CE2820-011
DE-10 LITE NIOS-II SoC

Rev. 1.3

System Architect: Evan Heinrich

Table of Contents

System Overview

This document targets application developers wishing to create applications on this specific NIOS-II System on a Chip. The system was designed using Quartus Prime and uses the Avalon memory mapped interface to incorporate all the custom and supplied IP blocks.

The system overall uses memory mapped control registers to interact with the peripherals. The overall memory map can be seen in the memory map section. DE-10 Lite development board also features an Arduino-based expansion header, which is used to connect the joystick and servos. This expansion board is optional, however the provided APIs assume that it is present.

Feature List

- 6x 7-segment displays with hardware ASCII decoders
- 10x Slider switches
- 10x SMD LEDs
- 2x Debounced, active-low pushbuttons
- VGA Subsystem
- 1x 40-Pin GPIO header, used to interface with LT24 2.4” resistive touchscreen LCD
- 1x Accelerometer, controlled via I2C
- 2x 32-bit Interval Timers

The following features are found on the expansion board which utilizes the Arduino header

- 2x PWM-controlled hobby servo motors with custom PWM controllers
- 1x Analog joystick with clicker, and matching dual-channel 12-bit ADC

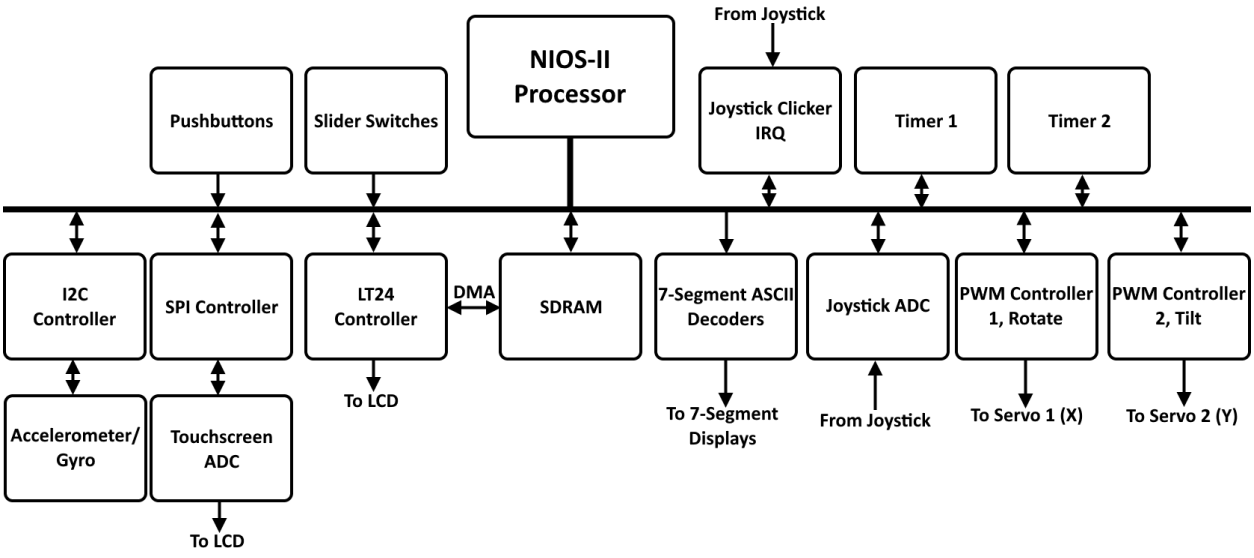
Additional Information

For more information on the Altera/Avalon IP blocks, see www.altera.com

Memory Map

Peripheral	Address Range
SDRAM	0x0000 0000 – 0x03FF FFFF
PWM-X	0xBEEF 0000 – 0xBEEF 0003
PWM-Y	0xBEEF 0004 – 0xBEEF 0007
ADC Sequencer	0xBEEF 1000 – 0xBEEF 1077
ADC Sample Storage	0xBEEF 1200 – 0xBEEF 13FF
LT24 Controller	0xBEEF 1400 – 0xBEEF 140F
Joystick GPIO	0xBEEF 1500 – 0xBEEF 150F
LT24 Touchscreen Interface	0xBEEF 1600 – 0xBEEF 161F
VGA Pixel Buffer	0xDEAD 1000 – 0xDEAD 100F
VGA Character Buffer	0xDEAD 2000 – 0xDEAD 3FFF
LED GPIO	0xFF20 0000 – 0xFF20 000F
HEX3 – HEX0	0xFF20 0020 – 0xFF20 002F
HEX5 – HEX4	0xFF20 0030 – 0xFF20 003F
Slider Switch GPIO	0xFF20 0040 – 0xFF20 004F
Pushbutton GPIO	0xFF20 0050 – 0xFF20 005F
Interval Timer 1	0xFF20 2000 – 0xFF20 201F
Interval Timer 2	0xFF20 2020 – 0xFF20 203F

System Diagram



Pushbutton GPIO

Overview

Provided on the DE-10 Lite development board are two SMD pushbuttons. These pushbuttons are automatically debounced and provide active-low signals. Interrupts can be generated by this component as IRQ1.

This component is comprised of IP provided by Intel/Avalon

Interfacing and Memory Map

Base Address: 0xFF20 0050

Offset	Register	R/W Mode
0x0	Data	R

Slider Switch GPIO

Overview

Provided on the DE-10 Lite development board are ten slide/toggle switches. These switches are not debounced and provide either a logic '1' when in the up position or a logic '0' when in the down position. Interrupts are disabled for these components.

This component is comprised of IP provided by Intel/Avalon

Interfacing and Memory Map

Base Address: 0xFF20 0040

Offset	Register	R/W Mode
0x0	Data	R

LED GPIO

Overview

Provided on the DE-10 Lite development board are ten SMD LEDs. They are controlled by active-high signals.

This component is comprised of IP provided by Intel/Avalon

Interfacing and Memory Map

Base Address: 0xFF20 0000

Offset	Register	R/W Mode
0x0	Data	W

`

7-Segment ASCII Decoders

Overview

Provided on the DE-10 Lite development board are a total of six seven-segment displays which also include decimal point lights. The segments operate on active low signals, however this was accounted for with the custom ASCII decoders.

With the seven-segment ASCII decoders, you can write ASCII characters to the corresponding memory location and have them appear on the development board's seven-segment displays encoded in the dSeg7 font.

Each individual seven-segment display has its own decoder attached to it.

Due to the system using a 32-bit address and data bus, the seven-segment displays had to be split into two batches. One of the batches is comprised of the lower 4 displays, as each display takes 1 byte/8 bits of data to display a full character. The remaining two upper displays are in a different memory location. This unconventional layout is accounted for with the provided API.

This component is comprised of IP provided by Intel/Avalon, as well as custom decoders

Interfacing and Memory Map

Base Address, HEX3..HEX0: 0xFF20 0020

Offset	Register	R/W Mode
0x0	Data	R/W

Base Address, HEX5..HEX4: 0xFF20 0030

Offset	Register	R/W Mode
0x0	Data	R/W

Data Register, HEX3..HEX0

[31..24]	[23..16]	[15..8]	[7..0]
HEX3 Data	HEX2 Data	HEX1 Data	HEX0 Data

Data Register, HEX5..HEX4

[31..16]	[15..8]	[7..0]
Unused	HEX5 Data	HEX4 Data

Interval Timers

Overview

This SoC includes two 32-bit interval timers, both of which function as down counters only. They both feature the ability to generate interrupts when their counts reach zero, which can be enabled or disabled in the respective control register. They also feature a continuous or single count mode.

This component is comprised of IP provided by Intel/Avalon

Interfacing and Memory Map

Base Address, Timer 1: 0xFF20 2000

Base Address, Timer 2: 0xFF20 2020

Offset	Register	R/W Mode
0x00	Status	R*
0x04	Control	R/W
0x08	Period_L	R/W
0x0C	Period_H	R/W
0x10	Snap_L	R**
0x14	Snap_H	R**

* Performing a write to the status register will clear the timeout flag. Write data ignored.

** Performing a write to either of the snapshot registers will cause a snapshot of the current count to be stored in the snapshot registers. Write data will be ignored.

Status Register

[31..2]	[1]	[0]
Unused	RUN	TIMEOUT

For the run field,

0) Timer not running
1) Timer is running

For the TIMEOUT field,

0) Timer count has not reached zero
1) Time count has reached zero

Note that for the TIMEOUT field, it is automatically set by hardware and must be deliberately cleared in software by writing to the status register. Write data is ignored.

Control Register

[31..4]	[3]	[2]	[1]	[0]
Unused	STOP	START	CONT	IRQEN

For the IRQEN field,

0) Timer will not generate interrupt requests
1) Timer will generate interrupt requests when TIMEOUT is set

For the CONT field,

0) Timer will stop when the count reaches 0
1) Timer will reload values from the period registers when the count reaches 0

For the START field, writing a '1' will start the internal timer. Input is ignored if the timer is already running. Writing a '0' regardless of status will be ignored.

For the STOP field, writing a '1' will stop the internal timer if already running. Input is ignored if the timer is already stopped. Writing a '0' regardless of status will be ignored.

Note: Simultaneously writing a '1' to both START and STOP will produce undefined behavior.

Period Registers

The period registers function as a form of auto reload register. Whenever the internal count reaches zero, the value stored within these registers will be loaded into the count. This value is split into a high and low register.

Period_H

[31..16]	[15..0]
Unused	Period[31..16]

Period_L

[31..16]	[15..0]
Unused	Period[15..0]

Snapshot Registers

Writing a ‘1’ to either of the snapshot registers will request the internal count to be copied into the snapshot registers. Write data ignored.

Snap_H

[31..16]	[15..0]
Unused	Snapshot[32..16]

Snap_L

[32..16]	[15..0]
Unused	Snapshot[15..0]

PWM Controllers

Overview

Implemented into this SoC is a custom designed PWM controller which targets the SG90 line of servos. These servo motors require a PWM signal with a variable pulse width to operate. The overall length of one PWM cycle for these servos is 20ms long, with the pulse ranging from 1ms to 2ms wide. The 20 millisecond period includes this pulse, so from one rising edge to the next is 20 milliseconds.

With the given pulse width, the hardware controllers inside of the servo will actuate the motor. With a pulse width of 1 millisecond, the servo will rotate clockwise as far as possible. With a pulse width of 2ms, the servo will rotate counterclockwise as far as possible. This means that with a pulse width of 1.5 milliseconds, the servo should ideally be in the center of its motion range.

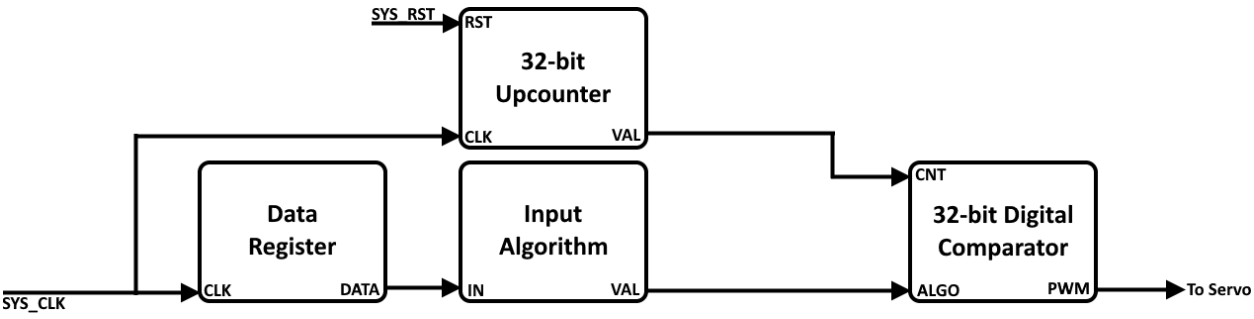
Note there is a noticeable amount of variation from servo to servo in regards to range of motion, so your results may vary.

With the provided API for these components, the user can specify either degrees or “counts.” One count is roughly equivalent to 0.9 degrees of rotation, but this varies from servo to servo. If a value of over 180 degrees (or 200 counts) is specified, the servos will disable their holding torque.

Two of these controllers exist in this system. One of which will control the rotation servo (also referred to as the “X” servo) while the other controls the pitch/tilt servo (also referred to as the “Y” servo).

This component is comprised of custom designed IP

System Diagram



Interfacing and Memory Map

Base Address, X: 0xBEEF 0000

Base Address, Y: 0xBEEF 0004

Offset	Register	R/W Mode
0x0	Data	R/W

Joystick and ADC

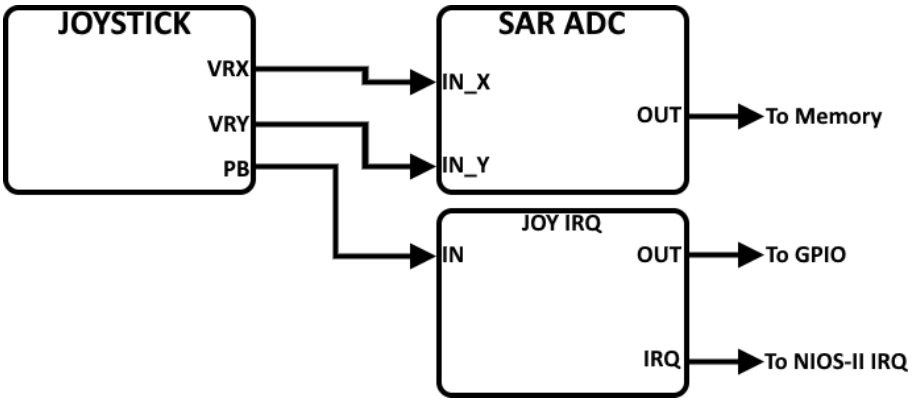
Overview

The ADC used to convert the analog readings from the joystick to digital readings used by the system is a 2-channel 12-bit successive approximation ADC. The ADC is configured with a sequencer to scan the variable X voltage first and store it in one memory location, then scan the variable Y voltage and store it in a different memory location. The results are viewable in the ADC sample/store registers.

There also is a GPIO connection to the joystick's clicker. This is to enable reading the status of said clicker, as well as there is an interrupt attached to it. This input is debounced.

This component is comprised of IP provided by Intel/Avalon

System Diagram



Interfacing and Memory Map

ADC Control Register Base Address: 0xBEEF 1000

Offset	Register	R/W Mode
0x0	Control	R/W

ADC Sample Storage Base Address: 0xBEEF 1200

Offset	Register	R/W Mode
0x0	Sample 0 (X)	R
0x4	Sample 1 (Y)	R

Joystick GPIO Base Address: 0xBEEF 1500

Offset	Register	R/W Mode
0x0	Data	R

Control Register

[31..4]	[3..1]	[0]
Unused	Mode	Run

For the Run field,

0) ADC is stopped
1) ADC is running

For the Mode field,

- Continuous conversion: ‘0’
- Single conversion: ‘1’
- Recalibrate: ‘7’
- Reserved: 6 – 2

Note that writes to the mode bits when the run bit is set will be ignored.

In order to retrieve data from the ADC, first the ADC must be enabled. After the ADC is enabled, the run bit must be set. The ADC will then either do a single conversion per channel or continuously sample depending on what is set in the Mode field of the control register. Data can then be read from the Sample Storage addresses, with offset 0 containing the X-value and offset 4 containing the Y-value.

When reading from the GPIO port connected to the joystick clicker, it will produce an active-low signal if the button is pressed.

LT24 LCD Controller

Overview

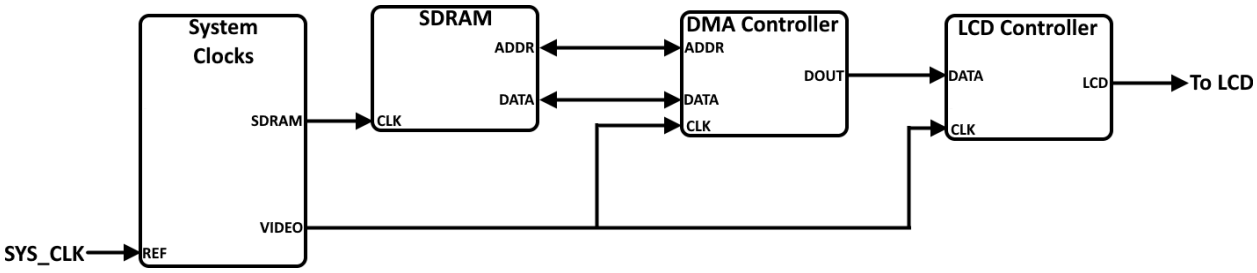
The target display for this controller is the LT24 LCD, which is a 2.4” 320x240 pixel display. The display itself interfaces using the DE-10 Lite’s 40-pin GPIO header. Due to the constraints of the GPIO header, only 16 pins could be used for data. As such, the display will operate using the RGB565 color space. If using the provided API, RGB888 or ARGB8888 values can be accepted, however they will be converted down to RGB565 with the alpha channel (if present) being ignored.

To improve performance, a DMA controller was also implemented. This allows the LCD to refresh itself automatically from a section of memory without requiring the CPU to interfere. This also allows for two pixel buffers to exist. The first one, which will be referred to as the “front buffer,” is the buffer being actively displayed on the LCD. There is also a “back buffer,” which can be modified without being displayed to the LCD. The frames can then be swapped when the user wishes.

The DMA controller is configured to read memory in a consecutive display pattern, meaning that the overall memory footprint for one pixel buffer is 320 pixels * 240 pixels * 2 bytes/pixel = approx. 154KB.

This component is comprised of IP provided by Intel/Avalon

System Diagram



Interfacing and Memory Map

Base Address: 0xBEEF 1400

Offset	Register	R/W Mode
0x0	Front Buffer [31..0]	R *
0x4	Back Buffer [31..0]	R/W
0x8	Resolution	R
0xC	Status	R

* By writing to the front buffer register, this sets the pending swap flag. This means that the back/front buffers will swap after the current frame is drawn.

Front/Back Buffer

[31..0]
Memory Address

Both the front and back buffer registers hold a memory address which points to the starting location of the respective pixel buffer.

Resolution Register

[31..16]	[15..0]
Vertical Resolution	Horizontal Resolution

Status Register

[31..24]	[23..16]	[7..4]	[1]	[0]
Unused*	Unused*	Color Mode	Consecutive/XY Mode	Swap Pending

* Unused in the current configuration. However, if the system were configured in XY mode, these two fields would represent the number of bits representing the X and Y coordinates where [31..24] = Y and [23..16] = X

For the “color mode” field,

1) 8-bit grayscale
2) 16-bit color, RGB565
3) 24-bit color, RGB888
4) 30/32 bit color, ARGB

For the “Consecutive/XY mode” field,

0) XY Mode
1) Consecutive Mode

For the “Swap Pending” field,

0) Not pending
1) Pending

LCD Touchscreen Interface

Overview

The touchscreen interface of the LT24 LCD display interfaces via SPI. The exact details of this interface can be found in the appendix.

It is strongly recommended to use the Intel/Avalon HAL for the SPI controller utilized in this implementation. As such, this section will focus on using the HAL to interface with the touchscreen component.

This component is comprised of IP provided by Intel/Avalon

Interfacing

Base Address: 0xBEEF 1600

As mentioned above, it is strongly recommended to use the Intel/Avalon HAL for this peripheral. To utilize this HAL, import the `altera_avalon_spi.h` header file.

The primary function used to communicate with the SPI controller is `alt_avalon_spi_command()`. This function takes in a total of seven arguments laid out in the following order:

1) Peripheral base address

2) Chip select value (0 for this peripheral as it is the only connected device)

3) Length of value being sent, in bytes (1 in this processor's case)

4) `uint8_t` pointer to the value being sent

5) Length of data being received, in bytes (2 in this processor's case)

6) `uint8_t` pointer to where the received data will be stored.

7) Flags (0 in this processor's case)

The two primary options for transmitted data should be either 0x92 (get X-value) or 0xD2 (get Y-value)

Note that the received data is 12-bits long, and as such, will require two bytes for storage as well as some value manipulation to get the data in the correct format. The most significant byte is stored in the first entry of the received data location with the least significant byte being stored in the second entry. The second entry also will have some zero-padding, so the overall method to ensure the data is combined into a correct value is with the following code:

```
(touch_rx[0]<<5) | (touch_rx[1]>>3)
```

Where `touch_rx` is where the received value will be stored.

Accelerometer/Gyroscope Interface

Overview

Included on the DE-10 Lite development board is an accelerometer. This accelerometer can be interfaced with via SPI or I2C, but in this system, it was implemented with an I2C interface.

As implementing a custom I2C driver can be difficult, it is recommended that either the included C API or the Intel/Avalon HAL be used to interface with the device.

This component is comprised of IP provided by Intel/Avalon

Interfacing via Included C API

If interfacing with the accelerometer using the included C API, you must include the header `accel.h` in whichever software is being created. The initialization method, **`Accel_Init()`** returns a "Boolean" (`uint8_t`) value to indicate whether the device was successfully initialized.

After initialization, the method **`Accel_getVals()`** can be called to retrieve the current values from the accelerometer. This function returns a structure (defined in `accel.h`) which is comprised of a Boolean "valid" flag and three `int16_t` values named "X," "Y," and "Z." Those last three fields contain the actual measurements. If the returned data has the "valid" flag marked false (0), the data will all contain values of -32768 as a sentinel value.

Interfacing via Intel/Altera HAL

If interfacing with the accelerometer using the Intel/Altera C API, you must include the header `altera_avalon_i2c.h` in the software being written. To initialize the device, you must first call the function `alt_avalon_i2c_open()` with the argument being the *device name* as opposed to the base address. On this system, the device name is `"/dev/gyro_i2c"`. This will return an `ALT_AVALON_I2C_DEV_t` pointer, which is used in the communication methods. If the device could not be initialized, the returned pointer will be null.

You must then call the `alt_avalon_i2c_master_target_set()` function. This function takes two arguments; the `ALT_AVALON_I2C_DEV_t` pointer obtained from initializing the device, as well as the I2C address of the device. On this system, the I2C address for the accelerometer is 0x53. This function will then return an `ALT_AVALON_I2C_STATUS_CODE`, which can then be compared against a few values defined in the header to determine if the target was set correctly.

You must then prepare a transmission buffer of two `uint8_t` values. The first of these values will be the target register in the peripheral, with the second value being the data to be stored in that register. See the appendix for more details.

Data is then transmitted with the `alt_avalon_i2c_master_tx()` function. The arguments for this function are as follows:

1) `ALT_AVALON_I2C_DEV_t` pointer obtained from initializing the device
2) `uint8_t*` transmission buffer
3) Size of the transmission buffer, in bytes
4) Additional flags

Note that the additional flags should be kept as `ALT_AVALON_I2C_NO_INTERRUPTS`, which is defined in the header file. This function will also return a status code.

Once the device is configured, data can then be received from the accelerometer. This is done in a similar way to transmission. The function to be used is `alt_avalon_i2c_master_tx_rx()`. This function will require a receive buffer in addition to the transmission buffer. The arguments for this function are as follows:

1) `ALT_AVALON_I2C_DEV_t` pointer obtained from initializing the device
2) `uint8_t*` transmission buffer
3) Size of the transmission buffer, in bytes
4) `uint8_t*` receive buffer
5) Size of the receive buffer, in bytes
6) Additional flags

Like the transmission function, the additional flags value should be kept as `ALT_AVALON_I2C_NO_INTERRUPTS`. This function will also return a status code. To receive the values from the accelerometer, the transmit buffer should hold the command 0x32.

One example way to set up the receive buffer is to use an `int16_t*` receive buffer. For use with the transmit and receive function, this must be cast to a `uint8_t*`, and the buffer size must also be set accordingly (two times the number of 16-bit values in the receive buffer).

The three values returned in that case will be the three values from the accelerometer.

VGA Subsystem

Overview

Included on the DE10-Lite development board is a VGA header. This allows for a connection to an external display, along with the potential for a standard output to the connected display.

Provided by Intel/Avalon's University Program was a VGA subsystem which can then drive the VGA port built onto the DE-10 Lite. It includes two independent buffers: a character buffer and a pixel buffer. The VGA subsystem can then mix these two buffers using an alpha blender such that the pixel buffer behaves as the background where the character buffer is the foreground.

The character buffer holds 60 rows with each row containing 80 characters. The character buffer uses 8-bit ASCII characters.

The pixel buffer operates at a resolution of 160 pixels by 120 pixels, which is then upscaled by hardware to the target resolution of 640 pixels by 480 pixels. The memory for the pixel buffer is configured to use an XY addressing system.

The pixel buffer component behaves extremely similar to the LT24 LCD Pixel buffer, with the only differences being the target resolution and the addressing modes. Otherwise, the behavior is identical.

This component is comprised of IP provided by Intel/Avalon

Interfacing and Memory Map

Base Address, Pixel Buffer: 0xDEAD 1000

Offset	Register	R/W Mode
0x0	Front Buffer [31..0]	R *
0x4	Back Buffer [31..0]	R/W
0x8	Resolution	R
0xC	Status	R

* By writing to the front buffer register, this sets the pending swap flag. This means that the back/front buffers will swap after the current frame is drawn.

Front/Back Buffer

[31..0]
Memory Address

Both the front and back buffer registers hold a memory address which points to the starting location of the respective pixel buffer.

Resolution Register

[31..16]	[15..0]
Vertical Resolution	Horizontal Resolution

Status Register

[31..24]	[23..16]	[7..4]	[1]	[0]
Y-bits	X-bits	Color Mode	Consecutive/XY Mode	Swap Pending

For the “color mode” field,

1) 8-bit grayscale
2) 16-bit color, RGB565
3) 24-bit color, RGB888
4) 30/32 bit color, ARGB

For the “Consecutive/XY mode” field,

1) XY Mode
2) Consecutive Mode

For the “Swap Pending” field,

1) Not pending
2) Pending

To interface with the character buffer, you must treat the base address, 0xDEAD 2000, as one `uint8_t` pointer. The overall dimensions of the character buffer are 80 characters per line with 60 lines total. You may then address this pointer as an array. The character buffer is also arranged in an XY format, with the number of X (horizontal width) bits being 7 and the number of Y (vertical height) bits being 6. For an example, see the following code:

```
charbuf[(y<<char_wBits)+x] = print;
```

Where `charbuf` is a `uint8_t` pointer referencing the base address of the character buffer, `y` is the Y-coordinate of the target character location, `char_wBits` is the number of horizontal bits, `x` is the X-coordinate of the target character location, and `print` is the character to be displayed.

Appendix

Other Resources

- Accelerometer datasheet
  - https://www.analog.com/media/en/technical-documentation/data-sheets/ADXL345.pdf
- Assorted DE-10 Lite documentation
  - https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=234&No=1021&PartNo=4
- LT24 LCD documentation
  - https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=65&No=892&PartNo=4#contents

Revision History

Revision 1.0: Initial Release

Revision 1.1: Touchscreen interface implemented and documented

Revision 1.2: Accelerometer interface implemented and documented

Revision 1.3: VGA Subsystem implemented and documented

Included APIs

- `accel.h`
  - Provides a basic structure for accelerometer data, as well as methods for initializing the accelerometer and its I2C interface and retrieving the data encapsulated in the provided structure.
- `cstick.h`
  - Provides methods for interfacing with the optional control stick. This includes analog readings and an interrupt controller for the clicker.
- `display.h`
  - Provides methods for printing to the 7-segment displays and individual LEDs.
- `keys.h`
  - Provides methods for reading the pushbuttons as well as the slider switches.
- `LT24.h`
  - Provides methods for controlling the optional LT24 LCD expansion card, as well as reading information about the touchscreen input.
- `servo.h`
  - Provides methods for controlling the servos which connect via the optional Arduino header port.
- `timer.h`
  - Provides a basic structure for controlling the interval timers.
- `vga.h`
  - Provides methods for controlling the VGA pixel buffer as well as the VGA character buffer.

Accelerometer API

Accelerometer data structure

```
typedef struct{
      uint8_t valid;
      int16_t X;
      int16_t Y;
      int16_t Z;
} accel_reading;
```

`uint8_t` **`Accel_Init`**`()`

Initializes the accelerometer peripheral, as well as the I2C controller used to interface with the peripheral. Returns 1 if successful, otherwise 0.

`accel_reading` **`Accel_getVals`**`()`

Attempts to read data from the accelerometer. Encapsulated said data using the accelerometer data structure which is specified above. The valid flag of said data structure will be 1 if the encapsulated data is valid. The valid flag will be set to 0 if any of the read data reports as -32768, which is a sentinel value indicating a false reading.

Control Stick API

`void cstick_Init()`

Initializes the ADC which samples the X and Y values of the control stick. Does not automatically start said ADC sampling.

`void cstick_Start()`

Starts the ADC, which will then alternate between sampling the X and Y values of the control stick.

`void cstick_Stop()`

Stops the ADC after the current sample finishes.

`uint32_t` **`cstick_getx()`**

Reads the most recent X value from the ADC's sample storage registers.

`uint32_t` **`cstick_gety()`**

Reads the most recent Y value from the ADC's sample storage registers.

`void click_init(void* ISR)`

Initializes the interrupt associated with the clicker of the control stick.

Parameters:

- **`void*`** `ISR:` Function pointer to the interrupt service routine that will be executed when the interrupt triggers

LED Display API

```
void Hex_printString(const char* string)
```

Prints the specified null-terminated string onto the seven segment displays of the DE10-Lite. Note there are only six total seven-segment displays, so if the input string is seven characters long, only the first six will print.

```
void Hex_scrollPrint(const char* scrollMe)
```

Prints the specified null-terminated string onto the seven segment displays of the DE10-Lite. If the length of the string is seven characters or longer, the message will then scroll from left to right with an adjustable delay between shifts. Will not reset to the first six characters after the whole message is scrolled through.

```
void Hex_Clear()
```

Clears the seven segment displays on the DE10-Lite.

```
void LED_Print(uint16_t value)
```

Prints a specified unsigned 16-bit value onto the individual LED's on the DE10-Lite. Note that there are only ten total individual LEDs, so the specified value is truncated to the lowest 10 bits.

```
void LED_Clear()
```

Clears the individual LEDs.

```
void delay_Set(useconds_t);
```

Sets the delay between shifts for the **`Hex_scrollPrint`** method. Note that `useconds_t` is defined in the `unistd.h` header as an unsigned long, also known as `uint32_t`,which represents microseconds in this case.

```
useconds_t delay_Get()
```

Returns the currently specified delay between shifts for the **`Hex_scrollPrint`** method.

Pushbutton and Slide Switch API

```
uint8_t button_Read()
```

Returns an 8-bit value representing the current status of the pushbuttons. Bit 0 represents pushbutton 0, and bit 1 represents pushbutton 1 on the DE10-Lite board.

```
uint16_t switch_Read()
```

Returns a 16-bit value representing the current status of the slider switches. Note that there are only ten slider switches on the DE10-Lite board, so only the lower ten bits of the returned value will be used.

LT24 LCD API

```
void LCD_Init()
```

Initializes the pixel buffer for the LT24 LCD.

```
void LCD_swapFrames()
```

Blocking method which will initiate a pixel buffer swap. Unblocks when the swap is complete.

```
void LCD_Fill(uint32_t color)
```

Fills the rear pixel buffer with the specified color, then automatically switches said buffer to the front. Note the input color is expected to be in either ARGB8888 or RGB888 format. This input color will be converted to RGB565 to match the display interface, with the alpha channel being ignored if present.

```
void LCD_drawPixel(uint32_t x, uint32_t y, uint32_t color)
```

Sets an individual pixel at the specified coordinates to the input color. See the description of **`LCD_Fill`** for the required color input format. Draws to the back buffer and does not automatically swap to the front.

```
void LCD_drawRect(uint32_t anchorX, uint32_t anchorY, uint32_t xLen,
                  uint32_t yLen, uint32_t Color)
```

Draws a rectangle with the specified (top-left) anchor position and dimensions. This only draws a solid filled rectangle. See the description of **`LCD_Fill`** for the required color input format. Draws to the back buffer and does not automatically swap to the front.

```
uint16_t getTouchX()
```

Reads the currently reported X-coordinate of the touchscreen. Some experimentation might be required to determine what the acceptable values for your individual LCD may be, but on the test system(s), this range was approximately 170 to 3000 to fall within the portion of the LCD with the actual pixels.

`uint16_t` **`getTouchY`**`()`

Reads the currently reported Y-coordinate of the touchscreen. As previously mentioned, some experimentation might be required to determine the acceptable values for your individual LCD. On the test system(s), this value was approximately 210 to 3170 to fall within the portion of the LCD with the actual pixels.

`uint32_t` **`getPixelX`**`()`

Determines the approximate pixel using the currently reported X-coordinate of the touchscreen. Assumes the same range as specified in the **`getTouchX`** description.

`uint32_t` **`getPixelY`**`()`

Determines the approximate pixel using the currently reported Y-coordinate of the touchscreen. Assumes the same range as specified in the **`getTouchY`** description.

Servo Controller API

```
void servo_move(uint8_t newpos, uint8_t sel)
```

Moves a specified servo motor to the specified absolute position. The absolute position may be any value from 0 to 255, however if the value is not in the range of 0 to 200 inclusive, the holding torque of the servos will be disabled. If the `sel` value is 1, the target servo will be the Y/tilt servo. If the `sel` value is 0, the target servo will be the X/rotate servo. A value of 0 represents fully counterclockwise and a value of 200 represents fully clockwise.

```
void servo_deg(uint8_t degrees, uint8_t sel)
```

Moves the specified servo to the specified angle in degrees, with zero being fully counterclockwise and 180 being fully clockwise. See the **`servo_move`** description for the behavior of `sel`.

```
uint8_t servo_read(uint8_t sel)
```

Reads the current value of the position for the specified servo. This does not account for possible slipping of the servos. See the **`servo_move`** description for the behavior of `sel`.

```
void servo_disable(uint8_t sel)
```

Disables the holding torque for the specified servo. See the **`servo_move`** description for the behavior of `sel`.

VGA Subsystem API

```
void VGA_Pixelbuf_Swapframes()
```

Swaps the rear and front pixel buffers after the current frame is drawn. Blocks until the swap is performed.

```
void VGA_Charbuf_Init()
```

Initializes the character buffer and “clears” by setting all characters to the ASCII code for space. The character buffer is comprised of 60 rows each containing 80 characters.

```
void VGA_Charbuf_Clear()
```

Clears the character buffer by setting all characters to the ASCII code for space.

```
void VGA_Charbuf_Char(const char print, uint32_t x, uint32_t y)
```

Prints the specified character to the specified character location. Enforces dimension checking. If the character does not fall within the 80 by 60 resolution, nothing is changed.

```
void VGA_Charbuf_String(const char* str, uint32_t x, uint32_t y)
```

Prints the specified null-terminated string to the specified character location. Enforces dimension checking. If the entire string does not fall within one 80-character row, nothing is printed. This method does not handle vertical scrolling, newlines, tabs, etc.

```
void VGA_Pixelbuf_Init()
```

Initializes the pixel buffer for the VGA connection. The display resolution is 160 pixels horizontally by 120 pixels vertically. This is then upscaled in hardware to 640 pixels horizontally by 480 pixels vertically.

```
void VGA_Pixelbuf_Fill(uint32_t color)
```

Fills the rear buffer with the specified color, then automatically swaps the filled rear buffer to the front. The input color is expected to be ARGB8888 or RGB888 encoded and will be converted to RGB565 for display with the alpha channel being ignored.

```
void VGA_Pixelbuf_Pixel(uint32_t x, uint32_t y, uint32_t color)
```

Sets the pixel at the specified coordinates to the specified color. This changes the pixel in the rear buffer, but does not automatically swap it to the front. Enforces dimension checking. See the description of **`VGA_Pixelbuf_Fill`** for expected color formats.

```
void VGA_Pixelbuf_Rect(uint32_t anchorX, uint32_t anchorY, uint32_t xLen,
                       uint32_t yLen, uint32_t Color)
```

Draws a solid filled rectangle with the specified (top-left) anchor, width, and length on the rear buffer. Does not automatically switch buffers after drawing. Enforces dimension checking. See the description of **`VGA_Pixelbuf_Fill`** for expected color formats.

This processor and all revisions:

https://msoe.box.com/s/h09bbek6li369k3fnhayun76gz5w0k2k

Thank you for reading this far! Please note that this processor was designed for academic purposes. It is not designed to achieve maximum efficiency or performance. The target platform was a DE10-Lite with the MAX10 10M50DAF484C7G FPGA.