interfaces

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

3.6.1 Asynchronous Display Interface

The i.MX31 ADC can be configured to handle several types of devices and interfaces. The ADC is able to support up to three smart displays simultaneously with time multiplexed access. The DISP1 and DISP2 provide parallel and serial display interfaces while DISP0 provides only the parallel interface.

The interfaces supported by the ADC are as follows:

- System-80 interface
  - Type 1 (sampling with the chip select signal) with or without byte enable signals
  - Type 2 (sampling with the read and write signals) with or without byte enable signals
- System 68 K interface
  - Type 1 (sampling with the chip select signal) with or without byte enable signals
  - Type 2 (sampling with the read and write signals) with or without byte enable signals
- Serial interfaces
  - 3-wire (with bidirectional data line)
  - 4-wire (with separate data input and output lines)
  - 5-wire type 1 (with sampling RS by the serial clock)
  - 5-wire type 2 (with sampling RS by the chip select signal)

For more details, refer to the Image Processing Unit chapter of the *MCIMX31 and MCIMX31L Multimedia Applications Processors Reference Manual* (MCIMX31RM).

For parallel interfaces, the data bus is time multiplexed with DISP3 (synchronous interface) and with the other parallel asynchronous interfaces. If there is a synchronous panel in the system, the asynchronous displays can be accessed only when the i.MX31 does not send any data to the dumb display (implies back and front porches). Based on this, it is observed that the bandwidth for parallel asynchronous LCDs are limited by the synchronous panel frame rate, back porch, and front porch (both horizontal and vertical). When the Synchronous Display Controller (SDC) or ADC communicates with one of the four displays through the parallel interface, the Micro Controller Unit (MCU) is allowed to access another display through the serial interface simultaneously.

The i.MX31 is able to support a wide variety of system data interfaces. In general, for System-80 type 2, the i.MX31 supports data bus, whose width varies from 8 to 18 lines (DISPB_DATA [17:0]). The internal Display Interface (DI) format for data and command is a 24-bit word which is divided into three byte components (for a 16-bit word, eight zeros are added to the MSB from the ADC). Because of this, it is always better to assume that the RGB888 is available internally and also assume that there are 24 bits available for commands, even when the bits can be masked to 8 or 16 bits. This word can be either as an output or input in one, two, or three of the display clock cycles. Due to this, the ADC is able to provide the outputs—RGB565, RGB666, and RGB888 color depths and few other variations of color, which fits in the 24 bits. Another important feature of i.MX31 is that it has a separate system transfer configuration for both commands and data.

It is important to mention the registers that are used to configure the system interface. For DISP0, the registers used for data mapping are:

- DI_DISP0_DB0_MAP
- DI_DISP0_DB1_MAP
- DI_DISP0_DB2_MAP
- DI_DISP_ACC_CC

For command mapping, the registers used are:

- DI_DISP0_CB0_MAP
- DI_DISP0_CB1_MAP
- DI_DISP0_CB2_MAP
- DI_DISP_ACC_CC

For more details, refer to the Image Processing Unit chapter of the *MCIMX31 and MCIMX31L Multimedia Applications Processors Reference Manual* (MCIMX31RM).

The examples available in Figure 10 to Figure 13 can be reviewed using proper values to gain more clarity.

Table 7 shows the i.MX31 register values for a RGB565 transfer per pixel system interface.

Table 7. i.MX31 Register Values for 16 Bit Data Interface RGB565 1 Transfer Per Pixel

	OFFS0	**OFFS1**	**OFFS2**	**M7**	**M6**	**M5**	**M4**	**M3**	**M2**	**M1**	**M0**
Byte2 Red	15	0	0	3	3	3	0	0	0	0	0
Byte1 Green	10	0	0	3	3	0	0	0	0	0	0
Byte0 Blue	4	0	0	3	3	3	0	0	0	0	0
DISPX_IF_CLK_CNT_Y	0										

The OFFS1 and OFFS2 columns are set to 0 because the data is packed and sent in a single transfer and therefore, no data is sent in the second or third transfer. The OFFS0 determines how the color components are ordered in the first transfer (clock cycle). Here, the MSB of blue component is located in D4 (five bits), MSB of green component is located in D10 (6 bits) and MSB of red component is located in D15 (five bits). Since RGB565 is used, masking can reflect the color depth by specifying which is the MSB (not masked). The M5-M7 columns are always masked (3 implies masked) for red and blue components (five bits) and for the green component, only the M7 and M6 columns are masked (six bits). The M0-M4 columns in the red and blue components and the M0-M5 columns in the green component are filled with zeros because the data has to be enabled in the first clock cycle. The DISPX_IF_CLK_CNT_Y row is set to 0 (display clock cycles - 1) because only one transfer per pixel is used and therefore, only one clock cycle is required.

NOTE

X in DISPX_IF_CLK_CNT_Y can be 0, 1, or 2 depending on which display is selected, and Y can be C (command mapping) and D (data mapping).

Table 8 shows the i.MX31 register values for the RGB666 two transfer per pixel system interface.

Table 8. i.MX31 Register Values for 16 Bit Data Interface RGB666 2 Transfers Per Pixel

	OFFS0	OFFS1	OFFS2	M7	M6	M5	M4	M3	M2	M1	M0
Byte2 Red	15	0	0	3	3	0	0	0	0	0	0
Byte1 Green	9	0	0	3	3	0	0	0	0	0	0
Byte0 Blue	3	5	0	3	3	0	0	0	0	1	1
DISPX_IF_CLK_CNT_Y	1										

In this example, two transfers are required and so DISPX_IF_CLK_CNT_Y is set to one (number of clock cycles - 1). Using OFFS0, the first cycle is set up, where the red (six bits) and green (six bits) components are sent and hence, the red MSB is on D15 and the green MSB is on D9. Also, part of the blue component is sent in the first transfer (four MSB bits) and hence, the blue MSB is on D3. In the second transfer, only the remaining bits of the blue component is sent. Therefore, OFFS1 column contains only the blue offset, which is placed on D5, so that B0 and B1 are placed in the LSB. The M6 and M7 columns are masked for all colors (bytes), which imply that all color components are of six bits width. Using 0 or 1 in masks is used to specify whether the bits are in the first cycle (0 = first transfer) or second cycle (1 = second transfer).

Table 9 shows the i.MX31 register values for the RGB666 two transfer per pixel system interface.

Table 9. RGB666 16 Bit Data Bus, 2 Transfers Per Pixel System Interface Data Format

	OFFS0	OFFS1	OFFS2	M7	M6	M5	M4	M3	M2	M1	M0
Byte2 Red	15	17	0	3	3	0	0	1	1	1	1
Byte1 Green	0	11	0	3	3	1	1	1	1	1	1
Byte0 Blue	0	5	0	3	3	1	1	1	1	1	1
DISPX_IF_CLK_CNT_Y	1										

This example also requires two transfers and therefore, DISPX_IF_CLK_CNT_Y is set to one. Since RGB666 is used, M6 and M7 columns must be masked (3) to indicate the color depth. In the first transfer, only the two MSB of the red component are sent and so, red OFFS0 is 15. For the second transfer (OFFS1), the blue component starts at D5 and the green component at D11, but the remaining red information has an offset of 17 because the four least significant bits of the red component are sent using the lines, D12-D15. The masks determine that only R5 and R4 have to be sent in the first transfer and all the other bits are sent in the second cycle.

Table 10 shows the i.MX31 register values for the RGB666 three transfer per pixel system interface.

Table 10. RGB666 16 Bit Data Bus, 3 Transfers Per Pixel System Interface Data Format

	OFFS0	OFFS1	OFFS2	M7	M6	M5	M4	M3	M2	M1	M0
Byte2 Red	5	0	0	3	3	0	0	0	0	0	0
Byte1 Green	0	5	0	3	3	1	1	1	1	1	1
Byte0 Blue	0	0	5	3	3	2	2	2	2	2	2
DISPX_IF_CLK_CNT_Y	2										

The DISPX_IF_CLK_CNT_Y row indicates that three transfer per pixel are required and masks determine that the RGB666 is used. Here, the red component is sent in the first transfer (0), green component in the second transfer (1), and blue component in the third transfer (2). The MSB's of all the components are placed on D5, and this is reflected in the offsets. (OFFS0 for red, OFFS1 for green, and OFSS2 for blue)

The same concept can be applied to the command interface by considering the byte0, byte1, and byte2 store command information instead of color information. For more details, refer to the Bus Mapping Unit section in the *MCIMX31 and MCIMX31L Multimedia Applications Processors Reference Manual* (MCIMX31RM).

3.7 Command and Template Mode

For asynchronous panels, the two types of display accesses for registers and data are—the command and template mode. The command mode is used for initialization process and the template mode is used for the frame buffer update. Once the interfaces have been configured, both modes generate all the signals to write data or send commands to the smart panel.

3.7.1 Command Mode

The command mode is used for the initialization routine. In this mode, the asynchronous command interface is set up with the desired characteristics by using the DI_DISP_LLA_CONF register:

- Display Interface (DISP0, DISP1, or DISP2)
- Data or command (RS polarity)

Write the command or data to be sent to the smart panel in the DI_DISP_LLA_DATA register. With these two actions, the ADC sends the data or command from the DI_DISP_LLA_DATA register to the LCD in one, two, or three transfers according to the system command configuration that is described in the DI_DISP0_CB0_MAP, DI_DISP0_CB1_MAP, DI_DISP0_CB2_MAP, and DI_DISP_ACC_CC registers (offsets and masks). If another command or data with the same configuration needs to be sent, write the command or data again on the DI_DISP_LLA_DATA register.

The WINCE600 BSP provides a function where the low-level aspects such as DI_DISP_LLA_CONF and DI_DISP_LLA_DATA are encapsulated in the `ADCWriteCommand()` function. It is important to mention that the offsets and masks for the command system interface must be configured before using the `ADCWriteCommand()` function. See the following example:

Consider an interface where either the data (RS = 1) or command (RS = 0) is of 16-bit width. While using an interface, first send the register address (command) and then the register contents (data) in order to configure the register. Also, assume that the DISP0 16-bit width command is used with only one transfer per command.

Figure 14 shows the 16 bit command one transfer per command system interface data format.

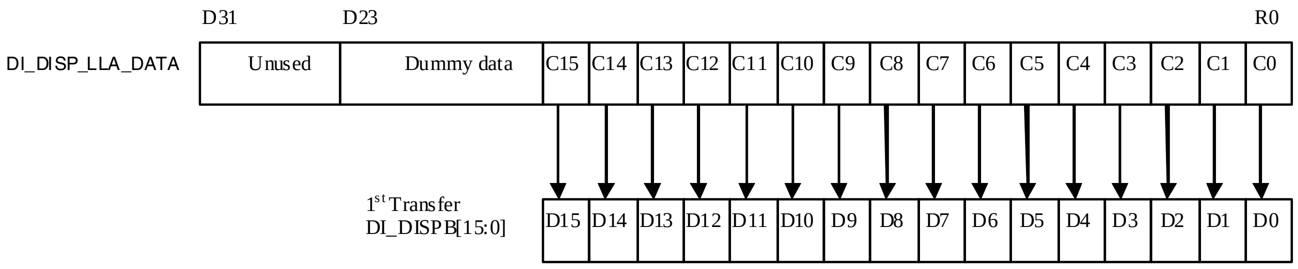

D31
D23
R0
DI_DISP_LLA_DATA
Unused
Dummy data
C15 C14 C13 C12 C11 C10 C9 C8 C7 C6 C5 C4 C3 C2 C1 C0
1st Transfer
DI_DISPB[15:0]
D15 D14 D13 D12 D11 D10 D9 D8 D7 D6 D5 D4 D3 D2 D1 D0

Figure 14. 16 Bit Command 1 Transfer Per Command System Interface Data Format

The following configuration shows that the DI_DISP_LLA_DATA byte0 and byte1 are sent to the display during the first clock cycle. Byte2 is ignored because there is only one transfer per command (DISP0_IF_CLK_CNT_C=0). Configure the interface before writing data to DI_DISP_LLA_DATA.

For more details, refer to the Memory Map and Register Definition section in the *MCIMX31 and MCIMX31L Multimedia Applications Processors Reference Manual* (MCIMX31RM).

Table 11 shows the data format of the 16 bit command one transfer per command system interface.

Table 11. 16 Bit Command one Transfer Per Command System Interface (Refer Figure 14)

	OFFS0	OFFS1	OFFS2	M7	M6	M5	M4	M3	M2	M1	M0
Byte2	0	0	0	3	3	3	3	3	3	3	3
Byte1	15	0	0	0	0	0	0	0	0	0	0
Byte0	7	0	0	0	0	0	0	0	0	0	0
DISP0_IF_CLK_CNT_C	0										

3.7.2 Template Mode

The template mode is used for sending pixels to the LCD. Once the LCD is initialized, the display data on the smart panel memory can be considered as an external memory where the i.MX31 display driver modifies either some part of the content or the complete frame buffer. In this way, the driver attempts to access only that memory and the ADC sets up all the signals in order to write those pixels in the smart panel memory.

A template is a micro program that is executed every time when a data burst has to be written to or read from a smart display. A typical structure of the template includes the following parts:

- Sending a pre-command sequence to the display.
- Sending an address to the display.

- Waiting for acknowledgement from the display or for the given number of display clocks and waiting is required only for the read access.
- Sending a data sequence to the display or receiving a data sequence from the display. This step is repeated when addressing is sequential (refer to the Sequential Addressing Access section of the *MCIMX31 and MCIMX31L Multimedia Applications Processors Reference Manual* (MCIMX31RM).
- Sending a post-command sequence to the display.
- Depending on the access type (read or write) and access, the template structure can be varied. For the sequential access (refer to the Sequential Addressing Access section of the *MCIMX31 and MCIMX31L Multimedia Applications Processors Reference Manual* (MCIMX31RM), both commands and addresses are not sent to the display. Therefore, templates are not used for the sequential access. Two templates—read and write exists for each of the three displays and the maximal template length is 32 commands.

There is no standard that specifies how this transfer is going to happen. For example, in the GiantPlus GPM722A0 if the frame buffer contents has to be modified, send the X (pixel offset) and Y (line) position of the pixel to the smart panel, by writing the addresses in the Graphics RAM (GRAM) horizontal (R20h) and GRAM vertical address (R21h) registers, and then write the pixel contents in the R22h register (write data to GRAM). After this operation, if the display interface writes again on the R22h register, the smart panel stores the RGB data in the next pixel location and so on.

Even when the addresses of X and Y are set every time a pixel is referenced, the i.MX31 is able to distinguish the continuous access and run only the address set up (R20h, R21h, and R22h). When a non-sequential access is referenced, the functionality needs to be configured in the i.MX31 IPU’s ADC_CONF register. This helps to reduce the transfer per pixel in the display interface. For more details, refer to the *MCIMX31 and MCIMX31L Multimedia Applications Processors Reference Manual* (MCIMX31RM).

3.7.2.1 Sending Pre-Command Sequence to the Display

There is no pre-command sequence for GiantPlus GPM722A0.

3.7.2.2 Sending an Address to the Display

The addresses to the GRAM horizontal/vertical address Set (R20h, R21h) registers need to be sent and the template command for this is as follows:

1. Send the GRAM horizontal address set command to the register (R20h).

Figure 15 shows the GRAM horizontal address set command (R20h).

RS	Flow control	Opcode	Data
0	Step-by-step	WR_CMND	0x20 (HOR_GRAM_ADDR_SET)

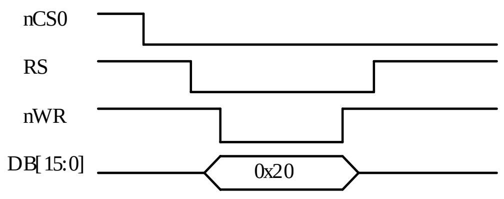


Figure 15. Horizontal Address Set Command

2. Send the contents of the GRAM horizontal address to the register.

Figure 16 shows the contents of the GRAM horizontal address.

RS	Flow control	Opcode	Data
1	Step-by-step	WR_XADDR	0x01 (send address bits [7:0])

Because this LCD is 240 pixels width we need 8 bits for X Addresses.

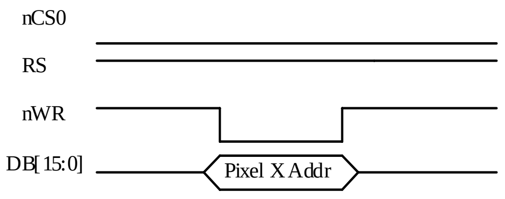


Figure 16. Contents of the GRAM Horizontal Address

3. Send the GRAM vertical address set command to the register (R21h).

Figure 17 shows the GRAM vertical address set command.

RS	Flow control	Opcode	Data
0	Step-by-step	WR_CMND	0x21 (VER_GRAM_ADDR_SET)

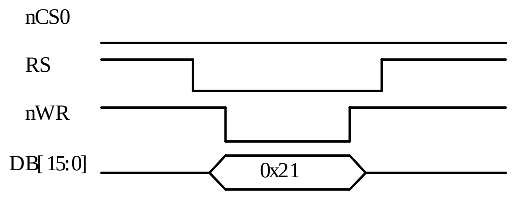


Figure 17. Vertical Address Set Command

4. Send the contents of the GRAM vertical address to the register.

Figure 18 shows the contents of the GRAM vertical address.

RS	Flow control	Opcode	Data
1	Step-by-step	WR_YADDR	0x01 (send address bits [22:8])

Because this LCD is 240 pixels width we need 8 bits for X Addresses. So Y address LSB is bit 8

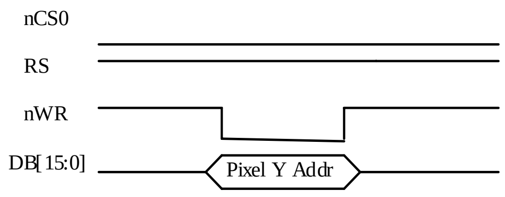


Figure 18. Contents of the GRAM Vertical Address

5. Send the write data to the GRAM command (R22h).

Figure 19 shows sending of the write data to the GRAM command.

RS	Flow control	Opcode	Data
0	Step-by-step	WR_CMND	0x22 (WR_DATA_TO_GRAM)

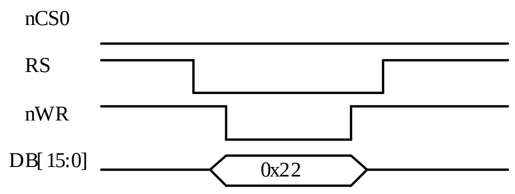


Figure 19. Send Write Data to the GRAM Command

3.7.2.3 Sending a Data Sequence to the Display

Finally, send the pixel information to the panel.

Figure 20 shows sending of the pixel information to the panel.

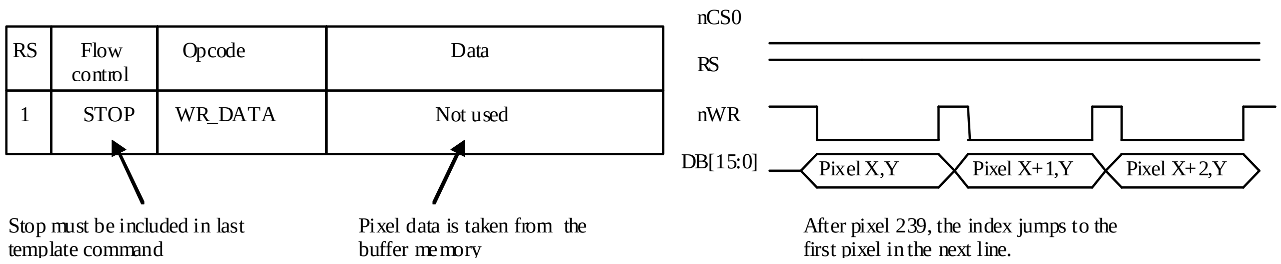

RS
Flow control
Opcode
Data
1
STOP
WR_DATA
Not used
Stop must be included in last template command
Pixel data is taken from the buffer memory
nCS0
RS
nWR
DB[15:0]
Pixel X,Y
Pixel X+1,Y
Pixel X+2,Y
After pixel 239, the index jumps to the first pixel in the next line.

Figure 20. Sending Pixel Information to the Panel

3.7.2.4 Sending a Post-Command Sequence to the Display

There is no post-command sequence for GiantPlus GPM722A0.

It is observed that nearly six transfers are required for loading the initial pixel data, but if the sequential access is used correctly, the total number of transfers can be reduced to one transfer per pixel.

4 Display Configuration in WINCE 6.0

LCD support is one of the most important features for any multimedia device. Display support enables the device to have a Graphical user interface (GUI) and the possibility to become an entertainment artifact.

The graphic context is composed of several layers where the i.MX31 display interface is the final part in the abstraction. All the ADC and display interface characteristics that are reviewed in previous sections describe only how the i.MX31 sends the frame buffer to the panel. However, it is important to know who is going to create the frames that need to be sent to the panel. If the screen is refreshed 60 times per second (60 Hz), every line and every single pixel has to be created to maintain the coherence of the graphic context.

The i.MX31 PDK BSP bases its display driver on the Display Driver Interface (DDI) defined by Microsoft® for all WINCE600 devices. Implementing a driver using this model ensures the compatibility of the hardware with the operating system. In other words, once WINCE is loaded and if the driver is created using the MS model, the operating system handles the graphic context, providing all frames.

4.1 WINCE600 Display Driver Development Concepts

Display drivers are loaded and called directly by the graphics, windowing, and event subsystem, called Gwes.exe. Drivers are most commonly written using a layered architecture because of the number of hardware-independent operations.

The Graphics Primitive Engine (GPE) library handles the default drawing, acting as the display driver's Model Device Driver (MDD) upper layer.

The user develops the hardware-specific code that corresponds to the display drivers lower layer, called the Platform-Dependent Driver (PDD).

Table 12 shows the elements that constitute the Windows CE graphics pipeline.

Table 12. Elements of Windows CE Graphics Pipeline

Element	Description
Application	The application can be simple such as a Hello World application or complex such as a three-dimensional engineering application. Whichever it is, the application calls GDI functions. Coredll.dll exposes these functions.
Coredll.dll	The major set of functions is exposed through a single DLL, called Coredll.dll. In most cases, this library does not perform the work. Instead, the library packages the parameters for the function call and then triggers a Local Procedure Call (LPC) to another process. The specific process depends on the function call. All drawing and windowing calls are sent to Gwes.exe.
Gwes.exe	The Graphics, Windowing, and Events Subsystem (GWES) is responsible for all graphical output and all interactions with the user. The drivers that reside in the GWES address space include display drivers, printer drivers, keyboard drivers, mouse drivers, and touch screen drivers.
Ddi.dll	The default name for the display driver is Ddi.dll. As with most DLLs, Ddi.dll communicates through exported functions. Ddi.dll exports only the `DrvEnableDriver` function, which returns a pointer to an array of 27 function pointers to the caller. When GWES requires a display driver, it calls one of the 27 functions. Writing a device driver involves writing the code for the 27 functions. Three of these functions are specific to printer drivers, which leaves 24 for the display driver developer.
Hardware	The graphic pipeline ends at the hardware. The display driver communicates to the hardware using the mechanism required by the hardware. This process typically involves a combination of memory-mapped video buffers and I/O registers.

Figure 21 shows the Windows CE graphics architecture.

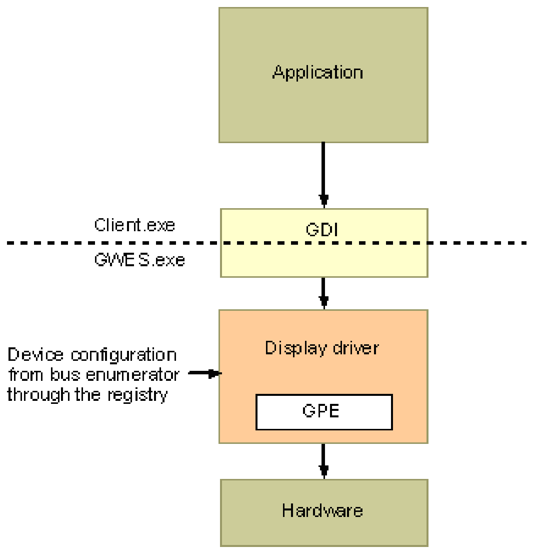

Application
Client.exe
GDI
GWES.exe
Device configuration from bus enumerator through the registry
Display driver
GPE
Hardware

Figure 21. Windows CE Graphics Architecture

More details can be found under Display Drivers (Developing a Device Driver > Windows CE Drivers > Display Drivers) topic of Platform Builder for Microsoft® Windows CE 5.0 help.

It is important to mention that developing a WINCE Display driver from scratch implies a considerable effort and knowledge, specially the WINCE architecture. Freescale provides the i.MX31 display driver for asynchronous displays in the WINCE600 BSP. The i.MX31 Windows CE 6.0 BSP display driver is based on the Microsoft® DirectDraw Graphics Primitive Engine (DDGPE) classes and supports the Microsoft® DirectDraw interface. This driver combines the functionality of a standard LCD display with DirectDraw support and the display driver interfaces with IPU. This driver supports more than one panel which can be selected by using the Windows Register. The support for a new asynchronous panel can be added using the procedure described in the following section.

4.2 Adding Support for a New LCD Panel

The following sections describes the procedure used to add the support for a new asynchronous panel.

4.2.1 Identify LCD Characteristics and Timing

To add the support for a new System-80 smart panel for the i.MX31 BSP, ensure that this panel is compatible with the i.MX31. The panel must have the following interface characteristics:

— Asynchronous display (smart display)

— System-80 Type 2 (sampling with the read and write signals) interface

— Resolution not greater than SVGA

Once a compatible LCD panel is selected, it is important to find the timing and system data interface characteristics of the display. Following tables are related to the GPM722A0 display.

Table 13 shows the artifacts of the GPM722A0 display.

Table 13. Artifacts of the GPM722A0 Display

Parameter	Symbol	Value
Display Interface	DI	System-80 Type 2 (sampling with the read and write signals)
Burst Mode	BRSTM	Parallel interface burst mode
Data Polarity	DEP	Straight
Chip Select Polarity	CSPOL	Active low
Register Select (PAR_RS) Polarity	RSPOL	Straight polarity
Write Signal Polarity	WRPOL	Active low
Read Signal Polarity	RDPOL	Active low
Color Depth (bits per pixel)	BPP	16(RGB565)/18(RGB666) bits (configurable)
System interface data format	DFMT	One or two transfer per pixel depending on color depth, one transfer per command
LCD width	HDISP	240 pixels
LCD height	VDISP	320 lines

Table 14 also shows the artifacts of the GPM722A0 display.

Table 14. Artifacts of the GPM722A0 Display

Parameter	Symbol	Minimum	Type	Maximum	Unit
Write Cycle Time	t_{CYCW}	100	—	—	ns
Read Cycle Time	t_{CYCR}	300	—	—	ns
Write Low Pulse Width	t_{WL}	50	—	500	ns
Write High Pulse Width	t_{WH}	50	—	—	ns
Read Low Pulse Width	T_{RL}	150	—	—	ns
Read High Pulse Width	T_{RH}	150	—	—	ns
Setup Time for Write	T_{DCSW}	10	—	—	ns
Setup Time for Read	T_{DCSR}	5	—	—	ns
Write Signal Fall Time	T_{WF}	—	—	25	ns
Write Signal Rise Time	T_{WR}	—	—	25	ns
Read Signal Fall Time	T_{RF}	—	—	25	ns
Read Signal Rise Time	T_{RF}	—	—	25	ns
Hold Time for Write	T_{DCHW}	15	—	—	ns
Hold Time for Read	T_{DCHR}	5	—	—	ns
Write Data set up Time	T_{DS}	10	—	—	ns
Write Data Hold Time	T_{DH}	15	—	—	ns
Read Data Delay Time	T_{RACC}	—	—	100	ns
Read Data Hold Time	T_{ROH}	5	—	—	ns

Note: Check if these values are same as the `PANEL_INFO` structure.

More details on how to deduce these values from the LCD data sheet can be found in the Section 1, “Overview of i.MX31 Display,” to Section 3, “Asynchronous Display Interfaces,” of this application note.

4.2.2 i.MX31 WINCE600 PDK LCD Driver Initialization Flow

The following snippet represents the WINCE600 PDK LCD driver initialization flow:

```
DDIPU::DDIPU()
+ DDIPU::Init()
+ DDIPU::SetupVideoMemory()
         + Set up the memory for video.
         + video memory size comes from a constant in image_cfg.h
         + IMAGE_WINCE_IPU_RAM_PA_START
+ InitHardware()
         + DDKClockSetpointRequest() - HSP_CLK = 132 MHz
         + BSPInitializeLCD(eIPU_ADC) - GiantPlusInitializeLCD()
                  + Configure Reset pin as GPIO:  LCS1, MCU3_24
                  + LCD_RESET_GPIO_PIN=1
+ InitializeADC()
```

```
         + Configure ADC and DI: IPU_CONF, IPU_CHA_DB_MODE_SEL, IPU_CHA_CUR_BUF,
                DI_DISPx_DB0_MAP, DI_HSP_CLK_PER, ADC_CONF
         + _init_dma()
         + CreateTemplate()
         + ReadTemplateMemory()
+ ADCSetSrcBuffer()
         + BSPEnableLCD(eIPU_ADC) - GiantPlusEnableLCD()
                  + PmicVoltageRegulatorOn(VMMC1); VMMC1 = 2.8 V
                  + PmicVoltageRegulatorOn(VGEN);  VGEN  = 2.8 V
                  + BSPDisplayIOMUXEnable() - GiantPlusDisplayIOMUXEnable()
                           + Configure LCD pins: LD0-17, LCS0, PAR_RS, WRITE, READ, etc)
                  + BSPResetLCD(eIPU_ADC) - GiantPlusResetLCD()
                           + LCD_RESET_GPIO_PIN=0
                           + Sleep(3000)
                           + LCD_RESET_GPIO_PIN=1
                  + ADCEnableModules()
                           + Enable DI
                           + Enable ADC
                  + Startup Command Sequence
                           ADCWriteCommand() - GiantPlusWriteRegister(
         + EnableADC()
                  + Enable DI
                  + Enable ADC
+ DDIPU::AdvertisePowerInterface()
```

The LCD driver is initialized in the `DDIPU_SDC::DDIPU_SDC()` function. The video memory size is not taken from the registry, because this value is a constant (IMAGE_WINCE_IPU_RAM_SIZE) declared in the `image_cfg.hc` file. Panel type, rotation parameters, pixel depth, and TV modes supported by the platform are extracted from the registry (`platform.reg` file). Based on the selected panel, the driver notifies the WINCE display properties such as width, height, bits per pixel, and so on. The `GPEMode` data is automatically set by using the `PANEL_INFO` structure of the current panel. With this information, WINCE graphic context creates the frame buffers in the width, height, and format required by the LCD, which is to be displayed on the screen. The information regarding the `GPEMode` must also be modified to complete the support for the new LCD. The width and height must be provided in the natural orientation of the LCD.

The next step is the hardware initialization. Here, the IPU registers are configured to enable the ADC and DI for working with the selected panel. The LCD timing features are not stored in registers, but are located in the array `PANEL_INFO g_ADCPanelArray[numPanel]` placed in the `WINCE600\PLATFORM\iMX313DS\SRC\DRIVERS\IPU\ADC\adc.c` file. So, to add the support for a new panel, this array must be updated by adding another `PANEL_INFO` structure with the panel timing information. The IDMAC ADC channels are configured, and the IOMUX is also configured to enable the LCD pin interface (LD0-LD17, LCS0, PAR_RS, WRITE, READ, and so on). Then using `bspdisplay.cpp`, the driver configures the specific LCD panel pins for initialization which includes reset and other enable pins that are required. At the end of the process, the power levels for LCD are enabled, the panel is reset and all initialization commands are sent to the LCD panel.

4.2.3 i.MX31 WINCE600 PDK LCD Display Interface Related Files

The files related to the i.MX31 WINCE600 PDK LCD display are contained in the following locations:

```
WINCE600\PLATFORM\iMX313DS\SRC\INC\adc.h
WINCE600\PLATFORM\iMX313DS\SRC\DRIVERS\IPU\ADC\adc.c
WINCE600\PLATFORM\iMX313DS\SRC\DRIVERS\IPU\DISPLAY\COMMON\ddipu.cpp
```

```
WINCE600\PLATFORM\iMX313DS\SRC\DRIVERS\IPU\DISPLAY\COMMON\ddipu_cursor.cpp
WINCE600\PLATFORM\COMMON\SRC\SOC\FREESCALE\MXARM11_FSL_V1\IPU\INC\ipu.h
WINCE600\PLATFORM\iMX313DS\SRC\DRIVERS\IPU\DISPLAY\DLL\bspdisplay.cpp
WINCE600\PLATFORM\iMX313DS\FILES\platform.reg
WINCE600\PLATFORM\COMMON\SRC\SOC\FREESCALE\PMIC\MC13783_FSL_V1\INC\pmic_regulator.h
WINCE600\PLATFORM\iMX313DS\SRC\INC\bsp_cfg.h
```

4.2.4 i.MX31 PDK LCD Structures

It is important to understand how and where the information related to the LCD panel settings (see Section 4.2, “Adding Support for a New LCD Panel,”) are stored because the communication channel is used to inform the Asynchronous LCD driver about the settings for the new LCD display. `PANEL_INFO`, which contains `ADC_IPU_DI_SIGNAL_CFG` and `SDC_IPU_DI_SIGNAL_CFG` is the structure that needs to be changed in order to add the support for a new panel. The `g_ADCPanelArray[]` array in `adc.c` file is the global array that stores the `PANEL_INFO` for all supported smart displays (Toshiba, Epson, and so on). To modify the driver, it is recommended to add a new entry at the end of the structure with the new `PANEL_INFO` structure.

4.2.4.1 PANEL_INFO

`PANEL_INFO` is the main structure for the LCD timing and features. It also contains two structures related to the signal polarity. The `ADC_IPU_DI_SIGNAL_CFG` is used for asynchronous displays. Alternatively, the `SDC_IPU_DI_SIGNAL_CFG` describes the polarities and characteristics of the RGB interface.

```
struct PANEL_INFO_ST {
         PUCHAR NAME;
         IPU_PANEL_TYPE TYPE;
         IPU_PIXEL_FORMAT PIXEL_FMT;
         INT MODEID;
         INT WIDTH;
         INT HEIGHT;
         INT FREQUENCY;
         INT VSYNCWIDTH;
         INT VSTARTWIDTH;
         INT VENDWIDTH;
         INT HSYNCWIDTH;
         INT HSTARTWIDTH;
         INT HENDWIDTH;
         INT RD_CYCLE_PER; // in ns
         INT RD_UP_POS; // in ns
         INT RD_DOWN_POS; // in ns
         INT WR_CYCLE_PER; // in ns
         INT WR_UP_POS; // in ns
         INT WR_DOWN_POS; // in ns
         INT PIX_CLK_FREQ; // in Hz
         INT PIX_DATA_POS; // in ns
         ADC_IPU_DI_SIGNAL_CFG ADC_SIG_POL;
         SDC_IPU_DI_SIGNAL_CFG SDC_SIG_POL;
};
typedef struct PANEL_INFO_ST PANEL_INFO;
```

Table 15 provides the description of each element in the `PANEL_INFO` structure.

Table 15. PANEL_INFO Structure Elements

Data Type	Variable Name	Description	Symbol	Unit
PUCHAR	NAME	Name of the panel	—	—
IPU_PANEL_TYPE	TYPE	Index of the panel type [1]	—	—
IPU_PIXEL_FORMAT	PIXEL_FMT	Pixel format [2]	BPP	bit
INT	MODEID	Mode ID [3]	—	ns
INT	WIDTH	Active frame width	VDISP	pixel
INT	HEIGHT	Active frame height	HDISP	line
INT	FREQUENCY	Refresh rate	FV	Hz
INT	VSYNCWIDTH	Not used by smart panels	—	—
INT	VSTARTWIDTH	Not used by smart panels	—	—
INT	VENDWIDTH	Not used by smart panels	—	—
INT	HSYNCWIDTH	Not used by smart panels	—	—
INT	HSTARTWIDTH	Not used by smart panels	—	—
INT	HENDWIDTH	Not used by smart panels	—	—
INT	RD_CYCLE_PER	Read cycle period	t_{CYCR}	ns
INT	RD_UP_POS	Read up position	T_{RH}	ns
INT	RD_DOWN_POS	Read down position	T_{RL}	ns
INT	WR_CYCLE_PER	Write cycle period	t_{CYCW}	ns
INT	WR_UP_POS	Write up position	t_{WH}	ns
INT	WR_DOWN_POS	Write down position	t_{WL}	ns
INT	PIX_CLK_FREQ	Pixel clock frequency [4]	—	—
INT	PIX_DATA_POS	Pixel data position [5]	—	—

[1] The enum on this data field is used by `adc.c` file to distinguish the proper timing settings between the supported displays (LCD, NTSC TV, and PAL TV) when the selected display is being loaded. The `ipu.h` header file contains IPU_PANEL_TYPE.

[2] There are three different RGB pixel formats, RGB565, RGB666, and RGB888. By selecting one of these formats, the ADC's interpretation of the frame buffer data can be specified.

[3] For LCD panels, use DISPLAY_MODE_DEVICE as MODEID. Two other supported modes are also available, but those values belong to the TV out functionality.

4.2.4.2 ADC_IPU_DI_SIGNAL_CFG

The following snippet represents the bit fields of the ADC display interface signal polarities:

```
// Bitfield of ADC Display Interface signal polarities
typedef struct {
        UINT32 DISP_NUM  :2;
        UINT32 DISP_IF_MODE:2;
        UINT32 DISP_PAR_BURST_MODE:2;
        UINT32 DATA_POL :1;            // true = inverted
        UINT32 CS_POL  :1;          // true = active high
        UINT32 PAR_RS_POL   :1;     // true = inverse
        UINT32 WR_POL    :1;        // true = active high
        UINT32 RD_POL  :1;          // true = active high
        UINT32 VSYNC_POL  :1;       // true = active high
        UINT32 SD_D_POL :1;         // true = inverse
        UINT32 SD_CLK_POL :1;       // true = inverse
        UINT32 SER_RS_POL :1;       // true = inverse
        UINT32 BCLK_POL :1;         // true = inverted
        UINT32 Dummy      :16;      // Dummy variable for alignment.
} ADC_IPU_DI_SIGNAL_CFG;
```

Table 16 provides the name and description of the ADC display interface bit fields.

Table 16. ADC Display Interface Bit Fields

Offset	Bit-Field Name	Description	Symbol
0	DISP_NUM	Display number[1]	—
2	DISP_IF_MODE	Display interface mode [2]	—
4	DISP_PAR_BURST_MODE	Bust mode for parallel interfaces [3]	—
5	DATA_POL	Data polarity [4]	DEP
6	CS_POL	Chip select polarity [5]	CSPOL
7	PAR_RS_POL	Parallel register strobe (REG/Data) polarity [6]	RSPOL
8	WR_POL	Write signal polarity [7]	WRPOL
9	RD_POL	Read signal polarity [8]	RDPOL
10	VSYNC_POL	VSYNC polarity (not used in GPM722A0) [9]	—
11	SD_D_POL	Data polarity for serial interface (not used in parallel)	—
12	SD_CLK_POL	Clock polarity for serial interface (not used in parallel)	—
13	SER_RS_POL	Register polarity for serial interface (not used in parallel)	—
14	BCLK_POL	Burst clock polarity [10]	—
15	Dummy	Dummy data for memory alignment	—

[1] Display number refers to the i.MX31 display that is used for this interface. For parallel interfaces (System-80 Type 2), DISP0, DISP1, or DISP3 can be used. Since this option is available, DISP0 (IPU_ADC_DISPLAY_0) can be selected.

[2] There are seven different types of interfaces supported by the i.MX31, but this application note uses only the IPU_DI_DISP_IF_CONF_IF_MODE_SYSTEM80_TYPE2 (0x01) interface, which is defined in the `mxarm11_ipu.h` file.

[3] The burst access mode must be IPU_DI_DISP_IF_CONF_PAR_BURST_MODE_BURST_CS for the GiantPlus GPM722A0.

[4] Data polarity must be straight (IPU_DI_DISP_SIG_POL_DATA_POL_STRAIGHT). If required, invert the signals in the data bus (used only for B/W displays).

[5] When the CS signal goes low to indicate it is active, the polarity of the CS_POL must be IPU_DI_DISP_SIG_POL_CS_POL_ACTIVE_LOW. Active low is the most common configuration for this signal. However, the CS_POL can also be IPU_DI_DISP_SIG_POL_CS_POL_ACTIVE_HIGH at times.

[6] Setting PAR_RS_POL to IPU_DI_DISP_SIG_POL_PAR_RS_POL_STRAIGHT implies low for commands and high for data in most common configuration. Additionally, IPU_DI_DISP_SIG_POL_PAR_RS_POL_INVERSE can be used if the interface requires a configuration, which is HIGH for commands and LOW for data.

[7] Write signal polarity is IPU_DI_DISP_SIG_POL_CS_POL_ACTIVE_LOW, which implies that the signal is active when it goes low. If the LCD requires, the IPU_DI_DISP_SIG_POL_CS_POL_ACTIVE_HIGH can also be used.

[8] Read signal polarity is IPU_DI_DISP_SIG_POL_WR_POL_ACTIVE_LOW, which implies that the signal is active when it goes low. Active Low is the most common configuration for this signal, similar to the Chip select and Write signal. The IPU_DI_DISP_SIG_POL_WR_POL_ACTIVE_HIGH configuration is also available for rest of the cases.

[9] VSYNC_POL refers to the polarity of VSYNC signal, which is used in some of the smart panels to synchronize the writings. But, in the case of GPM722A0, the module does not provide this signal and therefore, this polarity can be ignored in this driver. Additionally, if this signal is not available, the driver must be configured in order to ignore this synchronization. This is done by setting an environmental variable (BSP_ADC_ENABLE_TEARING_PREVENTION) in the `bsp_cfg.h` header file to false.

4.2.4.3 SDC_IPU_DI_SIGNAL_CFG

The following snippet represents the bit fields of the SDC display interface signal polarities:

```
// Bitfield of SDC Display Interface signal polarities.
typedef struct {
        UINT32 DATAMASK_EN:1;
        UINT32 CLKIDLE_EN :1;
        UINT32 CLKSEL_EN  :1;
        UINT32 VSYNC_POL  :1;
        UINT32 ENABLE_POL :1;
        UINT32 DATA_POL   :1;      // true = inverted
        UINT32 CLK_POL    :1;      // true = rising edge
        UINT32 HSYNC_POL  :1;      // true = active high
        UINT32 Dummy      :24;     // Dummy variable for alignment.
} SDC_IPU_DI_SIGNAL_CFG;
```

This structure is not used for asynchronous display panels and therefore, all these bitfields can be set to zero.

4.2.4.4 GPEMode

The following code snippet provides the `GPEMode` structure:

```
// STRUCT GPEMode
//
// This structure describes a display mode.
struct GPEMode
{
        int          modeId;
        int          width;
        int          height;
        int          Bpp;
        int          frequency;
        EGPEFormat   format;
};
```

The `GPEMode` structure is used to communicate to the WINCE graphics engine, the size and format of the screen. Using this information, the OS creates the frame in an appropriate size (required by the i.MX31), which is to be processed and sent to the panel using the display interface.

Table 17 provides the description of each element in the `GPEMode` structure.

Table 17. GPEMode Structure Elements

Data Type	Variable Name	Description	Symbol	Unit
int	modeId	Display or TV modes [1]	—	—
int	Width	Active frame width	VDISP	pixel
int	Height	Active frame height	HDISP	line
int	Bpp	Bits per pixel [2]	BPP	bit
int	Frequency	Screen refresh rate	FV	Hz
int	Format	EGPEFormat enum [3]	—	—

[1] Display mode can be either DISPLAY_MODE_DEVICE or DISPLAY_MODE_NTSC.
[2] The mode must be set to DISP_BPP, whose value is 16.
[3] This field represents the bits per pixel of the GPE frames and it takes any of the following values.

It is important to mention that the i.MX31 supports maximum of 18 bits as the display interface width. If more than 18 bits are used as BPP, the less significant bits are discarded when they are sent to the LCD. Due to this reason, the recommended values for this field are `gpe16Bpp` and `gpe24Bpp`.

```
enum EGPEFormat
{
         gpe1Bpp,
         gpe2Bpp,
         gpe4Bpp,
         gpe8Bpp,
         gpe16Bpp,
         gpe24Bpp,
         gpe32Bpp,
         gpe16YCrCb,
         gpeDeviceCompatible,
         gpeUndefined
};
```

In WINCE600, `GPEMode` is automatically set by the ADC driver.

5 Modifying the BSP

To add support for the i.MX31 PDK WINCE600 BSP, perform the following steps:

1. Modify the catalog
2. Modify the `platform.reg` file
3. Modify the `platform.bib` file
4. Set up the power LCD voltages
5. Set up the reset signal
6. Set up the backlight
7. Configure wait for VSYNC signal
8. Add a panel type enum for the new LCD

9. Create the `PANEL_INFO` entry for the new LCD
10. Configure data and command system interface
11. Write initialization command routine
12. Create a template for the frame buffer write access
13. Add the mouse support for asynchronous displays
14. Rebuild the project

For example, consider the Giantplus GPM722A0 LCD panel. This smart panel is a multi-color depth RGB565/RGB666, which uses the System-80 interface with 16-bit width transfers. But, choose RGB565 one transfer per pixel for the final implementation. Figure 5 and Figure 6 show how to read and write from/to the LCD memory/registers. Also, Figure 10 and Figure 14 show the data and command system interface transfers.

5.1 Modifying the WINCE600 Catalog

The WINCE600 catalog is changed by using the Microsoft® Visual Studio. The procedure to modify the WINCE600 catalog using the Microsoft® Visual Studio is as follows:

1. If the i.MX313dsmobility project is open, click File > Close Solution to close the project.
2. Click File > Open > File to open the catalog file under WINCE600\PLATFORM\iMX313DS\CATALOG folder.

Figure 22 shows the catalog file.

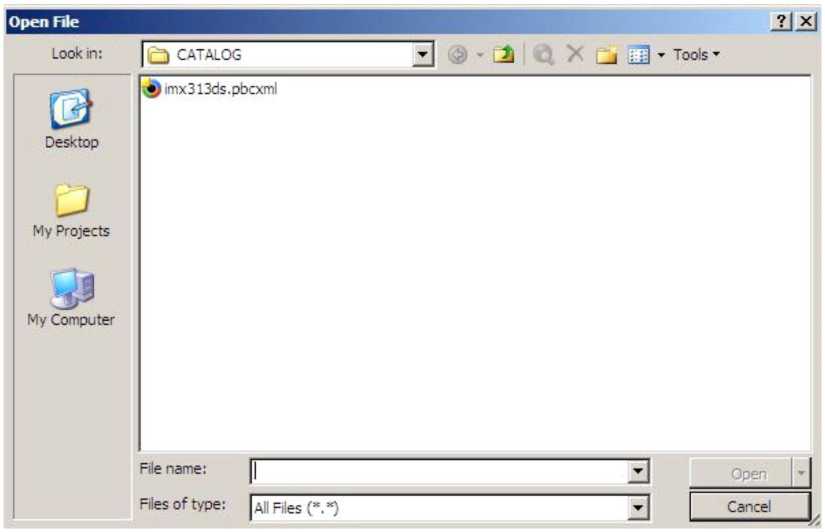


Figure 22. Opening the Catalog File

3. Create a new entry for the GiantPlus display under Catalog > Third Party > BSP > Freescale i.MX31 3DS: ARMV4I > Device Drivers > Display folder. Right click the Display folder and select Add Catalog Item as shown in Figure 23.

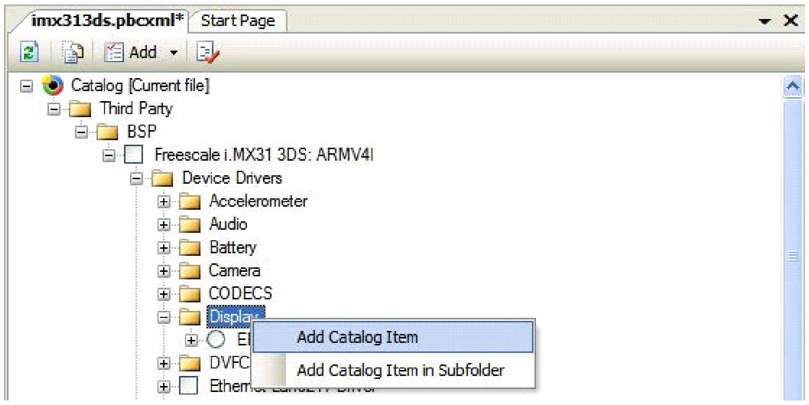


Figure 23. Adding a New Catalog Item

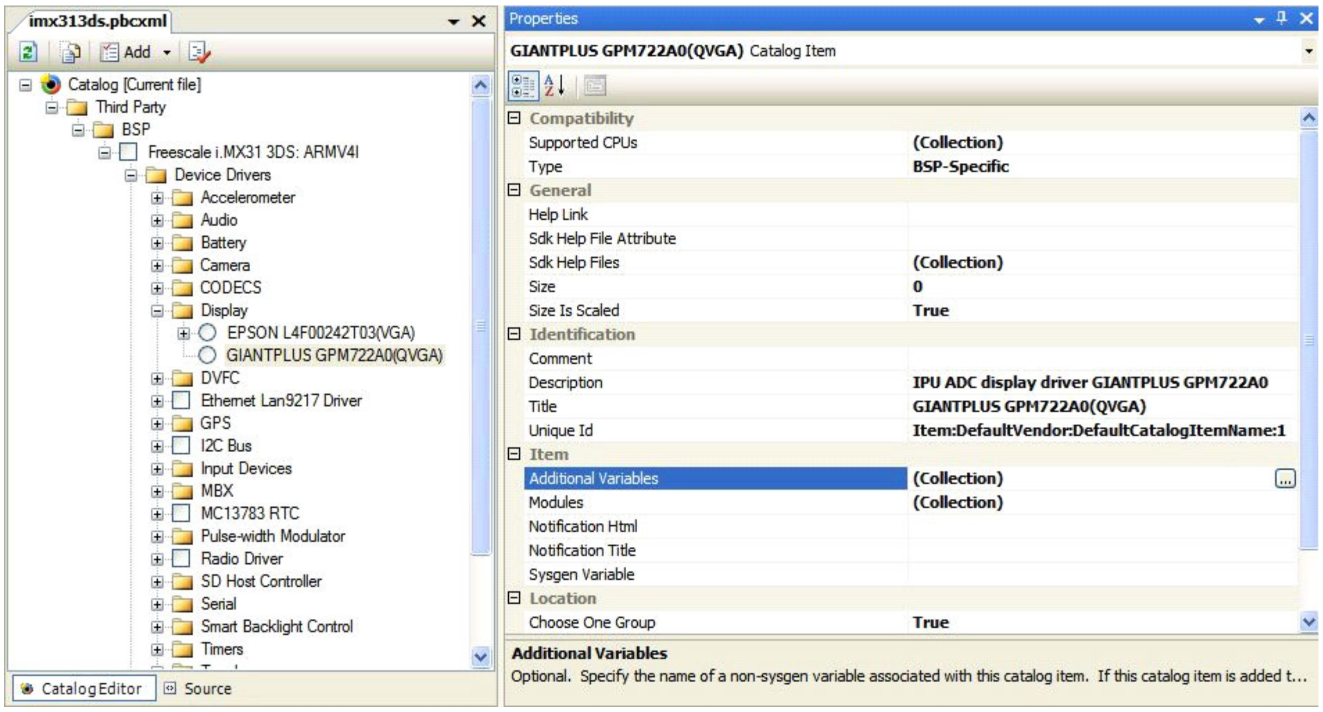
4. Modify the properties of the catalog item as listed in Table 18.

Table 18 shows the modification of the properties of the newly added catalog item.

Table 18. Modifying the Catalog Item Properties

Properties	Value
Description	IPU ADC display driver GIANTPLUS GPM722A0
Title	GIANTPLUS GPM722A0(QVGA)
Additional Variables	BSP_DISPLAY_GIANTPLUS_GPM722A0 BSP_PP
Modules	ddraw_ipu.dll
Choose One Group	True

As shown in Figure 24, all the other properties must have the default value.



imx313ds.pbcxml

Add

- Catalog [Current file]
  - Third Party
    - BSP
      - Freescale i.MX31 3DS: ARMV4I
        - Device Drivers
          - Accelerometer
          - Audio
          - Battery
          - Camera
          - CODECS
          - Display
            - EPSON L4F00242T03(VGA)
            - GIANTPLUS GPM722A0(QVGA)
          - DVFC
          - Ethernet Lan9217 Driver
          - GPS
          - I2C Bus
          - Input Devices
          - MBX
          - MC13783 RTC
          - Pulse-width Modulator
          - Radio Driver
          - SD Host Controller
          - Serial
          - Smart Backlight Control
          - Timers

CatalogEditor | Source

Properties

GIANTPLUS GPM722A0(QVGA) Catalog Item

Compatibility	
Supported CPUs	(Collection)
Type	BSP-Specific
General	
Help Link	
Sdk Help File Attribute	
Sdk Help Files	(Collection)
Size	0
Size Is Scaled	True
Identification	
Comment	
Description	IPU ADC display driver GIANTPLUS GPM722A0
Title	GIANTPLUS GPM722A0(QVGA)
Unique Id	Item:DefaultVendor:DefaultCatalogItemName:1
Item	
Additional Variables	(Collection)
Modules	(Collection)
Notification Html	
Notification Title	
Sysgen Variable	
Location	
Choose One Group	True

Additional Variables
Optional. Specify the name of a non-sysgen variable associated with this catalog item. If this catalog item is added t...

Figure 24. Catalog Item Properties

5. Click OK on variables collection message box. Save and close the file. Open the i.MX313dsmobility project, and then refresh the catalog by using the button on the top of the catalog items view.

When the i.MX313dsmobility project in the Visual Studio is opened again, it is observed that the catalog is changed as the GiantPlus display entry is also available in the catalog. Since the `platform.bib` and `platform.reg` files has to be modified, the GPM722A0 entry is excluded from the build.

Also, comment out the BSP_DISPLAY_EPSON_L4F00242T03 = 1 entry in the `imx313ds.bat` file under the `\WINCE\PLATFORM\iMX313DS` folder because this flag includes the Epson display unconditionally.

```
@REM For Display and Smart backlight support.
@REM set BSP_DISPLAY_EPSON_L4F00242T03=1
```

5.2 Platform.reg

Open the `platform.reg` file and add the register configuration settings for the new panel. Search all the BSP_DISPLAY_EPSON_L4F00242T03 entries and add another OR condition for the GiantPlus display. `HKEY_LOCAL_MACHINE\Drivers\Display\DDIPU` must be unique for each display.

```
;-------------------------------------------------------------------------------
; IPU Display Driver
;
#if (defined BSP_CAMERA || defined BSP_PP || defined BSP_DISPLAY_EPSON_L4F00242T03 || defined
BSP_DISPLAY_GIANTPLUS_GPM722A0 || defined BSP_MBX)
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\IPU_BASE]
         "Prefix"="IPU"
         "Dll"="ipu_base.dll"
         "Order"=dword:1
         "Index"=dword:1
#endif // (defined BSP_CAMERA || defined BSP_PP || defined BSP_DISPLAY_EPSON_L4F00242T03 ||
defined BSP_DISPLAY_GIANTPLUS_GPM722A0 || defined BSP_DISPLAY_SHARP_LQ035Q7DB02 || defined
BSP_MBX)

#if (defined BSP_DISPLAY_EPSON_L4F00242T03 || defined BSP_DISPLAY_GIANTPLUS_GPM722A0 || defined
BSP_DISPLAY_SHARP_LQ035Q7DB02)
[HKEY_LOCAL_MACHINE\System\GDI\Drivers]
         "Display"="ddraw_ipu.dll"
         "Order"=dword:10
...

IF BSP_DISPLAY_GIANTPLUS_GPM722A0
[HKEY_LOCAL_MACHINE\Drivers\Display\DDIPU]

         "Bpp"=dword:10                ; 16bpp
         "VideoBpp"=dword:10           ; RGB565
         "PanelType"=dword:7           ; GiantPlus QVGA smart panel
ENDIF
...
#endif // (defined BSP_DISPLAY_EPSON_L4F00242T03 || defined BSP_DISPLAY_GIANTPLUS_GPM722A0 ||
defined BSP_DISPLAY_SHARP_LQ035Q7DB02)
```

5.3 Platform.bib

The `platform.bib` file determines the files that are to be included in the image. For the display driver, the `ddraw_ipu.dll` and `ipu_base.dll` files has to be included and the `platform.bib` entries must be updated. Also, the BSP_DISPLAY_GIANTPLUS_GPM722A0 variable must be set.

```
IF BSP_NODISPLAY !

#if (defined BSP_DISPLAY_EPSON_L4F00242T03 || defined BSP_DISPLAY_GIANTPLUS_GPM722A0 || defined
BSP_DISPLAY_SHARP_LQ035Q7DB02)
         ddraw_ipu.dll $(_FLATRELEASEDIR)\ddraw_ipu.dll                  NK   SHK
#endif

ENDIF ; BSP_NODISPLAY !
; -----------------------------------------------------------------------------
; IPU Common Driver
;
#if ( defined BSP_CAMERA || defined BSP_PP || defined BSP_DISPLAY_EPSON_L4F00242T03 || defined
BSP_DISPLAY_GIANTPLUS_GPM722A0 || defined BSP_DISPLAY_SHARP_LQ035Q7DB02 || defined BSP_MBX )
         ipu_base.dll    $(_FLATRELEASEDIR)\ipu_base.dll                 NK   SHK
#endif
; -----------------------------------------------------------------------------
```

After the `platform.reg` and `platform.bib` files are modified, perform a build using the current BSP and subprojects.

5.4 Setting Up the Power LCD Voltages

Before connecting any LCD panel to the i.MX31 PDK, it is important to verify if the LCD can be powered with the proper supply voltages and also if the display data interface has the correct value. Power settings are handled by the ATLAS PMIC and during its initialization, the display driver must configure the PMIC settings in order to set up the power voltage configuration. Unlike EPSON display (L4F00242T03), which uses 1.8 V, the GiantPlus display uses 2.8 V (LCD_VIO) for connection. The voltage settings can be modified using the `BSPEnableLCD()` function in the `bspdisplay.cpp` file. The PMIC settings can also be modified using the `pmic_regulator.h` file because the 2.8 V enumeration for `VGEN` has not been declared.

pmic_regulator.h

```
typedef enum _MC13783_REGULATOR_VREG_VOLTAGE_VGEN{
         VGEN_1_20V = 0,    //output  1.20V,
         VGEN_1_30V,         //output   1.30V,
         VGEN_1_50V,         //output   1.50V,
         VGEN_1_80V,         //output   1.80V,
         VGEN_1_10V,         //output   1.10V,
         VGEN_2_00V,         //output   2.00V
         VGEN_2_80V,         //output   2.775V
         VGEN_2_40V,         //output   2.40V,
} MC13783_REGULATOR_VREG_VOLTAGE_VGEN;
```

bspdisplay.cpp

```
BOOL BSPEnableLCD(IPU_DRIVE_TYPE dispType)
{
         ....
         switch (dispType)
         {
                  case eIPU_SDC:
                  ...
                  break;
                  case eIPU_ADC:
                  {
                           PMIC_REGULATOR_VREG_VOLTAGE voltage;
                           PmicVoltageRegulatorOn(VMMC1);
                           PmicVoltageRegulatorOn(VGEN);
                           voltage.vmmc = VMMC_6;  //2.8V
                           PmicVoltageRegulatorSetVoltageLevel(VMMC1, voltage);
                           voltage.vgen = VGEN_2_80V;
                           PmicVoltageRegulatorSetVoltageLevel(VGEN, voltage);
                           Sleep(100);
                  }
                           .....
                  break;
         }
}
```

5.5 Setting Up the Reset Signal

Many synchronous and asynchronous displays require a reset signal and it is recommended that this signal must be controlled by the microprocessor. For GPM722A0, the reset signal is controlled by the i.MX31 GPIO MCU3_24 (LCS1). Due to this reason, during initialization, this pin needs to be configured as GPIO.

The reset pin is configured using the `BSPInitializeLCD()` function in the `bspdisplay.cpp` file:

```
BOOL BSPInitializeLCD(IPU_DRIVE_TYPE dispType)
{
         ...
         switch (dispType)
```

```
        {
                case eIPU_ADC:
                 // Reset pin:LCS1, MCU3_24
                 DDKIomuxSetPinMux(DDK_IOMUX_PIN_LCS1, DDK_IOMUX_OUT_GPIO,
DDK_IOMUX_IN_GPIO);
                 DDKIomuxSetPadConfig(DDK_IOMUX_PAD_LCS1, DDK_IOMUX_PAD_SLEW_SLOW,
DDK_IOMUX_PAD_DRIVE_NORMAL, DDK_IOMUX_PAD_MODE_CMOS, DDK_IOMUX_PAD_TRIG_SCHMITT,
DDK_IOMUX_PAD_PULL_UP_100K);
                 DDKGpioSetConfig(DDK_GPIO_PORT3, LCD_RESET_GPIO_PIN, DDK_GPIO_DIR_OUT,
DDK_GPIO_INTR_NONE);
                 DDKGpioWriteDataPin(DDK_GPIO_PORT3, LCD_RESET_GPIO_PIN, 1);
                 break;
        case eIPU_SDC:
                ...
                break;
}
```

The reset pulse is active low and it is created by setting the `LCD_RESET_GPIO_PIN` variable in the `BSPEnableLCD()` function:

```
BOOL BSPEnableLCD(IPU_DRIVE_TYPE dispType)
{
        ....
        switch (dispType)
        {
                case eIPU_SDC:
                 ...
                break;
                case eIPU_ADC:
                 .....
                 //Reset LCD module, MCU3_24, LOW active
                 DDKGpioWriteDataPin(DDK_GPIO_PORT3, LCD_RESET_GPIO_PIN, 0);
                 Sleep(100);
                 DDKGpioWriteDataPin(DDK_GPIO_PORT3, LCD_RESET_GPIO_PIN, 1);
                 Sleep(100);
                 .....
                break;
        }
}
```

5.6 Setting Up the Backlight

The backlight must be turned ON because the contents on the LCD panel can be viewed only when the backlight is ON. Also, the backlight must be turned ON before continuing with the display driver development.

In this example, initialize the keypad backlight, which is the ATLAS line that handles the GiantPlus backlight in the PDK by using the `BSPEnableLCD()` function:

```
BOOL BSPEnableLCD(IPU_DRIVE_TYPE dispType)
{
        ....
        switch (dispType)
        {
        case eIPU_SDC:
         ...
        break;
        case eIPU_ADC:
         .....
         // Turn-on GPM722A0 backlight through ATLAS KEYPAD LED
         PmicBacklightSetMode(BACKLIGHT_KEYPAD, BACKLIGHT_CURRENT_CTRL_MODE);
         // set to the minimum period
         PmicBacklightSetCycleTime(MC13783_LED_MIN_BACKLIGHT_PERIOD);
         // set the current level for Display
         PmicBacklightSetCurrentLevel(BACKLIGHT_KEYPAD,
MC13783_LED_MAX_BACKLIGHT_CURRENT_LEVEL);
         // set the PWM dutycycle
PmicBacklightSetDutyCycle(BACKLIGHT_KEYPAD,MC13783_LED_MAX_BACKLIGHT_DUTY_CYCLE);
         // turn on Master enable
         PmicBacklightMasterEnable();
         .....
        break;
        }
}
```

5.7 Configuring Wait for the VSYNC Signal

There are some panels, which send a VSYNC signal to synchronize the writing to the panel and avoid tearing. Since this is not applicable for GPM722A0, the panel can be removed from the driver. If the panel is not removed, then the driver waits for the VSYNC signal (that does not appear), after every write operation. If the tearing prevention flag is not removed, the timeout for this signal appears and the refresh rate is extremely slow and limited for the VSYNC timeout (less than one write per second).

The panel is disabled by setting the `BSP_ADC_ENABLE_TEARING_PREVENTION` option in the `bsp_cfg.h` header file to FALSE.

```
#define BSP_ADC_ENABLE_TEARING_PREVENTION FALSE
```

5.8 Adding a Panel Type enum for the New LCD

A new `IPU_PANEL_TYPE` enumeration must be created for this new panel if another LCD panel support is to be added. This enumeration can be found in the `ipu.h` header file. Here, all the asynchronous panel entries must be placed after `ADCPanelOffset`, and the offset after this entry represents the position of the new asynchronous `PANEL_INFO` structure in the `g_ADCPanelArray[]` array in the `adc.c` file.

ipu.h

```
typedef enum {
        IPU_PANEL_SHARP_TFT,        // Registry value is 0
        IPU_PANEL_NEC_TFT,          // Registry value is 1
        IPU_TV_NTSC,                // Registry value is 2
        IPU_TV_PAL,                 // Registry value is 3
        ADCPanelOffset,
        IPU_PANEL_TOSHIBA,          // Registry value is 5
        IPU_PANEL_EPSON,            // Registry value is 6
        IPU_PANEL_GIANTPLUS,        // Registry value is 7
        // New panel goes here ,    // Registry value is 8
        // New panel goes here ,    // Registry value is 9
        numPanel,
} IPU_PANEL_TYPE;
```

5.9 Create the PANEL_INFO Entry for the New LCD

Based on the information described in Section 4.2.4.1, “PANEL_INFO,” fill the `PANEL_INFO` structure for the new asynchronous panel and add it to the `g_ADCPanelArray[]` array in the `adc.c` file. The position of the array for the new LCD panel is determined by the `IPU_PANEL_TYPE` enumeration.

The following code snippet describes the GiantPlus GPM722A0 smart panel:

```
PANEL_INFO g_ADCPanelArray[numPanel] =
{
         // Toshiba Smart Panel Definitions
         {
          ...
         },
         //Epson Smart Panel definitions
         {
          ...
         },
         // GiantPlus GPM76722A0 QVGA Smart Panel Definitions
         {
                  (PUCHAR) "GiantPlus QVGA Panel",// Name
                  IPU_PANEL_GIANTPLUS,          // type
                  IPU_PIX_FMT_RGB565,           // Pixel Format
                  DISPLAY_MODE_DEVICE,          // Mode ID
                  240,                          // width
                  320,                          // height
                  60,                           // frequency (refresh rate)
                  0,                            // Vertical Sync width
                  0,                            // Vertical Start Width 34
                  0,                            // Vertical End Width
                  0,                            // Horizontal Sync Width
                  0,                            // Horizontal Start Width 144
                  0,                            // Horizontal End Width
                  600,                          // Read Cycle Period (600)
                  150,                          // Read Up Position  (100)
                  150,                          // Read Down Position (130)
                  300,                          // Write Cycle Period (100)
                  10,                           // Write Up Position  (10)
                  80,                           // Write Down Position (45)
                  8000000,                      // Pixel Clock Cyle Frequency (8000000)
                  200,                          // Pixel Data Offset Position (200)
                   {        // ADC Display Interface signal polarities
                   IPU_ADC_DISPLAY_0,                           // Display Number
                   IPU_DI_DISP_IF_CONF_IF_MODE_SYSTEM80_TYPE2, // Interface mode
                   IPU_DI_DISP_IF_CONF_PAR_BURST_MODE_BURST_CS,//Parallel interface single
                   IPU_DI_DISP_SIG_POL_DATA_POL_STRAIGHT,      // Data Pol
                   IPU_DI_DISP_SIG_POL_CS_POL_ACTIVE_LOW,      // Clock Select Pol
                   IPU_DI_DISP_SIG_POL_PAR_RS_POL_STRAIGHT,    // Parallel RS Pol
                   IPU_DI_DISP_SIG_POL_WR_POL_ACTIVE_LOW,      // Write Pol
                   IPU_DI_DISP_SIG_POL_RD_POL_ACTIVE_LOW,      // Read Pol
                   IPU_DI_DISP_SIG_POL_VSYNC_POL_ACTIVE_LOW,   // VSync Pol
                   IPU_DI_DISP_SIG_POL_SD_D_POL_STRAIGHT,      // Serial Data Pol
                   IPU_DI_DISP_SIG_POL_SD_CLK_POL_STRAIGHT,    // Serial Interface Clk Pol
                   IPU_DI_DISP_SIG_POL_SER_RS_POL_STRAIGHT,    // Serial Infc Addr bit Pol
                   IPU_DI_DISP_SIG_POL_BCLK_POL_STRAIGHT,      // Burst clock Pol
         },
         {       // SDC Display Interface signal polarities
```

```
                    0,
                    0,
                    0,
                    0,
                    0,
                    0,
                    0,
                    0,
          }
},
}
```

5.10 Configuring the Data and Command System Interface

After reviewing the LCD datasheet, and based on the information in Section 3.5, “System Interface Data/Command Format,” and Section 3.6, “LCD Panels Supported by the i.MX31,” configure the system interface by modifying the DISP0 properties.

For data, the registers used to modify are:

- DI_DISP0_DB0_MAP
- DI_DISP0_DB1_MAP
- DI_DISP0_DB2_MAP
- DI_DISP_ACC_CC

Modify the following registers in the `InitializeADC()` function in the `adc.c` file:

- DI_DISP0_CB0_MAP
- DI_DISP0_CB1_MAP
- DI_DISP0_CB2_MAP
- DI_DISP_ACC_CC

```
UINT32 InitializeADC(PANEL_INFO *currentPanel, int bpp)
{
         ...
         switch (m_SignalPol.DISP_NUM)
         {
          case IPU_ADC_DISPLAY_0:
                  // Data Mapping
                  //... DI_DISP0_B0_MAP
                  OUTREG32(&g_pIPU->DI_DISP0_DB0_MAP,
                           // data offset (BYTE0 - BLUE)
                           // MSB of blue component is located on bit D4
                           CSP_BITFVAL( IPU_DI_DISP0_DB0_MAP_MD00_OFFS0, 4)|
                           CSP_BITFVAL( IPU_DI_DISP0_DB0_MAP_MD00_OFFS1, 0)|
                           CSP_BITFVAL( IPU_DI_DISP0_DB0_MAP_MD00_OFFS2, 0)|
                           // data mapping (BYTE0 - BLUE)
                           // mask 3 bits, since blue component is 5 bits width
                           // M3-M7 are enabled on first clock cycle (0)
                           CSP_BITFVAL( IPU_DI_DISP0_DB0_MAP_MD00_M0, 3)|
                           CSP_BITFVAL( IPU_DI_DISP0_DB0_MAP_MD00_M1, 3)|
                           CSP_BITFVAL( IPU_DI_DISP0_DB0_MAP_MD00_M2, 3)|
                           CSP_BITFVAL( IPU_DI_DISP0_DB0_MAP_MD00_M3, 0)|
                           CSP_BITFVAL( IPU_DI_DISP0_DB0_MAP_MD00_M4, 0)|
                           CSP_BITFVAL( IPU_DI_DISP0_DB0_MAP_MD00_M5, 0)|
                           CSP_BITFVAL( IPU_DI_DISP0_DB0_MAP_MD00_M6, 0)|
                           CSP_BITFVAL( IPU_DI_DISP0_DB0_MAP_MD00_M7, 0));
                  //... DI_DISP0_B1_MAP
                  OUTREG32(&g_pIPU->DI_DISP0_DB1_MAP,
                           // data offset (BYTE1 - GREEN)
                           // MSB of green component is located on bit D10
                           CSP_BITFVAL( IPU_DI_DISP0_DB1_MAP_MD01_OFFS0, 10)|
                           CSP_BITFVAL( IPU_DI_DISP0_DB1_MAP_MD01_OFFS1, 0)|
                           CSP_BITFVAL( IPU_DI_DISP0_DB1_MAP_MD01_OFFS2, 0)|
                           // data mapping (BYTE1 - GREEN)
                           // mask 2 bits, since green component is 6 bits width
                           // M2-M7 are enabled on first clock cycle (0)
                           CSP_BITFVAL( IPU_DI_DISP0_DB1_MAP_MD01_M0, 3)|
                           CSP_BITFVAL( IPU_DI_DISP0_DB1_MAP_MD01_M1, 3)|
                           CSP_BITFVAL( IPU_DI_DISP0_DB1_MAP_MD01_M2, 0)|
                           CSP_BITFVAL( IPU_DI_DISP0_DB1_MAP_MD01_M3, 0)|
                           CSP_BITFVAL( IPU_DI_DISP0_DB1_MAP_MD01_M4, 0)|
                           CSP_BITFVAL( IPU_DI_DISP0_DB1_MAP_MD01_M5, 0)|
                           CSP_BITFVAL( IPU_DI_DISP0_DB1_MAP_MD01_M6, 0)|
                           CSP_BITFVAL( IPU_DI_DISP0_DB1_MAP_MD01_M7, 0));
                  //... DI_DISP0_B2_MA
                  OUTREG32(&g_pIPU->DI_DISP0_DB2_MAP,
                           // data offset (BYTE2 - RED)
                           // MSB of red component is located on bit D15
                           CSP_BITFVAL( IPU_DI_DISP0_DB2_MAP_MD02_OFFS0, 15)|
                           CSP_BITFVAL( IPU_DI_DISP0_DB2_MAP_MD02_OFFS1, 0)|
                           CSP_BITFVAL( IPU_DI_DISP0_DB2_MAP_MD02_OFFS2, 0)|
                           // data mapping (BYTE2 - RED)
                           // mask 3 bits, since red component is 5 bits width
                           // M3-M7 are enabled on first clock cycle (0)
                           CSP_BITFVAL( IPU_DI_DISP0_DB2_MAP_MD02_M0, 3)|
                           CSP_BITFVAL( IPU_DI_DISP0_DB2_MAP_MD02_M1, 3)|
                           CSP_BITFVAL( IPU_DI_DISP0_DB2_MAP_MD02_M2, 3)|
                           CSP_BITFVAL( IPU_DI_DISP0_DB2_MAP_MD02_M3, 0)|
                           CSP_BITFVAL( IPU_DI_DISP0_DB2_MAP_MD02_M4, 0)|
                           CSP_BITFVAL( IPU_DI_DISP0_DB2_MAP_MD02_M5, 0)|
                           CSP_BITFVAL( IPU_DI_DISP0_DB2_MAP_MD02_M6, 0)|
                           CSP_BITFVAL( IPU_DI_DISP0_DB2_MAP_MD02_M7, 0));
                           // Data System Interface is 1 transfer/pixel
         INSREG32BF(&g_pIPU->DI_DISP_ACC_CC,
                           IPU_DI_DISP_ACC_CC_DISP0_IF_CLK_CNT_D,
                           IPU_DI_DISP_ACC_CC_CLOCK_1_CYCLE);
                           // Command Mapping
                           // Enable byte during first clock cycle
         OUTREG32(&g_pIPU->DI_DISP0_CB0_MAP,
                           // command offset first byte DB7-DB0
                           CSP_BITFVAL( IPU_DI_DISP0_CB0_MAP_MC00_OFFS0, 0x7)|
                           CSP_BITFVAL( IPU_DI_DISP0_CB0_MAP_MC00_OFFS1, 0)|
                           CSP_BITFVAL( IPU_DI_DISP0_CB0_MAP_MC00_OFFS2, 0)|
                           // command mapping, enable complete Byte0 in first cycle
                           CSP_BITFVAL( IPU_DI_DISP0_CB0_MAP_MC00_M0, 0)|
                           CSP_BITFVAL( IPU_DI_DISP0_CB0_MAP_MC00_M1, 0)|
                           CSP_BITFVAL( IPU_DI_DISP0_CB0_MAP_MC00_M2, 0)|
                           CSP_BITFVAL( IPU_DI_DISP0_CB0_MAP_MC00_M3, 0)|
                           CSP_BITFVAL( IPU_DI_DISP0_CB0_MAP_MC00_M4, 0)|
                           CSP_BITFVAL( IPU_DI_DISP0_CB0_MAP_MC00_M5, 0)|
                              CSP_BITFVAL( IPU_DI_DISP0_CB0_MAP_MC00_M6, 0)|
                              CSP_BITFVAL( IPU_DI_DISP0_CB0_MAP_MC00_M7, 0));
          //... DI_DISP0_CB1_MAP
          OUTREG32(&g_pIPU->DI_DISP0_CB1_MAP,
                              // command offset, Byte1 DB15-DB8
                              CSP_BITFVAL( IPU_DI_DISP0_CB1_MAP_MC01_OFFS0, 0xF)|
                              CSP_BITFVAL( IPU_DI_DISP0_CB1_MAP_MC01_OFFS1, 0)|
                              CSP_BITFVAL( IPU_DI_DISP0_CB1_MAP_MC01_OFFS2, 0)|
                              // command mapping
                              CSP_BITFVAL( IPU_DI_DISP0_CB1_MAP_MC01_M0, 0)|
                              CSP_BITFVAL( IPU_DI_DISP0_CB1_MAP_MC01_M1, 0)|
                              CSP_BITFVAL( IPU_DI_DISP0_CB1_MAP_MC01_M2, 0)|
                              CSP_BITFVAL( IPU_DI_DISP0_CB1_MAP_MC01_M3, 0)|
                              CSP_BITFVAL( IPU_DI_DISP0_CB1_MAP_MC01_M4, 0)|
                              CSP_BITFVAL( IPU_DI_DISP0_CB1_MAP_MC01_M5, 0)|
                              CSP_BITFVAL( IPU_DI_DISP0_CB1_MAP_MC01_M6, 0)|
                              CSP_BITFVAL( IPU_DI_DISP0_CB1_MAP_MC01_M7, 0));
          //... DI_DISP0_CB2_MAP
          OUTREG32(&g_pIPU->DI_DISP0_CB2_MAP,
                              // command offset
                              CSP_BITFVAL( IPU_DI_DISP0_CB2_MAP_MC02_OFFS0, 0)|
                              CSP_BITFVAL( IPU_DI_DISP0_CB2_MAP_MC02_OFFS1, 0)|
                              CSP_BITFVAL( IPU_DI_DISP0_CB2_MAP_MC02_OFFS2, 0)|
                              // command mapping mask BYTE2, ignore third byte
                              CSP_BITFVAL( IPU_DI_DISP0_CB2_MAP_MC02_M0, 3)|
                              CSP_BITFVAL( IPU_DI_DISP0_CB2_MAP_MC02_M1, 3)|
                              CSP_BITFVAL( IPU_DI_DISP0_CB2_MAP_MC02_M2, 3)|
                              CSP_BITFVAL( IPU_DI_DISP0_CB2_MAP_MC02_M3, 3)|
                              CSP_BITFVAL( IPU_DI_DISP0_CB2_MAP_MC02_M4, 3)|
                              CSP_BITFVAL( IPU_DI_DISP0_CB2_MAP_MC02_M5, 3)|
                              CSP_BITFVAL( IPU_DI_DISP0_CB2_MAP_MC02_M6, 3)|
                              CSP_BITFVAL( IPU_DI_DISP0_CB2_MAP_MC02_M7, 3));
                              // 1 clk cycle per command
                     INSREG32BF(&g_pIPU->DI_DISP_ACC_CC,
                              IPU_DI_DISP_ACC_CC_DISP0_IF_CLK_CNT_C,
                              IPU_DI_DISP_ACC_CC_CLOCK_1_CYCLE);
           break;
case IPU_ADC_DISPLAY_1:
           break;
case IPU_ADC_DISPLAY_2:
           break;
          }
          ...

}
```

5.11 Write Initialization Command Routine

After configuring the command interface, the initialization routine must be included in the `BSPEnableLCD()` function. Since all the write registers have the same endianness, send the 16-bit address first and then the 16-bit data value. A new low-level routine can be created to avoid the usage of the `cmd_data[]` array. The `BSPEnableLCD()` function must be declared in the `adc.h` header file and also in the `adc.c` header file.

adc.h

```
void ADCWriteCommandRegister(UINT32 dispNum, BOOL b_cmd_data, UINT32 regNum, UINT32 regData);
```

adc.c

```
void ADCWriteCommandRegister(UINT32 dispNum, BOOL b_cmd_data, UINT32 regNum, UINT32 regData)
{
         // Write low-level access configuration register
         OUTREG32(&g_pIPU->DI_DISP_LLA_CONF,
                  CSP_BITFVAL(IPU_DI_DISP_LLA_CONF_DRCT_BE_MODE, 0) |
                  CSP_BITFVAL(IPU_DI_DISP_LLA_CONF_DRCT_MAP_DC, 1) |
                  CSP_BITFVAL(IPU_DI_DISP_LLA_CONF_DRCT_LOCK, 0) |
                  CSP_BITFVAL(IPU_DI_DISP_LLA_CONF_DRCT_DISP_NUM, dispNum) |
                  CSP_BITFVAL(IPU_DI_DISP_LLA_CONF_DRCT_RS, b_cmd_data ? 0 : 1));
         // Write low-level access data (send address first)
         OUTREG32(&g_pIPU->DI_DISP_LLA_DATA, regNum);
         // Write low-level access configuration register
         OUTREG32(&g_pIPU->DI_DISP_LLA_CONF,
                  CSP_BITFVAL(IPU_DI_DISP_LLA_CONF_DRCT_BE_MODE, 0) |
                  CSP_BITFVAL(IPU_DI_DISP_LLA_CONF_DRCT_MAP_DC, 1) |
                  CSP_BITFVAL(IPU_DI_DISP_LLA_CONF_DRCT_LOCK, 0) |
                  CSP_BITFVAL(IPU_DI_DISP_LLA_CONF_DRCT_DISP_NUM, dispNum) |
                  CSP_BITFVAL(IPU_DI_DISP_LLA_CONF_DRCT_RS, 1));
         // Write low-level access data (send register address first)
         OUTREG32(&g_pIPU->DI_DISP_LLA_DATA, regData);
}
```

In the following example, the initialization routine is provided by the LCD vendor.

bspdisplay.cpp

```
BOOL BSPEnableLCD(IPU_DRIVE_TYPE dispType)
{
         ....
         switch (dispType)
         {
          case eIPU_SDC:
                  ...
          break;
          case eIPU_ADC:
                  .....
                  UINT32 dispNum = 0; // We are using Display 0
         ADCWriteCommandRegister(dispNum, TRUE, 0x00E5, 0x8000);//????
         ADCWriteCommandRegister(dispNum, TRUE, 0x0000, 0x0001);// start oscilation
         Sleep(10);
         ADCWriteCommandRegister(dispNum, TRUE, 0x0001, 0x0100);//driver S1->S720
         ADCWriteCommandRegister(dispNum, TRUE, 0x0002, 0x0700); //linear control
         ADCWriteCommandRegister(dispNum, TRUE, 0x0003, 0x1030);
         ADCWriteCommandRegister(dispNum, TRUE, 0x0004, 0x0000);//no resizing
         ADCWriteCommandRegister(dispNum, TRUE, 0x0008, 0x0303); //display control2
         ADCWriteCommandRegister(dispNum, TRUE, 0x0009, 0x0000); //display control3
         ADCWriteCommandRegister(dispNum, TRUE, 0x000A, 0x0000); //display control4
         ADCWriteCommandRegister(dispNum, TRUE, 0x000C, 0x0000);//RGB ifc-18bit
         ADCWriteCommandRegister(dispNum, TRUE, 0x000D, 0x0000);//frame mark line0
         ADCWriteCommandRegister(dispNum, TRUE, 0x000F, 0x0000);
```

```
//power on sequence
ADCWriteCommandRegister(dispNum, TRUE, 0x0010, 0x0000); //power1
ADCWriteCommandRegister(dispNum, TRUE, 0x0011, 0x0007); //power2
ADCWriteCommandRegister(dispNum, TRUE, 0x0012, 0x0000); //power3
ADCWriteCommandRegister(dispNum, TRUE, 0x0013, 0x0000); //power4
Sleep(50);
ADCWriteCommandRegister(dispNum, TRUE, 0x0010, 0x1590); //power1
ADCWriteCommandRegister(dispNum, TRUE, 0x0011, 0x0007); //power2
Sleep(50);
// If we are using the suggested 2.8V supply,
// we can use the Giantplus-suggested voltage
// amplification settings.
ADCWriteCommandRegister(dispNum, TRUE, 0x0012, 0x0118); //power3
Sleep(50);
ADCWriteCommandRegister(dispNum, TRUE, 0x0013, 0x1700); //power4
Sleep(50);
ADCWriteCommandRegister(dispNum, TRUE, 0x0029, 0x000E); //power control
ADCWriteCommandRegister(dispNum, TRUE, 0x0020, 0x0000); //GRAM address set
ADCWriteCommandRegister(dispNum, TRUE, 0x0021, 0x0000); //GRAM address set
ADCWriteCommandRegister(dispNum, TRUE, 0x0030, 0x0000); //Gamma control
ADCWriteCommandRegister(dispNum, TRUE, 0x0031, 0x0404); //Gamma control
ADCWriteCommandRegister(dispNum, TRUE, 0x0032, 0x0305); //Gamma control
ADCWriteCommandRegister(dispNum, TRUE, 0x0035, 0x0000); //Gamma control
ADCWriteCommandRegister(dispNum, TRUE, 0x0036, 0x1F00); //Gamma control
ADCWriteCommandRegister(dispNum, TRUE, 0x0037, 0x0505); //Gamma control
ADCWriteCommandRegister(dispNum, TRUE, 0x0038, 0x0203); //Gamma control
ADCWriteCommandRegister(dispNum, TRUE, 0x0039, 0x0707); //Gamma control
ADCWriteCommandRegister(dispNum, TRUE, 0x003C, 0x0400); //Gamma control
ADCWriteCommandRegister(dispNum, TRUE, 0x003D, 0x0012); //Gamma control
ADCWriteCommandRegister(dispNum, TRUE, 0x0050, 0x0000);//hor/ver addr pos
ADCWriteCommandRegister(dispNum, TRUE, 0x0051, 0x00EF);//hor/ver addr pos
ADCWriteCommandRegister(dispNum, TRUE, 0x0052, 0x0000); //hor/ver addr pos
ADCWriteCommandRegister(dispNum, TRUE, 0x0053, 0x013F); //hor/ver addr pos
ADCWriteCommandRegister(dispNum, TRUE, 0x0060, 0x2700);//gate scan control
ADCWriteCommandRegister(dispNum, TRUE, 0x0061, 0x0001);//gate scan control
ADCWriteCommandRegister(dispNum, TRUE, 0x006A, 0x0000);//gate scan control
ADCWriteCommandRegister(dispNum, TRUE, 0x0080, 0x0000); //Partial image
ADCWriteCommandRegister(dispNum, TRUE, 0x0081, 0x0000); //Partial image
ADCWriteCommandRegister(dispNum, TRUE, 0x0082, 0x0000); //Partial image
ADCWriteCommandRegister(dispNum, TRUE, 0x0083, 0x0000); //Partial image
ADCWriteCommandRegister(dispNum, TRUE, 0x0084, 0x0000); //Partial image
ADCWriteCommandRegister(dispNum, TRUE, 0x0085, 0x0000); //Partial image
ADCWriteCommandRegister(dispNum, TRUE, 0x0090, 0x0014); //panel ifc 20 clk
ADCWriteCommandRegister(dispNum, TRUE, 0x0092, 0x0000);
ADCWriteCommandRegister(dispNum, TRUE, 0x0093, 0x0000);
ADCWriteCommandRegister(dispNum, TRUE, 0x0095, 0x0110);
ADCWriteCommandRegister(dispNum, TRUE, 0x0097, 0x0000);
ADCWriteCommandRegister(dispNum, TRUE, 0x0098, 0x0000);
ADCWriteCommandRegister(dispNum, TRUE, 0x0007, 0x0133); //display control1
// Write test pattern to display
ADCWriteCommandRegister(dispNum, TRUE, 0x0020, 0x0000); //GRAM address set
ADCWriteCommandRegister(dispNum, TRUE, 0x0021, 0x0000); //GRAM address set
Sleep(1);
int i;
//blue and green strips
for(i=0; i<(240*320/8-1); i++)
{
```

```
                 ADCWriteCommandRegister(dispNum, TRUE, 0x0022, 0x001F); //Write pixel
                 ADCWriteCommandRegister(dispNum, TRUE, 0x0022, 0x001F); //Write pixel
                 ADCWriteCommandRegister(dispNum, TRUE, 0x0022, 0x001F); //Write pixel
                 ADCWriteCommandRegister(dispNum, TRUE, 0x0022, 0x001F); //Write pixel
                 ADCWriteCommandRegister(dispNum, TRUE, 0x0022, 0x07E0); //Write pixel
                 ADCWriteCommandRegister(dispNum, TRUE, 0x0022, 0x07E0); //Write pixel
                 ADCWriteCommandRegister(dispNum, TRUE, 0x0022, 0x07E0); //Write pixel
                 ADCWriteCommandRegister(dispNum, TRUE, 0x0022, 0x07E0); //Write pixel
        }
        Sleep(2000);
        .....
        break;
        }
}
```

5.12 Creating the Frame Buffer Template for Write Access

Green and blue strips can be found on the screen, if all the steps are executed. With the help of this image, ensure that the panel is initialized correctly and also the backlight is enabled. As mentioned in Section 3.7.2, “Template Mode,” create a template to enable the display driver in order to write on the frame buffer. This template is defined in the `CreateTemplate()` function of the `adc.c` file.

```
//      Register name          Reg No.    R/W  RS  Description
//----- --------------------- -------    --- --- --------------------
#define HOR_GRAM_ADDR_SET       0x20   //  W   1  Horizontal GRAM Address Set
#define VER_GRAM_ADDR_SET       0x21   //  W   1  Vertical GRAM Address Set
#define WR_DATA_TO_GRAM         0x22   //  W   1  Write Data to GRAM
void CreateTemplate(TEMPLATE_CMD_REG *pCmd, unsigned int width, unsigned int height)
{
        unsigned int i = 0;
        UNREFERENCED_PARAMETER(width);
        UNREFERENCED_PARAMETER(height);
        // Create the template for GiantPlus LCD GPM722A0
         pCmd[i].reg.Data = HOR_GRAM_ADDR_SET;  // X coordinate command
         pCmd[i].reg.Opcode = WR_CMND;
         pCmd[i].reg.RS = 0;
         pCmd[i++].reg.FlowControl = SINGLE_STEP;
         pCmd[i].reg.Data = 0x01;               // send address bits [7:0]
         pCmd[i].reg.Opcode = WR_XADDR;
         pCmd[i].reg.RS = 1;
         pCmd[i++].reg.FlowControl = SINGLE_STEP;
         pCmd[i].reg.Data = VER_GRAM_ADDR_SET;  // Y coordinate command
         pCmd[i].reg.Opcode = WR_CMND;
         pCmd[i].reg.RS = 0;
         pCmd[i++].reg.FlowControl = SINGLE_STEP;
         pCmd[i].reg.Data = 0x01;               // send address bits [22:8]
         pCmd[i].reg.Opcode = WR_YADDR;
         pCmd[i].reg.RS = 1;
         pCmd[i++].reg.FlowControl = SINGLE_STEP;
         pCmd[i].reg.Data = WR_DATA_TO_GRAM;     // Write Data to GRAM
         pCmd[i].reg.Opcode = WR_CMND;
         pCmd[i].reg.RS = 0;
         pCmd[i++].reg.FlowControl = SINGLE_STEP;
         pCmd[i].reg.Data = 0;
         pCmd[i].reg.Opcode = WR_DATA;
         pCmd[i].reg.RS = 1;
         pCmd[i++].reg.FlowControl = STOP;
}
```

`UNREFERENCED_PARAMETER` tag is used to describe the following error:

```
error C2220: warning treated as error - no 'object' file generated
warning C4100: 'height' : unreferenced formal parameter
warning C4100: 'width' : unreferenced formal parameter
```

5.13 Adding the Mouse Support for the Asynchronous Displays

In some cases where the mouse is moving, it is necessary to inform the asynchronous thread to update the region where the cursor is changing. For this reason, it is necessary to add the mouse support feature to the `ddipu_cursor.cpp` file.

ddipu_cursor.cpp

```
SCODE DDIPU::MovePointer(int x, int y)
{
        ...
        CursorOff();
        if (x != -1 || y != -1)
        {
         ...
         // handle the TV mode case for updates
         if(m_bTVModeActive)
         {
                 ...
         }
         // handle ADC (Smart Panels) updates
         if(m_bADCActive)
         {
                 // Now, calculate the dirty-rect to refresh to the actual hardware
                 bounds.left     = m_CursorRect.left;
                 bounds.top      = m_CursorRect.top;
                 bounds.right    = m_CursorRect.right;
                 bounds.bottom   = m_CursorRect.bottom;
        EnterCriticalSection(&m_csDirtyRect);
        DEBUGCHK(m_pTVDirtyRect != NULL);
        if(m_pADCDirtyRect)
        {
         m_pADCDirtyRect->SetDirtyRegion((LPRECT)&oldBounds);
         m_pADCDirtyRect->SetDirtyRegion((LPRECT)&bounds);
        }
        // Signal to ADC thread to udpate the screen.
        SetEvent(m_hADCUpdateRequest);
        LeaveCriticalSection(&m_csDirtyRect);
        }
        }
        return    S_OK;
}
```

5.14 Rebuilding the Project

In the end, rebuild the project by using the Build current BSP and subprojects.

Go to Menu Build > Advanced Build Commands > Build Current BSP and Subprojects to rebuild the project.

6 Revision History

Table 19 provides the revision history for this application note.

Table 19. Document Revision History

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

Document Number: AN4180
Rev. 0
08/2010