Advanced Logic Synthesis for Electronics
www.ALSE-FR.com

A.L.S.E Application Note
Using the LT24 / ILI9341

Introduction

I got questioned a few days ago by an EE teacher who purchased the $39 Terasic **LT24 240 x 320 LCD** display extension to FPGA kits, and didn't understand how to use it. At ALSE, we have designed tens of interfaces for various LCD displays. Altera also offers a parametrizable Qsys module (CVO) that can accommodate most video outputs (parallel RGB + Hsync + Vsync + Pixel Clock), and I suspected it should be straightforward to drive the LT24, so I took a look at the datasheets involved and decided to dedicate a few hours of my Week-End to this matter.

The LT24 LCD array is controlled by an autonomous controller: the **ILITEK ILI9341**, which embarks a graphic memory array and can autonomously display and refresh the LCD array. However, the ILI9341 does not embed an internal graphic processor and it must receive all the pixels values (through various possible interfaces, either serial or parallel). This controller is very popular for older mobile phones and now small micro-controllers use, because of the display autonomous behavior (no need to continuously refresh) and the ability to drive it serially with very few signals (starting with 2 !).
While reading the controller's documentation, I noticed the "RGB mode" which is the usual "Hsync/Vsync" interface and thought it would be the easiest mode to use.
Very unfortunately, Terasic hasn't wired HSync-Vsync-DotClock signals, so the direct RGB mode cannot be used, and the **parallel "16-bits CPU interface"** must be used (this mode is hardwired).

Note that no serial interface has been wired either, but there is nothing to regret here : a complete HE10-40 connector is used by the display anyway, and the 16 bits parallel interface retained is the fastest (and is also very simple to implement in an FPGA). This link can allow the transfer of up to 10 Mbytes/s, fast enough to redraw the entire screen in just 10 ms (ie up to 100 times per second).

So it seemed that, even if some code had to be written (to drive the parallel interface), it would be straightforward... In reality, starting from scratch and displaying patterns on the screen is still quite a challenge, as can be witnessed in several forums :

1) The controller is quite complex, it has 48 Level 1 commands, 36 Level 2 commands, and the documentation (245 pages Manual) contains no practical examples.
2) Setting up the controller correctly is very challenging. The documentation does not provide a "typical initialization sequence". And the LT24 requires non-default settings to work !
3) I don't think there is *anything* useful in the Terasic "examples" nor "documentation". I recommend to completely ignore them and use this ApNote and the accompanying code to build your own solutions. The only useful data is the pin assignment to the HE-10 connector and the ILI9341.pdf Controller's Manual (make sure you don't use a preliminary version from the Internet).
4) I didn't find straight answers using Google. However, you'll see that C graphics libraries have been developed for this controller, and should be easy to port to your FPGA provided you decide to use a Nios II embedded processor, master the initialization sequence, and design the parallel interface.

About A.L.S.E

As of end 2016, ALSE is a 23 years old company consisting in 7 Expert Engineers (Hardware and Software), and being a valued Intel-FPGA partner, Certified Design Center and Certified Training Center.

ALSE is located in Paris France and offers a wide range of Digital Design Services, IPs and Training Courses, serving customers in France, in Europe and all around the world.

- **Design Services** : turnkey designs, complex hardware functions, hardware acceleration, System and Board Level Design, Audits, Methodology assistance…
- **Intellectual Property blocks** (IPs) including **1G** and **10G Ethernet** communication (hardware stacks), numerous high-performance **Memory Controllers**, **Video** & Im**age Compression & Decompression** (**JPEG**, **Wavelet**), **Video** IP blocks, **Security**, **Hardware GZIP**, **Industrial Networks**...
- **Training Services** : all the EE languages and Methodologies including Digital Design, VHDL, Verilog, SystemVerilog, SystemC, TLM2.0, UVM, PSL, Tcl/Tk etc…
- All the **Intel-FPGA training courses**.
- **Embedded Systems** Design Services and Training courses.

Please visit our web sites (in English and in French).

Principles of the LT24 Reference Designs

To debug the screen initialization and investigate how the display can be driven, I started with the design of a **Hardware Reference Design** : a simple HDL module based on a State Machine that parses a list of (easily modifiable) commands & data, and sends them to the LT24 with optional delays between commands (a requirement after some commands).
This list can be edited quickly and tried on the screen after a 30 seconds compilation and download.

This design can serve as a base for any application that does not use an embedded processor.

For applications that use a processor, the solution is also very straightforward :

- Add a 22 bits PIO (output only) to drive all the LT24 signals
- The required functions **`SendCmd`**`(alt_u8 cmd)` and **`SendData`**`(alt_u16 data)` are trivial to write. They are so small that they can be inlined to further speed up the software “driver”.
- Use the Initialization sequence that you have validated using this ApNote.
- Use an existing ILI9341 C library (Atmel, Arduino, Adafruit etc)

So after having experimented with the HDL solution, I have built a “**Software Reference Design**”.

Our Reference Designs (Hardware and Software) are described here and they are based on an existing Max10 FPGA kit, which is not in the list of the supported boards by Terasic. It was not difficult to connect the LT24 to this board, but the pin assignment had to be entered carefully. The entire Hardware design fits in just 145 Logic Elements !

The Hardware design was tested by simulation with ModelSim Intel Edition before testing on the board.

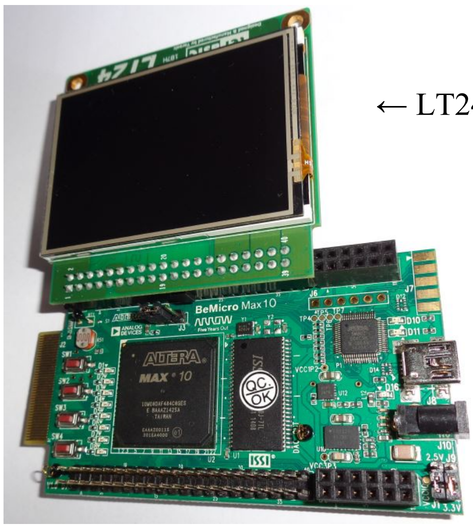
The Software design was tested using the NiosII debugger but worked directly.

Both designs used only the **free** Intel-FPGA tools : Quartus Lite v16.1 and NiosII Software Build Tool (and indeed the free NiosII/e processor).

Pin Assignment

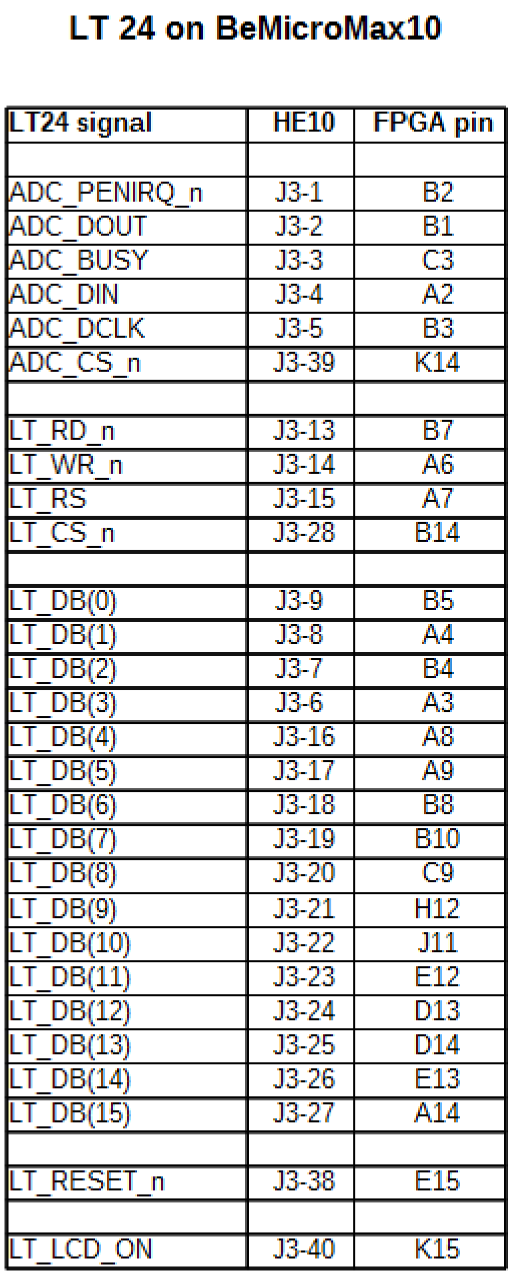
To test our design, we used the small and cheap Max10 kit from Arrow : the **BeMicroMax10**.

The LT24 must be plugged on J3 (top HE10-40) :



See the pin assignment on the right : →

IMPORTANT: the ILI9341 must be driven using 2.5V or 3.3V (max) signals ! This is perfect for FPGA boards like ours but some micro-controllers still use +5V I/Os and may require require level shifters.



LT 24 on BeMicroMax10

LT24 signal	HE10	FPGA pin
ADC_PENIRQ_n	J3-1	B2
ADC_DOUT	J3-2	B1
ADC_BUSY	J3-3	C3
ADC_DIN	J3-4	A2
ADC_DCLK	J3-5	B3
ADC_CS_n	J3-39	K14
LT_RD_n	J3-13	B7
LT_WR_n	J3-14	A6
LT_RS	J3-15	A7
LT_CS_n	J3-28	B14
LT_DB(0)	J3-9	B5
LT_DB(1)	J3-8	A4
LT_DB(2)	J3-7	B4
LT_DB(3)	J3-6	A3
LT_DB(4)	J3-16	A8
LT_DB(5)	J3-17	A9
LT_DB(6)	J3-18	B8
LT_DB(7)	J3-19	B10
LT_DB(8)	J3-20	C9
LT_DB(9)	J3-21	H12
LT_DB(10)	J3-22	J11
LT_DB(11)	J3-23	E12
LT_DB(12)	J3-24	D13
LT_DB(13)	J3-25	D14
LT_DB(14)	J3-26	E13
LT_DB(15)	J3-27	A14
LT_RESET_n	J3-38	E15
LT_LCD_ON	J3-40	K15

Top Level Interface to LT24

We have added the LT24 signals to the FPGA top level:

```
    -- LT24 graphic Controller
    LT_DB          : inout std_logic_vector(15 downto 0); -- we use it as OUT only
    LT_WR_n        : out   std_logic;  -- aka WRX
    LT_RD_n        : out   std_logic;  -- aka RDX
    LT_RS          : out   std_logic;  -- aka D/CX
    LT_CS_n        : out   std_logic;  -- aka CSX
    LT_RESET_n     : out   std_logic;  -- aka RESX
    LT_LCD_ON      : out   std_logic   -- Transistor to drive LCD lighting (1 = on)
```

Note that we drive one LED with the ADC_PEN_IRQ output from the TouchScreen logic, but we have left the touch screen logic outside this ApNote.

Physical Interface

The ILI9341is very versatile and even parallel interfaces can be 8-9-16 or even 18 bits wide ! The LT24 hardwired selection in the LT24 imposes a ***16 bits 8080 type I*** interface. Some configuration registers also need to be correctly programmed to take this into account.

A State Machine (FSM) handles the communication with the ILI9341 Controller.

The ILI9341 Manual explains the required timing for this interface. We can note that raising (inactivating) the Chip Select signal is NOT necessary (Tchw=0). We take CSX low at the beginning of the session and raise it when the FSM has finished.

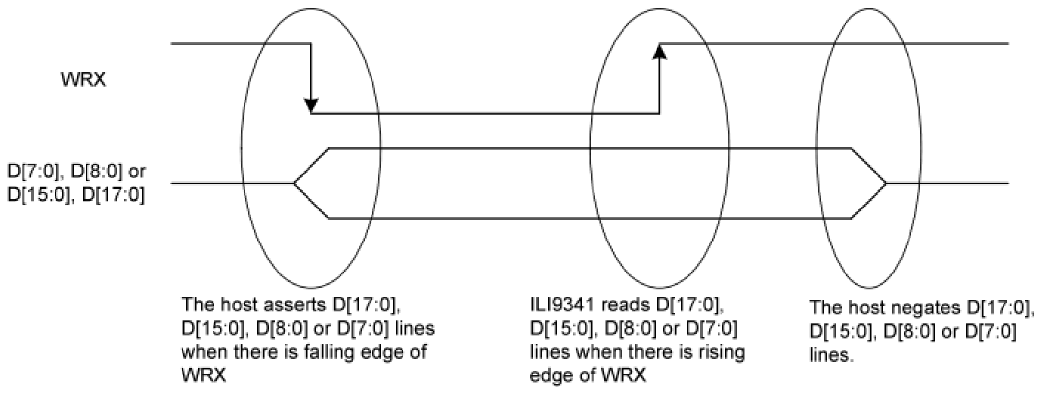
To Send a Command or Data :

- Set LT_RS (aka D/CX) to '0' (for command) or '1' (for Data)
  Set LT_DB to the Command (8 LSBs) or Data (16 bits) value
- Set LT_WR_n = '0' (active)
- Wait Nsetup system clock cycles
- Set LT_WR_n = '1' (going active triggers the actual writing)
- Wait Nhold system clock cycles

It's done through 3 states in the FSM, and our system clock beats at 50 MHz (20 ns).

The principle is documented in the Controller's Manual :

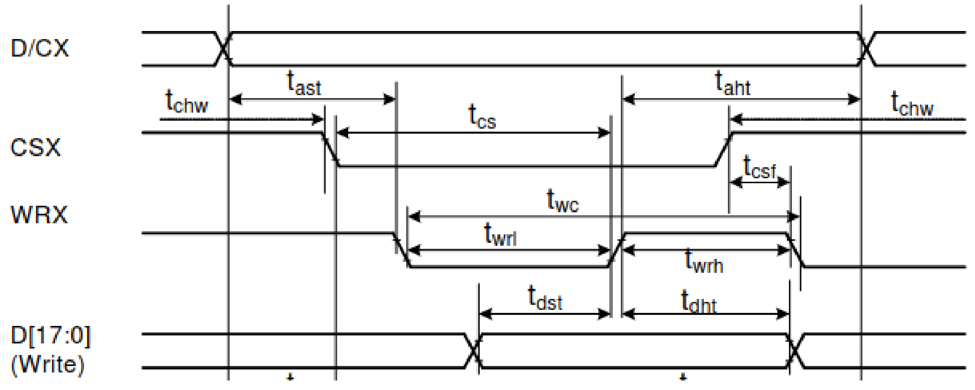
The following figure shows a write cycle for the 8080- I MCU interface.


WRX
D[7:0], D[8:0] or D[15:0], D[17:0]
The host asserts D[17:0], D[15:0], D[8:0] or D[7:0] lines when there is falling edge of WRX
ILI9341 reads D[17:0], D[15:0], D[8:0] or D[7:0] lines when there is rising edge of WRX
The host negates D[17:0], D[15:0], D[8:0] or D[7:0] lines.

And at the end of the manual, we find the complete timing diagram :

18.3 AC Characteristics

18.3.1 Display Parallel 18/16/9/8-bit Interface Timing Characteristics (8080- I system)



The timing values can be found under the waveform on the same page.

Initialization Sequence

The pattern is correctly displayed after the following (*minimal*) initialization :

- Hardware Reset (active for 15 ms) followed by 300 ms wait (conservative delay values)
- Exit Sleep command (0x11) followed by 800 ms delay (delay to see the effect of the next command)
- Display ON (0x29) followed by 1 ms wait
- Set COLMOD (0x3A) with parameter 0x0055 (16 bits per pixel, one data per pixel)
- SET Memory Access Control (0x36) with BGR filter !
- Enable 3G (0xF2) with 0x0000 parameter (disable 3 Gamma)

More settings can be experimented with, once the image is displayed as assured by the above sequence. You can review and test the (much longer) initialization sequences that can be found in various software libraries.

It is critical to set up the mode where one Pixel is stored by one unique 16-bits write, since the screen and the graphic memory are actually 18 bits (RGB 6-6-6) and some modes require several data for one pixel.

This sequence has been developed using the Hardware design and was used as is in the Software design.

Hardware Project

Principle

There is no processor involved in this Hardware project : the "driver" is a Finite State Machine written in VHDL which generates the timings described on the previous page.

This module could be used in a software project but we'll see that a simple PIO and two trivial functions do the job nicely, so this Hardware solution will be reserved to projects where no processor will be present.

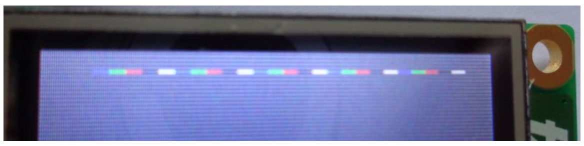
Pattern with the HDL design

To exercise an important feature, we set up the Starting Column Address and Page address. And we verify that the pattern starts inside the screen and not at 0,0.

Then we issue the Memory Write Command (0x2C) followed by the pixel values.

The pattern itself is 8 White pixels, 8 Black, 8 Red, 8 Green and 8 Blue, repeated 5 times (200 pixels total).

It is terminated by a NoOp command (no effect).



The (upside-down) picture above shows the displayed pattern (right to left) matching the vectors.

HDL Design Files

The most interesting file of the hardware project is the top-level **top.vhd** which contains all the logic except the Frequency divider that creates the 1 millisecond tick used for the various delays. The important parts are reproduced below, but you can contact ALSE and obtain this file.

The table of vectors is easy to edit :

```
  subtype Cmd_t is std_logic_vector (28 downto 0);
  -- Dly_ms(12) & C/D & Data(16) - 29 bits, could be a record
  type Commands_t is array (natural range <>) of Cmd_t;

  constant Commands_c : Commands_t := (

--  delay_ms D/Cd_n  DB
    x"001" & '0' & x"0011",  -- Exit Sleep
--  x"100" & '0' & x"0001",  -- Software reset
    x"800" & '0' & x"0029",  -- Display ON

    x"001" & '0' & x"003A",  --  Set COLMOD
    x"000" & '1' & x"0055",  --  non-default 16 bits

    x"000" & '0' & x"0036",  --  Memory Access Control
    x"000" & '1' & x"0008",  --  non-default BGR filter !

    x"000" & '0' & x"00F2",  --  Enable 3G
    x"000" & '1' & x"0000",  --  non-default = disable 3 gamma

    x"000" & '0' & x"002A",  -- Set Column address
    x"000" & '1' & x"0000",  --
    x"000" & '1' & x"0010",  -- 16

    x"000" & '0' & x"002B",  -- Set Page address
    x"000" & '1' & x"0000",  --
    x"000" & '1' & x"000A",  -- 10

    x"000" & '0' & x"002C",  -- Memory Write

    x"000" & '1' & x"FFFF", -- WHITE
    x"000" & '1' & x"FFFF", -- WHITE
    x"000" & '1' & x"FFFF", -- WHITE
    x"000" & '1' & x"FFFF", -- WHITE
   ETC ETC ETC
```

The State Machine reproduced below holds the parsing of the table above and the communication logic.

```
-- --------------------------
-- ILI9341 interface
-- --------------------------
-- CSX can be kept low
-- Write cycle > 66ns   = 100 ns (5c = 2 + 3) can work
-- Tdst data setup > 10 ns
-- Tdht data hold > 10 ns
-- /!\ Data read (not used here) is SLOW (500 ns)

LT_RD_n  <= '1'; -- we don't read

process (Clk,Rst)
begin
  if Rst='1' then
    State <= Boot;
    LT_DBi <= (others=>'0');
    LT_WR_n <= '1';
    LT_CS_n  <= '1';
    LT_RS   <= '0';
    LT_RESET_n <= '0';
    LT_LCD_ON  <= '0';
    Index <= 0;
    Cntr  <= 15;
    Cycle <= 0;
    Cmd   <= (others=>'0');


  elsif rising_edge(Clk) then
```

```
    Cmd <= Commands(Index);

    case State is

      when Boot =>
        LT_RESET_n <= '0';
        Index <= 0;
        if Tick1ms='1' then
          if Cntr=0 then
            LT_RESET_n <= '1';
            Cntr <= 300;
            State <= Idle;
          else
            Cntr <= Cntr - 1;
          end if;
        end if;

      when Idle =>
        if Cntr=0 then
          LT_LCD_ON <= '1'; -- Light ON
          LT_CS_n <= '0';
          State <= GetCmd;
        elsif Tick1ms='1' then
          Cntr <= Cntr - 1;
        end if;

      when GetCmd =>
        Cntr   <= to_integer(unsigned(Cmd(Cmd_t'high downto Cmd'high-11)));
        LT_DBi <= Cmd(LT_DBi'range);
        LT_RS  <= Cmd(LT_DBi'length);
        State  <= Dly;

      when Dly =>
        if Cntr = 0 then
          Cycle <= Nsetup-1;
          LT_WR_n <= '0';
          State <= Wr1;
        elsif Tick1ms='1' then
          Cntr <= Cntr-1;
        end if;

      when Wr1 =>  -- Setup
        if Cycle = 0 then
          Cycle <= Nhold-1;
          LT_WR_n <= '1'; -- rising edge (trig write)
          State <= Wr2;
        else
          Cycle <= Cycle-1;
        end if;

      when Wr2 =>  -- Hold
        if Cycle = 0 then
          State <= Done;
        else
          Cycle <= Cycle-1;
        end if;

      when Done =>
        if Index=Commands'length-1 then  -- last command has been sent
          LT_CS_n <= '1'; -- de-select the interface
          State <= Done;  -- deadlock (default)
        else
          Index <= Index + 1; -- fecth next vector
          State <= Done1;
        end if;

      when Done1 =>  -- compensate double pipeline
          State <= GetCmd;

    end case;

  end if;
end process;
```

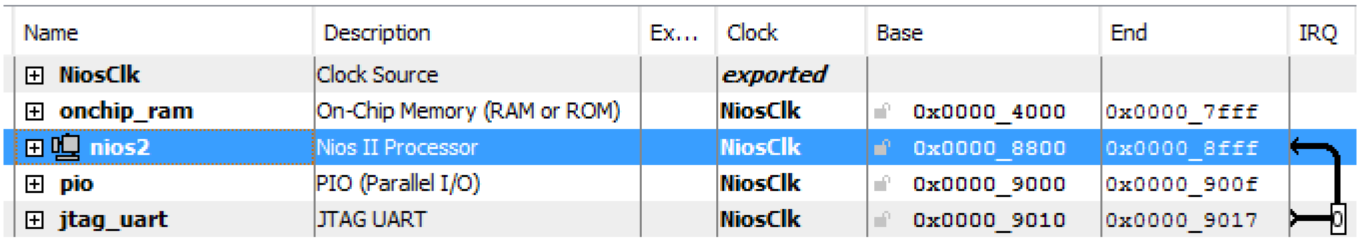
As we can see, interfacing to the Controller in hardware requires less than one page of code !
We'll see it's even simpler in software.

Software Project

Building the System

In just a few minutes, we build a System On Programmable Chip *System* using Qsys in which we place :

- A **clock** source (compulsory)
- A **Nios II / e** 32 bits RISC processor, with the simple **JTag debugger** active
- An 16k bytes Internal Memory block "**On_chip_RAM**" (used for NiosII Data and Program)
- A **22 bits PIO** used for Write only.
- A **JTag UART** to issue printf (not really useful !)



Name	Description	Ex...	Clock	Base	End	IRQ
⊞ **NiosClk**	Clock Source		*exported*			
⊞ **onchip_ram**	On-Chip Memory (RAM or ROM)		**NiosClk**	0x0000_4000	0x0000_7fff	
⊞ **nios2**	Nios II Processor		**NiosClk**	0x0000_8800	0x0000_8fff	
⊞ **pio**	PIO (Parallel I/O)		**NiosClk**	0x0000_9000	0x0000_900f	
⊞ **jtag_uart**	JTAG UART		**NiosClk**	0x0000_9010	0x0000_9017	0

This generates the following simple component :

```
component Nios is
  port ( clk_clk       : in  std_logic;
         reset_reset_n : in  std_logic;
         pio_export    : out std_logic_vector(21 downto 0) );  -- export
end component Nios;
```

We insert this component in our top level (we re-use the Hardware project's top level and remove the HDL controller)

```
sopc : Nios port map (
     clk_clk       => Clk,
     reset_reset_n => Reset_n,
     pio_export    => PIO_D    );
```

and we connect the PIO to the LT24 signals :

```
LT_DB <= PIO_D(15 downto 0);
LT_LCD_ON    <= PIO_D(21);
LT_RESET_n   <= PIO_D(20);
LT_CS_n      <= PIO_D(19);
LT_WR_n      <= PIO_D(18);
LT_RD_n      <= '1';  -- PIO_D(17) make sure we do NOT read !
LT_RS        <= PIO_D(16);
```

By safety, I have inhibited the read line since we drive the databus without a tristate (bidir).

The hardware system is indeed much bigger than the Hardware design, but it does include a processor and still occupies only a fraction (~ 1,700 LEs) of the small Max10-08 device !
It could be further reduced by removing the JTag UART (and the debugger).

NOTE !
For a better job, I could have added : the *SystemID* block (as we recommend in all our training courses !) and a *High-Resolution Timer* if we wanted to do some high precision profiling of the graphic interface.

Writing the Software

I used the free *"NiosII Software Build Tools for Eclipse"* to create the Board Support Package and write our software driver and application. Note that I am not an Embedded nor a Software specialist, so I just took the easiest path to create this Software Solution. Even for me, creating and programming the system is a really simple process.

A nice trick is to start from the *"Hello World Small"* template, since it does directly include the optimal settings to create very compact code for simple applications (we want to stay in the internal memory for this demonstrator and leave room for a library and graphic application code).

LT24 signals

We will drive the LT24 signals simply by writing in our PIO, with the signals assigned as described in the previous page : LCD_ON | RESETn | CSn | WRn | RDn_D/Cn.
We use constants to make the code easier to read.

Constants

```
#define WHITE       0xFFFF
#define BLACK       0x0000
#define RED         0xF800
#define GREEN       0x07E0
#define BLUE        0x001F

// LCD_ON | RESETn | CSn | WRn | RDn_D/Cn
// WriteCommand : 1_1_0_0_1_0 -> 1_1_0_1_1_0
// WriteData    : 1_1_0_0_1_1 -> 1_1_0_1_1_1
#define LT24_ResetS   (0x27 << 16)
#define LT24_ResetE   (0x37 << 16)
#define LT24_CmdS     (0x32 << 16)
#define LT24_CmdE     (0x36 << 16)
#define LT24_DataS    (0x33 << 16)
#define LT24_DataE    (0x37 << 16)

#define Exit_Sleep    0x11
#define Display_ON    0x29
#define COLMODE       0x3A
#define MemAccessCtr  0x36
#define Enable3G      0xF2
#define SetColAddr    0x2A
#define SetPageAddr   0x2B
#define MemWrite      0x2C
```

Functions

The two *inlined* functions are doing the job of sending Data and Commands to the ILI9341 Controller, and PrintBar() does display a pattern much like in the hardware project :

```
void inline SendCmd (alt_u8 cmd) {
      IOWR_ALTERA_AVALON_PIO_DATA (PIO_BASE, LT24_CmdS | cmd);
      IOWR_ALTERA_AVALON_PIO_DATA (PIO_BASE, LT24_CmdE | cmd);
}
void inline SendData (alt_u16 data) {
      IOWR_ALTERA_AVALON_PIO_DATA (PIO_BASE, LT24_DataS | data);
      IOWR_ALTERA_AVALON_PIO_DATA (PIO_BASE, LT24_DataE | data);
}
void PrintBar (void) {
      SendCmd(MemWrite);
      for (int i=0; i<32; i++) SendData(WHITE);
      for (int i=0; i<32; i++) SendData(BLACK);
      for (int i=0; i<32; i++) SendData(RED);
      for (int i=0; i<32; i++) SendData(GREEN);
      for (int i=0; i<32; i++) SendData(BLUE);
}
```

main Program

The main program is extremely simple and the complete demo is compact (450 words) :

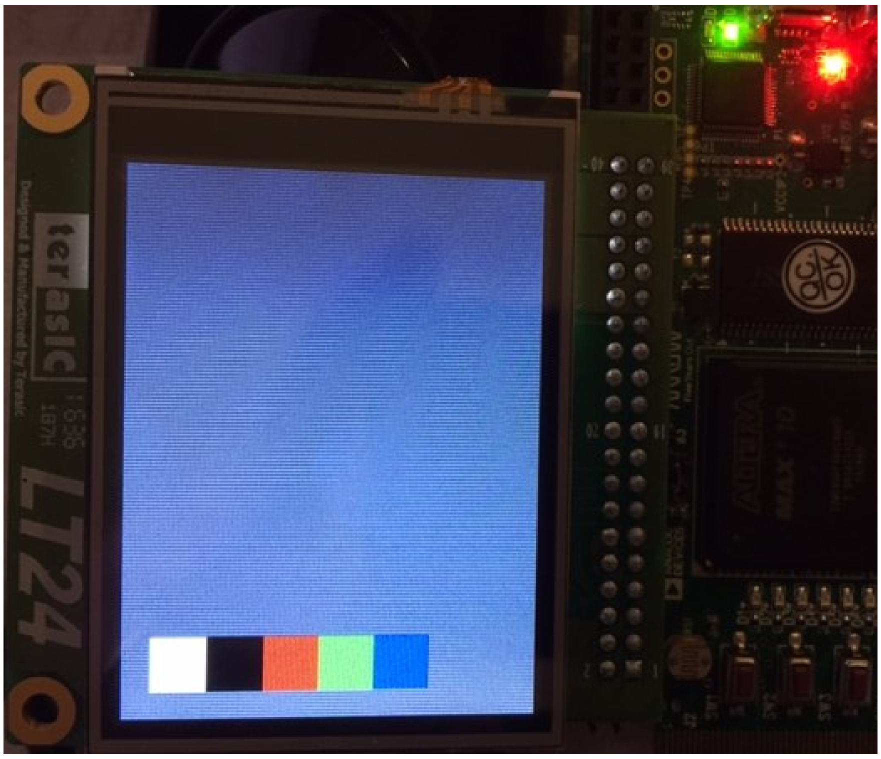
```
Info: (LT24.elf) 1804 Bytes program size (code + initialized data).
Info:            14 KBytes free for stack + heap.
```

```
int main()
{
  alt_putstr("LT24 Demo by ALSE !\n");
  IOWR_ALTERA_AVALON_PIO_DATA (PIO_BASE,LT24_ResetS);
  usleep (20);
  IOWR_ALTERA_AVALON_PIO_DATA (PIO_BASE,LT24_ResetE);
  usleep (150000);
  // Initialize the controller's parameters
  SendCmd(Exit_Sleep);
  SendCmd(Display_ON);
  SendCmd(COLMODE);
  SendData(0x0055);
  SendCmd(MemAccessCtr);
  SendData(0x0008);
  SendCmd(Enable3G);
  SendData(0x0000);

  SendCmd(SetColAddr);
  SendData(0x0000);
  SendData((alt_u16)16);

  // Display a pattern of 32x32 colors (White Black Red Green Blue)
  for (int j=16; j<48; j++) {
    SendCmd(SetPageAddr); SendData(0x0000); SendData((alt_u16)j);
    PrintBar();
  }
  /* Event loop never exits. */
  while (1);
  return 0;
}
```

Color Pattern result



We see exactly what we expected : 32x32 color patterns.
Again this did not involve much code nor much complexity.

Conclusion

As you should now realize, the Terasic LT24 may not be the simplest choice in terms of LCD display : other displays driven in RGB mode (Hsync/Vsync) are typically more common, but the LT24 is in the class of *autonomous* displays that do not need to receive data continuously. This is a tremendous advantage for use with a low cost microcontroller, but much less with an FPGA.

However, this kind of display can be found at low prices (< $10 per unit) and sending a frame buffer from an FPGA to the LT24 over the 16-bits MCU interface is not very difficult (as demonstrated here) and can achieve high enough frame rates (thus allowing video). If you want to go in this direction, you can easily modify the FSM provided here to suit your application needs.

If you plan to display Texts and 2D graphics (GUI), ALSE has developed a Sprite-based Hardware Graphic Engine ("Gradien") that could also be used for this purpose.
But a processor-based system as demonstrated in the second part of this Application Note, associated with a graphics library is a straightforward and more usual solution.

I hope you enjoyed these Reference Designs and they will help you use cheap LCDs for your FPGA applications.

Bertrand Cuzeau – A.L.S.E
Tel +33 (0)1 84 16 32 32 – info at alse-fr dot com