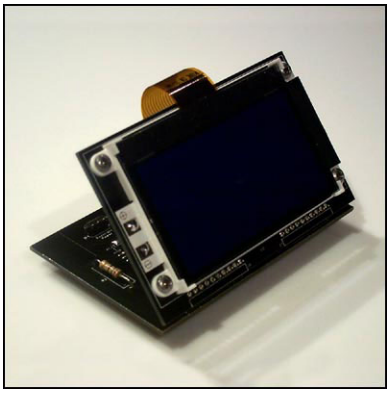
Display Layer



Description

The Display layer uses a 128x64 pixel white-on-blue graphical LCD display module to provide powerful visual output capabilities for the Tower. Capable of text or graphics modes, users have the ability to send text strings, draw graphical objects, and even interact with individual pixels.

Hardware Detail

The primary display module is an Optrex 51320 LCD screen. The PIC processor interfaces directly with the EPSON SED-1565 LCD controller located directly on-glass via a 30-pin ribbon cable. The LCD backlight is powered through the metal connectors used to hold the two circuit boards together, and can be powered off of either the primary or secondary bus, depending on power constraints for a given application.

Layer Code

The include file for the Display layer contains ten functions, one for blanking the display, two each for reading and writing data directly to the display RAM, three for interfacing with pixels, one for printing text strings, and one for drawing a graphical line on the screen. *(All of these functions are written for the PIC Foundation. The include files used with other foundations differ slightly, due to the presence of local variables.)*

The **display-clear** function takes no arguments, and is used to completely clear every pixel on the display simultaneously. The function simply sends a “0” to the layer, requesting that the operation be performed. The function then waits for a “1” to be sent back form the layer, indicating that the operation has been completed. Even though this is not necessary, it is there to ensure that the display is in a good state if we try to write to it immediately following the clear command.

```
to display-clear
    i2c-start
    i2c-write-byte $16
    i2c-write-byte 1
    i2c-write-byte 0
    i2c-stop
    i2c-start
    i2c-write-byte $17
    waituntil [(i2c-read-byte 1) = 1]
    seti2c-byte i2c-read-byte 0
    i2c-stop
end
```

The **display-write-command-byte** function is used to send a command to the display. Commands are used to set the contrast, toggle display power on and off, enter and exit write-mode, and adjust other low-level system parameters. The function takes a single argument, the value to be sent, and sends the layer a “1” indicating that a command-byte write is being performed, followed by the value to be written.

```
to display-write-command-byte :data
    i2c-start
    i2c-write-byte $16
    i2c-write-byte 2
    i2c-write-byte 1
    i2c-write-byte :data
    i2c-stop
end
```

The **display-write-data-byte** function is used to directly write data into the display RAM. This function is usually used following a **display-write-command-byte**, to send data after a write command has been initiated. The function takes a single argument, the value to be sent, and sends the layer a “2” indicating that a data-byte write is being performed, followed by the value to be written.

```
to display-write-data-byte :data
    i2c-start
    i2c-write-byte $16
    i2c-write-byte 2
    i2c-write-byte 2
    i2c-write-byte :data
    i2c-stop
end
```

The **display-read-command-byte** function is used to read the status of the display. Information can easily be obtained regarding current power and contrast settings at any time during operation. The first argument in this case however, is a “3”, signifying a status read. After the request is sent, the result is read out of the layer’s transmit buffer as a single byte. Actually two bytes are read out, but the first is ignored, as it is known to be a “1”, representing the number of arguments to follow.

```
to display-read-command-byte
    i2c-start
    i2c-write-byte $16
    i2c-write-byte 1
    i2c-write-byte 3
    i2c-stop
    i2c-start
    i2c-write-byte $17
    ignore i2c-read-byte 1
    seti2c-byte i2c-read-byte 0
    i2c-stop
    output i2c-byte
end
```

The **display-read-data-byte** function is used to directly read data from the display RAM. This function is usually used following a **display-write-command-byte**, to determine what pixels are currently turned on in a specific region of the screen. The function takes no arguments, and sends the layer a “2” indicating that a data-byte read is being performed. This function is normally performed in conjunction with a **display-write-data-byte** call, to read out the contents of a RAM location, modify it, and write it back to its original place.

```
to display-read-data-byte
    i2c-start
    i2c-write-byte $16
    i2c-write-byte 1
    i2c-write-byte 4
    i2c-stop
    i2c-start
    i2c-write-byte $17
    ignore i2c-read-byte 1
    seti2c-byte i2c-read-byte 0
    i2c-stop
    output i2c-byte
end
```

The **display-clear-line** function is used to erase one of the eight possible lines of text on the screen. The function takes a single argument, the line number (a value from 0 to 7). When the line is erased, its counter will also be reset, so that the next characters printed to it will begin at the left side of the screen. After the line is erased, A value of “5” is sent to the layer, followed by the line number to be cleared. Since the Display layer itself takes a bit of time to complete this operation, the function waits for the layer to return a value of “1” indicating its successful completion.

```
to display-clear-line :line
    i2c-start
    i2c-write-byte $16
    i2c-write-byte 2
    i2c-write-byte 5
    i2c-write-byte :line
    i2c-stop
    i2c-start
    i2c-write-byte $17
    waituntil [(i2c-read-byte 1) = 1]
    seti2c-byte i2c-read-byte 0
    i2c-stop
end
```

The **display-print-string** function is used to print a string of text on one of the eight available character lines on the display. The function takes two arguments, the line number (a value from 0 to 7), and the string itself, up to 21 characters in length. The string will begin printing at the location of the current line counter, so that the text will immediately follow anything that has already been printed on the line, and then increment the counter accordingly. The function first sends a “6” to the layer, indicating that a string-write is about to take place. The line number is then sent, followed by the string itself. The **display-get-string-length** function is used to determine the length of the string before it is sent. Since the Display layer itself takes a bit of time to complete this operation, the function waits for the layer to return a value of “1” indicating its successful completion.

```
to display-print-string :line :string
    setdisp-nn display-get-string-length :string
    i2c-start
    i2c-write-byte $16
    i2c-write-byte disp-nn + 2
    i2c-write-byte 6
    i2c-write-byte :line
    setdisp-n :string
    repeat disp-nn
    [
        i2c-write-byte read-prog-mem disp-n setdisp-n disp-n + 1
    ]
    i2c-stop
    i2c-start
    i2c-write-byte $17
    waituntil [(i2c-read-byte 1) = 1]
    seti2c-byte i2c-read-byte 0
    i2c-stop
end

to display-get-string-length :string
    setdisp-n :string
    loop
    [
        ifelse ((read-prog-mem disp-n) = 0)
            [output disp-n - :string]
            [setdisp-n disp-n + 1]
    ]
end
```

The **display-print-char** function is used to print a single character to the screen. The function takes two arguments, the line number (a value from 0 to 7), and the ASCII code of the character to print. The function sends three values to the layer itself, a “7” indicating that a character is to be printed, followed by the line and character arguments. The character will print at the current location of the line counter, directly after the last thing that was printed, and increment the counter accordingly.

```
to display-print-char :line :char
    i2c-start
    i2c-write-byte $16
    i2c-write-byte 3
    i2c-write-byte 7
    i2c-write-byte :line
    i2c-write-byte :char
    i2c-stop
end
```

The **display-print-num** function is used to print a number to the screen. The function takes two arguments, the line number (a value from 0 to 7), and the number itself. The function divides the number into its individual digits, and calls **display-print-digit** as many times as needed to print the full number. The **display-print-digit** function actually chops out the desired digit, adds the number 48 to convert it to an ASCII code, and then calls **display-print-char** to actually print the digit.

```
to display-print-num :line :n
    if :n > 9999 [display-print-digit :line :n 10000]
    if :n > 999 [display-print-digit :line :n 1000]
    if :n > 99 [display-print-digit :line :n 100]
    if :n > 9 [display-print-digit :line :n 10]
    display-print-digit :line :n 1
end

to display-print-digit :line :n :d
    display-print-char :line ((:n / :d) % 10) + 48
end
```

The **display-set-line-index** function is used to change the location of the line counter for a given line. This makes it possible to overwrite portions of a line without erasing the entire line itself. The function takes two arguments, the line number (a value from 0 to 7), and the desired character index (a value from 0 to 20). A total of three values are sent to the layer itself, beginning with an “8” indicating that an index set is being performed, and followed by the line and index arguments themselves.

```
to display-set-line-index :line :index
    i2c-start
    i2c-write-byte $16
    i2c-write-byte 3
    i2c-write-byte 8
    i2c-write-byte :line
    i2c-write-byte :index
    i2c-stop
end
```

The **display-set-pixel** function is used to turn on a specific pixel on the screen. The function takes two arguments, the x (from 0 to 127) and y (from 0 to 63) coordinates of the pixel to turn on. The 0,0 point on the screen is the upper-left corner, and both x and y increment as we move towards the lower right. The function sends a “9” to the layer, indicating that a set-pixel command is being issued, followed by the x and y values themselves.

```
to display-set-pixel :x :y
    i2c-start
    i2c-write-byte $16
    i2c-write-byte 3
    i2c-write-byte 9
    i2c-write-byte :x
    i2c-write-byte :y
    i2c-stop
end
```

The **display-clear-pixel** function is used to turn off a specific pixel on the screen. The function takes two arguments, the x (from 0 to 127) and y (from 0 to 63) coordinates of the pixel to turn on. The 0-0 point on the screen is the upper-left corner, and both x and y increment as we move towards the lower right. The function sends a “10” to the layer, indicating that a clear-pixel command is being issued, followed by the x and y values themselves.

```
to display-clear-pixel :x :y
    i2c-start
    i2c-write-byte $16
    i2c-write-byte 3
    i2c-write-byte 10
    i2c-write-byte :x
    i2c-write-byte :y
    i2c-stop
end
```

The **display-test-pixel** function is used to find out if a specific pixel on the screen is currently turned on. The function takes two arguments, the x (from 0 to 127) and y (from 0 to 63) coordinates of the pixel to turn on. The 0,0 point on the screen is the upper-left corner, and both x and y increment as we move towards the lower right. The function sends an “11” to the layer, indicating that a set-pixel command is being issued, followed by the x and y values themselves. It then reads back the result, ignoring the first byte back, which is known to be a “1”, indicating that one data value is being sent. The function returns a “0” if the pixel is currently turned off, and a “1” if it is on.

```
to display-test-pixel :x :y
    i2c-start
    i2c-write-byte $16
    i2c-write-byte 3
    i2c-write-byte 11
    i2c-write-byte :x
    i2c-write-byte :y
    i2c-stop
    i2c-start
    i2c-write-byte $17
    ignore i2c-read-byte 1
    seti2c-byte i2c-read-byte 0
    i2c-stop
    output i2c-byte
end
```

The **display-draw-line** function is used to draw a line on the screen from any x-y coordinate, to any other x-y coordinate. The function takes four arguments, the starting x and y coordinates, and the ending x and y coordinates. The x-values can range from 0 to 127, and the y values from 0 to 63. The function uses a complex stack of if-statements to determine exactly how to draw the line based on its slope and direction. The details of the function are too complex to explain here, but you can try to follow it yourself and figure it out.

```
to display-draw-line :x1 :y1 :x2 :y2
    if (:x1 = :x2)
        [
            ifelse (:y2 > :y1)
                [
                    setdisp-n 0
                    repeat (:y2 - :y1) + 1
                        [display-set-pixel :x1 :y1 + disp-n
                         setdisp-n disp-n + 1]
                    stop
                ]
```

```
            [
                setdisp-n 0
                repeat (:y1 - :y2) + 1
                    [display-set-pixel :x1 :y2 + disp-n
                     setdisp-n disp-n + 1]
                stop
            ]
    ]
if (:y1 = :y2)
    [
        ifelse (:x2 > :x1)
            [
                setdisp-n 0
                repeat (:x2 - :x1) + 1
                    [display-set-pixel :x1 + disp-n :y1
                     setdisp-n disp-n + 1]
                stop
            ]
            [
                setdisp-n 0
                repeat (:x1 - :x2) + 1
                    [display-set-pixel :x2 + disp-n :y1
                     setdisp-n disp-n + 1]
                stop
            ]
        ]
setxdiff (:x2 - :x1)
setydiff (:y2 - :y1)
if (xdiff < 0)
    [setxdiff xdiff - (2 * xdiff)]
if (ydiff < 0)
    [setydiff ydiff - (2 * ydiff)]
setdisp-n 0
ifelse (xdiff > ydiff)
    [
        ifelse ((:x2 - :x1) > 0)
            [
                ifelse (:y2 > :y1)
                    [
                        repeat xdiff + 1
                            [display-set-pixel
                             (:x1 + disp-n)
                             (:y1 + (disp-n * ydiff / xdiff))
                             setdisp-n disp-n + 1]
                    ]
                    [
                        repeat xdiff + 1
                            [display-set-pixel
                             (:x1 + disp-n)
                             (:y1 - (disp-n * ydiff / xdiff))
                             setdisp-n disp-n + 1]
                    ]
            ]
            [
                ifelse (:y2 > :y1)
```

```
                        [
                            repeat xdiff + 1
                                [display-set-pixel
                                 (:x1 - disp-n)
                                 (:y1 + (disp-n * ydiff / xdiff))
                                 setdisp-n disp-n + 1]
                        ]
                        [
                            repeat xdiff + 1
                                [display-set-pixel
                                 (:x1 - disp-n)
                                 (:y1 - (disp-n * ydiff / xdiff))
                                 setdisp-n disp-n + 1]
                        ]
                ]
        ]
        [
            ifelse ((:y2 - :y1) > 0)
                [
                    ifelse (:x2 > :x1)
                        [
                            repeat ydiff + 1
                                [display-set-pixel
                                 (:x1 + (disp-n * xdiff / ydiff))
                                 (:y1 + disp-n)
                                 setdisp-n disp-n + 1]
                        ]
                        [
                            repeat ydiff + 1
                                [display-set-pixel
                                 (:x1 - (disp-n * xdiff / ydiff))
                                 (:y1 + disp-n)
                                 setdisp-n disp-n + 1]
                        ]
                ]
                [
                    ifelse (:x2 > :x1)
                        [
                            repeat ydiff + 1
                                [display-set-pixel
                                 (:x1 + (disp-n * xdiff / ydiff))
                                 (:y1 - disp-n)
                                 setdisp-n disp-n + 1]
                        ]
                        [
                            repeat ydiff + 1
                                [display-set-pixel
                                 (:x1 - (disp-n * xdiff / ydiff))
                                 (:y1 - disp-n)
                                 setdisp-n disp-n + 1]
                        ]
                ]
        ]
end
```

Examples of Use

To use the Display layer, we probably want to start by clearing it like this:

```
display-clear
```

In most cases, users will not directly be using the write-byte and read-byte procedures, so let's skip ahead to the fun stuff and make things light up. We'll start by just turning on a pixel in about the middle of the screen:

```
display-set-pixel 63 31
```

How about we now turn on a random pixel on the screen, just for kicks.

```
display-set-pixel (random % 128) (random % 64)
```

To see something really cool, let's write a loop that just keeps setting and clearing random pixels:

```
loop [display-set-pixel (random % 128) (random % 64)
      display-clear-pixel (random % 128) (random % 64)]
```

Now, we need to stop this program from running before we can talk to the Foundation again. Just pushing the button won't do it though, because if that stops the program in the middle of an I^2C communication, the layer will be in a bad state. The best way to make sure everybody's happy is to turn the power off and then back on.

Now, why don't we go ahead and print some text in the middle of the screen:

```
display-print-string 3 "|LCD Displays are cool|
```

Let's say "Hello" on another line:

```
display-print-string 1 "|Hello|
```

Since the line pointer for line 1 now points to the next blank space after the word "Hello", the next thing that we print on that line will appear right after it. Let's print a number:

```
display-print-num 1 123
```

The line should now read "Hello123". Let's say that we want it to say "Hello456" instead. We could always clear the line and start over, but we can also move the line index back to the point we want the new number to be printed at. If you start counting at 0, the number "4" needs to be printed at position 5. We'll set the index like this:

```
display-set-line-index 1 5
```

Now, if we print the number 456 like this:

```
display-print-num 1 456
```

The line should read "Hello456". Let's now clear the line:

```
display-clear-line 1
```

Finally, let's draw two lines on the screen, forming a big "X" from corner to corner:

```
display-draw-line 0 0 127 63
display-draw-line 127 0 0 63
```

With these basic text and drawing primitives, it’s possible to write many complex graphics routines capable of displaying anything the user desires.