Digital Oscilloscope Technical Documentation

Updated July 2017

Contents

List of Figures

1 Oscilloscope Hardware

The following describes the hardware of the System on Programmable Chip (SOPC) digital oscilloscope. This section details the physical hardware as well as the hardware logic designed and uploaded to the FPGA.

1.1 Hardware Overview

The digital oscilloscope depends on various hardware components that interact with each other to make visualizing and analyzing an analog signal possible. The flow of data and other signals is shown in the below block diagram.

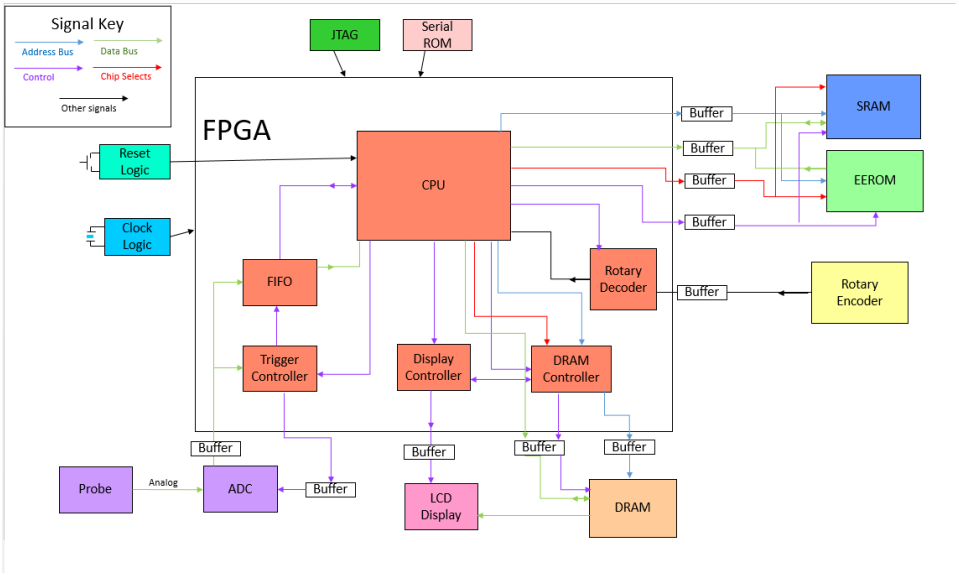


Figure 1: Block diagram of how all hardware components and signals interact for

The physical components in the block diagram were then arranged on a circuit board in the following layout.

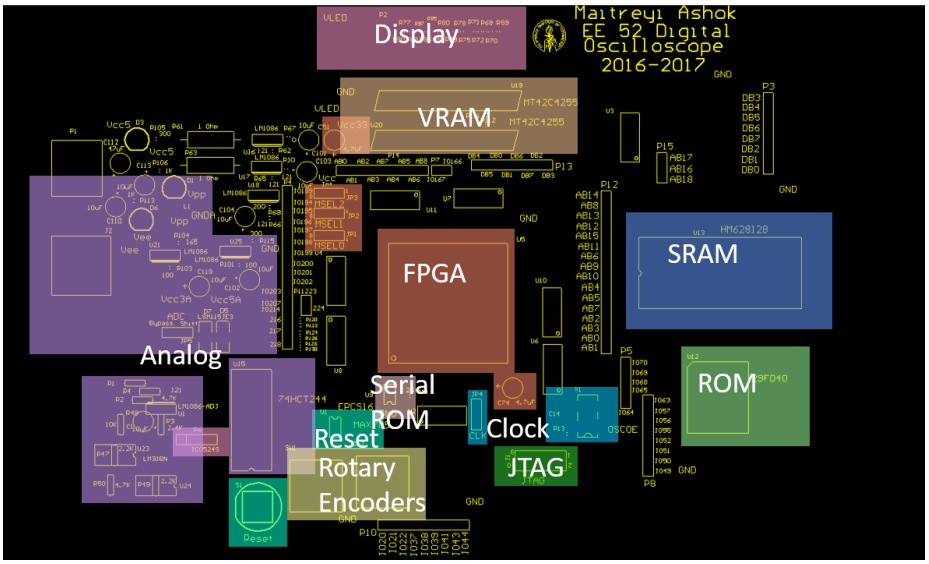


Figure 2: Layout of all components from block diagram on PCB (excluding external components)

Analog The first main component of the oscilloscope is the analog section (purple). Signals are first received by the oscilloscope from a signal source through a probe with a BNC connector. The oscilloscope only handles signals between -12 V and +12 V. These signals are then scaled down to an analog voltage between 0 and 3.3 V using a level shifter circuit with fractional gain. This reduced amplitude signal is then input to an analog to digital converter with 8 bit resolution and a maximum conversion rate of 20 megasamples per second. The sampling frequency is based on the sweep time for each sample and is set by the user using the rotary encoders and menu. The ADC is then clocked at this sampling frequency. The ADC was hooked up from a separate board to the necessary IO pins of the FPGA and power lines (which is not shown in the main PCB layout pictured above).

Analog Internals The 8 bit resolution sample measured of the signal then enters the FPGA into both a FIFO and trigger controller. The trigger controller takes the digital value of the signal and compares it to a trigger level with 7 bits of resolution. Depending on the comparison of these values as well as the trigger slope, a trigger event is generated when manual triggering. If auto trigger is selected, the trigger event is generated either by the above process or after a certain amount of time (auto trigger timeout), whichever occurs first. After the trigger event is generated, 480 samples taken at the sampling clock frequency set by the user are stored in a FIFO. When the FIFO is full, the samples stored will be read and saved externally.

FPGA The sampled data is stored in the Altera Cyclone III FPGA, which is the hub for all signals on the digital oscilloscope. Several of the hardware components are also implemented internally through VHDL and block diagram logic rather than through physical components. The FPGA also contains the design for a CPU, which uses a NIOS II processor. The CPU creates the signals to read and write data for the memory components, stores the user choices for analog settings, stores rotary encoder turns and pushes, and handles the reset and clock signals. The CPU uses hardware interrupts from rising edges in the Peripheral I/Os (PIO) that are handled in software.

Video RAM These 8 bit samples are then written to the two 256 K x 4 bit VRAMs (light orange) during write cycles for the VRAM. Each 18 bit address corresponds to the color for one pixel on the display. The VRAM is also refreshed regularly when no other operation is being performed. Rows are also regularly transferred from the DRAM portion to the Serial Access Memory port. This port is then clocked regularly to allow for a transfer of data 1 pixel at a time to the LCD display. The write, read, row transfer, and refresh cycles are all governed by a finite state machine, as well as a multiplexer to control the address lines.

Display The 3 red, 3 green, and 2 blue color bits specified in the VRAMs are received by the 480 x 272 pixel graphics LCD (pink) at a 10 MHz frequency. The data receiving by the display is enabled during the valid regions of the Horizontal and Vertical sync cycles of the LCD. The samples recorded by the ADC are displayed on the graphics display. The grid lines and axes can be shown or not depending on the option selected by the user. In addition, a menu is shown on the top right corner of the display for the users to select options for the oscilloscope. This menu can be turned on and off based on user input, and it allows for the user to select sweep rate, grid settings, trigger level, slope, delay, and mode. based on signals from the rotary encoders and internal decoder.

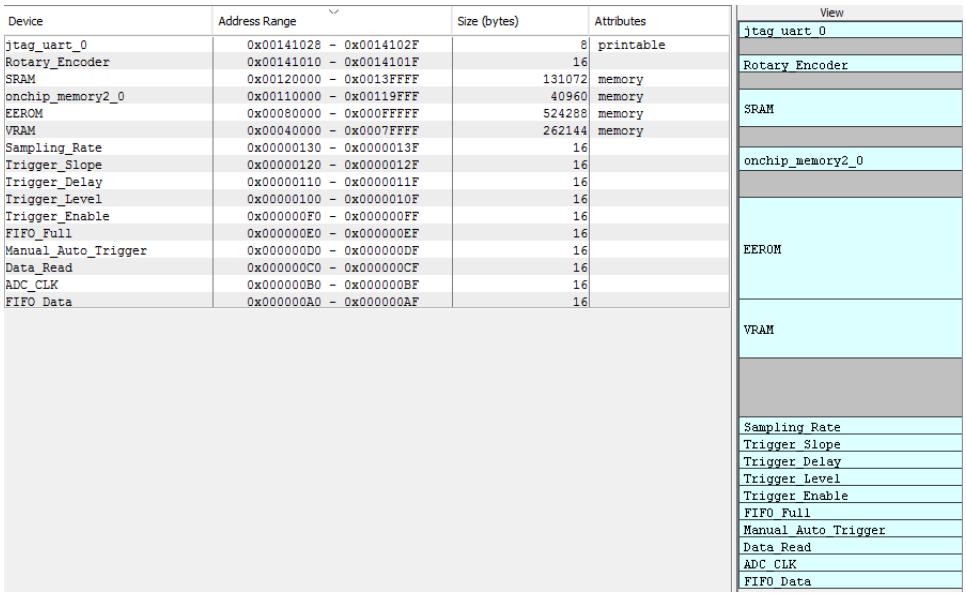
Rotary Encoders The parameters for the analog and display components of the scope are set through the rotary encoders (light yellow). The rotary encoders can be turned clockwise or counterclockwise and pressed. These turns are encoded in two pins for each rotary encoder. Depending on which of the pins changes values first, the direction of turning can be determined. The signals from pin A and B of each encoder is sent to the FPGA, where the decoding is done. The push button signal is debounced internally in the FPGA with a debounce time of 100 ms for the encoder to be considered pushed. For each of the clockwise/counterclockwise turning and pushing events, an interrupt through a PIO is generated, which is handled in software to set the parameters for the analog component and display.

Data The key buffer of the registered turns/pushes in the dials as well as the buffer of the analog samples made must be stored in random access memory to allow for reading and writing of the data. This is done in the 128 KByte SRAM (light blue), with 17 address lines and 8 data lines.

Other Storage In addition, to allow for interrupts to be generated in response to user input, exception and interrupt stacks must be present. The exception stack, interrupt stack, read only data, and code are all stored in the 512 KByte EEROM (light green), which has 19 address lines and 8 data lines, both of which are shared with the SRAM.

Other Components Some other components are necessary for the FPGA to run the digital oscilloscope successfully. The reset logic allows for a manual pushbutton reset of the system, which resets all hardware logic in the FPGA. In addition, the system is run at a 30 MHz clock. The hardware logic used in the FPGA to interface between the software and physical hardware components is stored in the Serial ROM to allow for a standalone system and is uploaded into the FPGA upon power up through the JTAG.

All signals entering and leaving the FPGA are buffered to account for varying voltage requirements of the chips used, and to properly drive the signals high or low as appropriate. In addition, to keep all the power lines clean, the power pins are all connected to ground through a 0.1 uF bypass capacitor on all chips. In addition, each of the power planes of the FPGA has bypass capacitors to further filter out high frequency noise. All components are also mapped into memory according to the following memory map.



Device	Address Range	Size (bytes)	Attributes
jtag_uart_0	0x00141028 - 0x0014102F	8	printable
Rotary_Encoder	0x00141010 - 0x0014101F	16	
SRAM	0x00120000 - 0x0013FFFF	131072	memory
onchip_memory2_0	0x00110000 - 0x00119FFF	40960	memory
EEROM	0x00080000 - 0x000FFFFF	524288	memory
VRAM	0x00040000 - 0x0007FFFF	262144	memory
Sampling_Rate	0x00000130 - 0x0000013F	16	
Trigger_Slope	0x00000120 - 0x0000012F	16	
Trigger_Delay	0x00000110 - 0x0000011F	16	
Trigger_Level	0x00000100 - 0x0000010F	16	
Trigger_Enable	0x000000F0 - 0x000000FF	16	
FIFO_Full	0x000000E0 - 0x000000EF	16	
Manual_Auto_Trigger	0x000000D0 - 0x000000DF	16	
Data_Read	0x000000C0 - 0x000000CF	16	
ADC_CLK	0x000000B0 - 0x000000BF	16	
FIFO_Data	0x000000A0 - 0x000000AF	16	

Figure 3: Mapping of all peripheral I/Os and memory devices in the address map

In Progress All components of the oscilloscope have been designed and bult. However, the analog components have not been tested yet, though the hardware design has been simulated.

1.2 Power

Power is provided from a wall wart with 4 pins through a MiniDin4 connector (P1) for 5 V, +12 V, -12 V, and ground. The +/- 12 V are used for analog power, and will be discussed further in the Analog section. To initially filter out most of the AC signals before the voltages are regulated, a large capacitor is placed at each major power line. A 10 uF capacitor (C110, C113) bypasses the +/- 12 V lines, and a 47 uF (C112) capacitor bypasses the +5 V line. All power pins on all ICs are bypassed by 0.1 uF capacitors, to filter out AC noise from the DC power.

There are 3 power LEDs for each of the 5 V (light green - D3), +12 V (dark green - D1), and -12 V (light green - D6) lines to indicate to the user that the oscilloscope is turned on. These diodes are in series with a resistor (R105, R106, R113) to limit the amount of current flowing through them to about 10 mA. These components are all located on the circuit board in the following configuration.

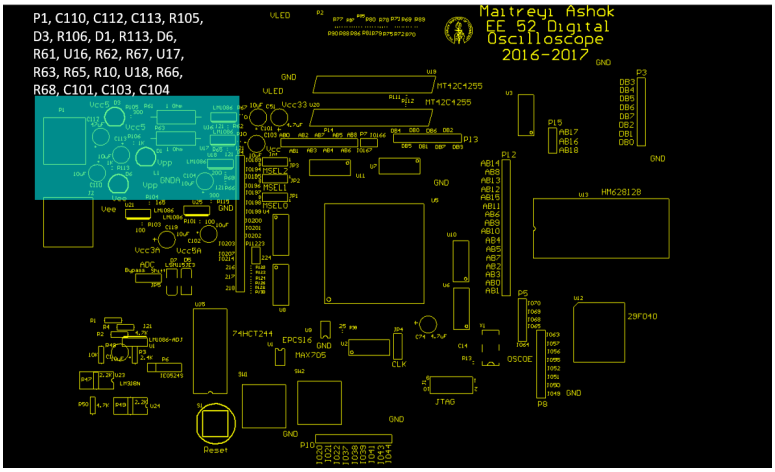


Figure 4: Layout of power component on PCB, as well as the designators for components that are part of power

The 5 V line is regulated down to the voltages necessary for the FPGA and the other ICs. This is done using LM1086-ADJ regulators, which were used for their general compatibility with the voltages necessary for the oscilloscope, as well as the ease of adjusting the output voltage with the same component based on external resistors. The LM1086-ADJ regulators have an internal reference voltage of 1.25 V, so are used in the following configuration. For all 3 regulators, a 10 uF high voltage rated capacitor (C101, C103, C104) is included at the output of the regulator, for stability of the power lines.

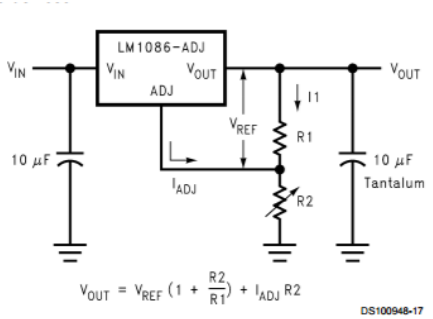


Figure 5: From LM1086 datasheet, how to connect regulator to get specific output voltage

Since small fluctuations in the power lines from the DC value are allowed, the small amount of current out of the reference pin is ignored. Thus for a 1.25 V regulator (U16), a 0 Ohm resistor is used for R_2 (R67) and a 121 Ohm resistor is used for R_1 (R62). Thus,

$$V_{out} = V_{ref} = 1.25V$$

A high power rated 1 Ohm resistor (R61) is also included before the regulator to limit the amount of current entering the regulator.

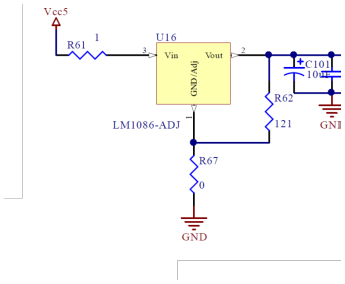


Figure 6: Connection in schematic for 5 V $\rightarrow$ 1.25 V regulation

For a 2.5 V regulator (U17), R_1 (R65) and R_2 (R10) are 121 Ohm resistors to allow for

$$V_{out} = V_{ref}(1 + 1) = 2.5V$$

Again a 1 Ohm resistor is included in series with the regulator (R63).

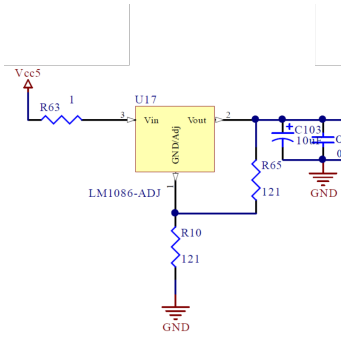


Figure 7: Connection in schematic for 5 V $\rightarrow$ 2.5 V regulation

For a 3.3 V regulator (U18), R_2 = 200 Ohm (R68) and R_1 = 121 Ohm (R66). This allows for

$$V_{out} = V_{ref}(1 + 1.65) = 3.3V$$

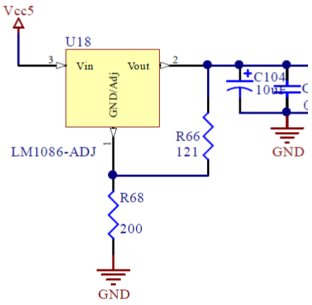


Figure 8: Connection in schematic for 5 V $\rightarrow$ 3.3 V regulation

1.3 FPGA Hardware

The layout on the PCB for the basic FPGA hardware as well as the hardware components used to build this portion of the oscilloscope is shown.

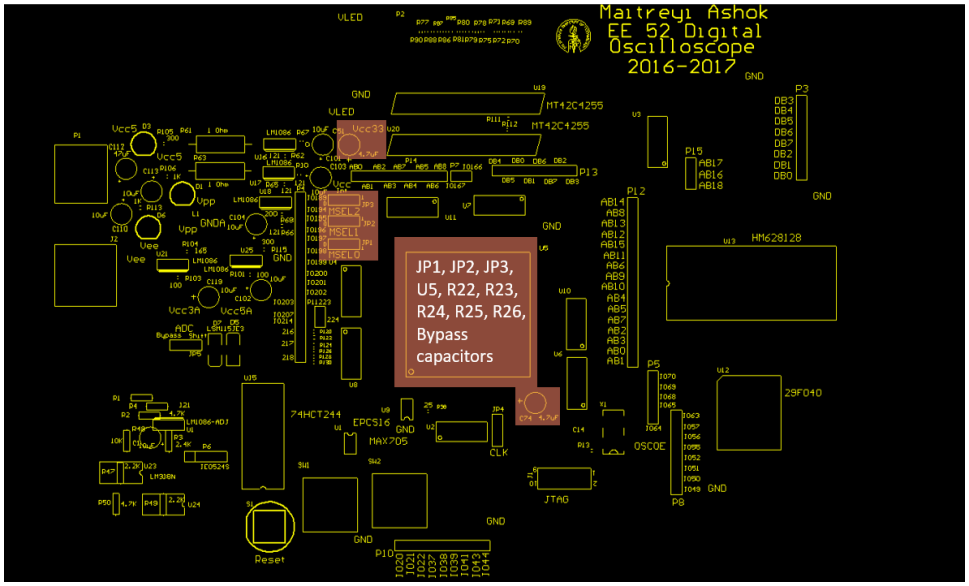


Figure 9: Layout of FPGA component (and MSEL) in PCB, including designators for parts involved in the component

The FPGA is the hub of all signals as seen in the block diagram, and contains the following internal blocks. Each block will be discussed in further detail in the corresponding section of the oscilloscope it works with.

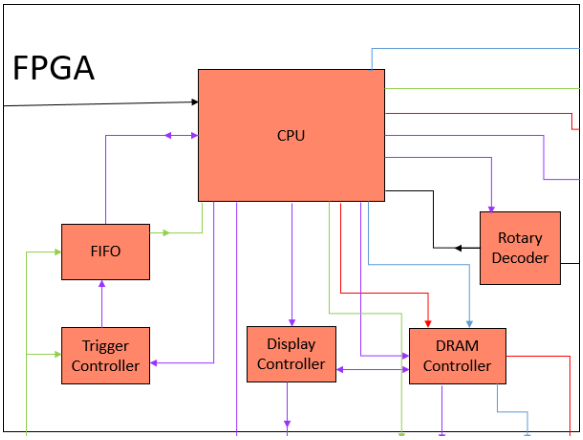


Figure 10: Components within FPGA in block diagram, and how these interact

The FPGA used is Altera Cyclone III EP3C25Q240 (U5), with 148 I/O pins, as pictured in the below schematic.

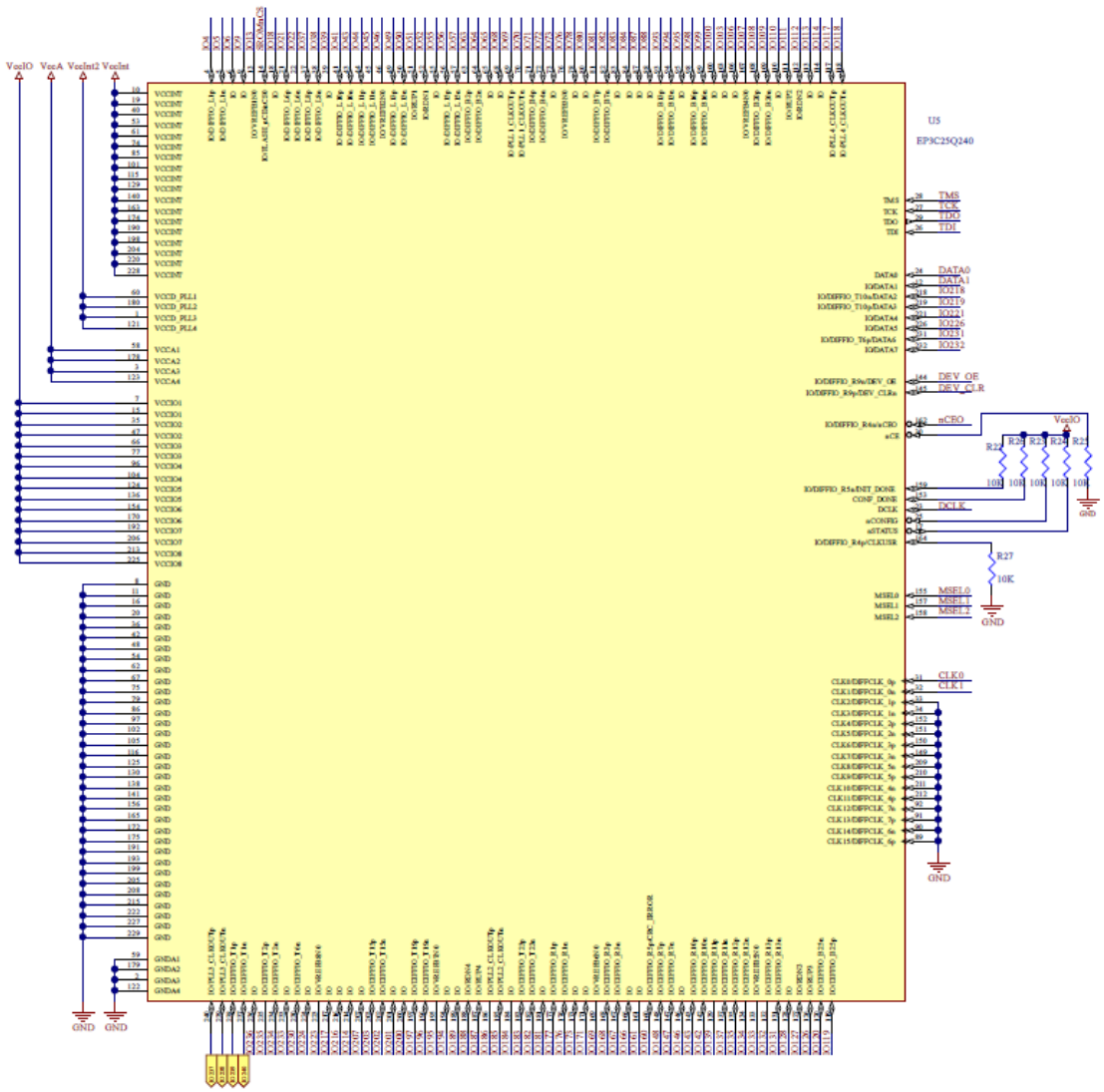


Figure 11: FPGA and its connections in schematic

Every power pin of the FPGA is bypassed by a 0.1 uF surface mount capacitor. In addition, the individual power lines are bypassed by the specific capacitors as required by the FPGA. As pictured on the next page, the 1.25 V line includes 2.2 nF, 4.7 nF, 10 nF, and 4.7 uF capacitors, and the 2.5 V line is bypassed by 22 nF, 10 nF, 4.7 nF, 2.2 nF, 4.7 uF, and 1 nF capacitors. These bypass capacitors help keep the noise in the power lines minimal, by decoupling AC signals from the DC voltage.

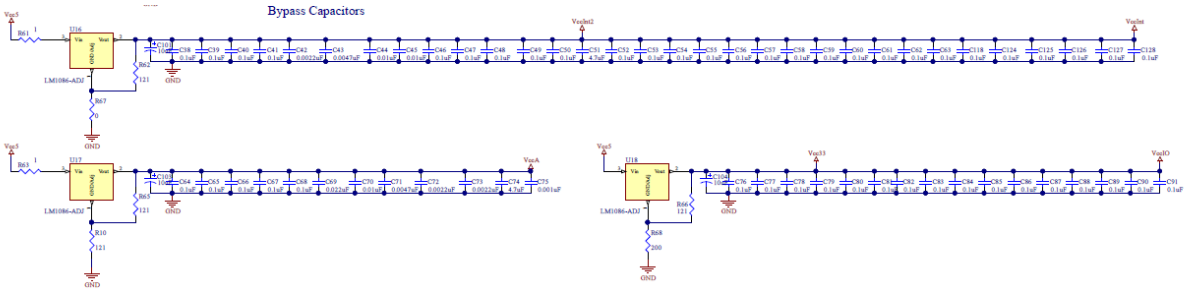


Figure 12: Regulated power lines for FPGA (1.25, 2.5, 3.3 V) and bypass capacitors for each power pin (including special value capacitors)

All default I/O pins on the FPGA as well as the unused special pins are buffered by a 16-bit bus transceiver with 3 state outputs, the SN74LVT16245.

The special pins of the FPGA (U5) were connected in specific configurations to allow for the FPGA to function as needed. The chip wide reset to override all clears on device registers (DEV_CLR) and pin to override all tri-states on device (DEV_OE) are not used, and are thus left unconnected. Since there is only one device configuration, the nCEO pin is also left floating. The nCE pin for active-low chip enable was tied to ground (R25) to enable the device and since only the JTAG and not the AS configuration scheme is being used. The INIT_DONE and CONF_DONE pins of the FPGA are also pulled high by 10 KOhm pull up resistors (R22 and R26). The INIT_DONE is not enabled in the Quartus configuration. CONF_DONE is driven low before and during configuration and released once the initialization cycle starts as an output, and it also goes high after all data is received as an input. Since the JTAG configuration scheme is being used, the nCONFIG pin is connected to VCCIO through a 10 KOhm resistor (R23). In addition nSTATUS is pulled high by a 10 KOhm pull up (R24). It will be driven low immediately after power up and released later. If an error occurs during configuration, it is pulled low. If it is driven low externally, the FPGA will enter an error state. The CLKUSR pin is also not used as an user supplied clock input to synchronize the initialization of mulitple devices, so is connected to ground.

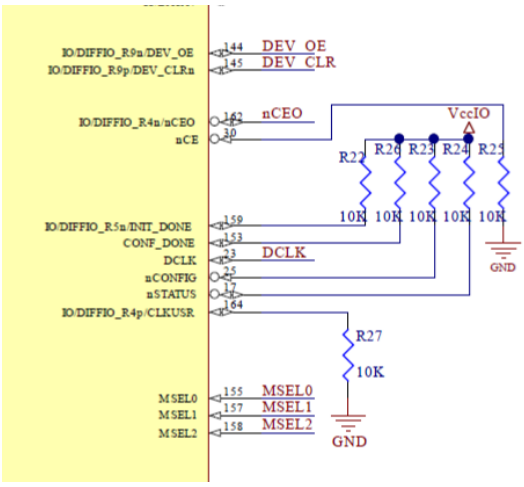


Figure 13: Connection of pins which are not I/O or power/ground in FPGA

A NIOS II processor is run on the FPGA, which is configured in Qsys with the following settings. The NIOS II/f core is chosen for a RISC processor, instruction cache, branch prediction, hardware multiply, hardware divide, barrel shifter, data cache, dynamic branch, prediction, and three M9Ks and cache for memory usage. Embedded multipliers were chosen for hardware multiplication. The reset and exception vector are both stored in EEROM. A 4 Kbyte instruction cache is used with burst transfers disabled. A

2 KByte data cache is used with a 32 Byte data line. Burst transfers are again disabled. The processor includes data and instruction memory mapped masters which are connected to the memory mapped slaves of the other components in the CPU. An internal interrupt controller was used with no exception checking, so there are 32 interrupts for use by other components of the CPU. When considering timing cycles for any process, the delay due to signals travelling through the FPGA was estimated to be around 3 ns.

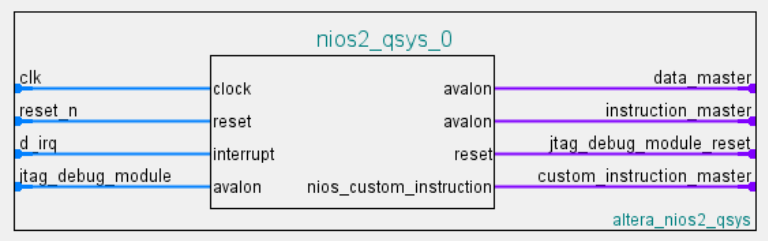


Figure 14: Inputs and outputs of NIOS II processor block in Qsys

1.3.1 MSEL

The mode selects are included in the PCB layout and block diagram of the FPGA hardware. The mode selects are jumpered (JP1, JP2, JP3) to allow the users to choose the configuration scheme they want, by switching between tying each select high or low. Currently, they are configured to allow for fast active serial standard and power on reset, with JTAG taking precendence over AS. Thus MSEL0 is connected to ground, MSEL1 to VCCA, MSEL2 to ground, and MSEL3 is not used. The pins are connected to either VCCA or GND directly to avoid any issues with detecting the wrong configuration scheme due to pull-up resistors.

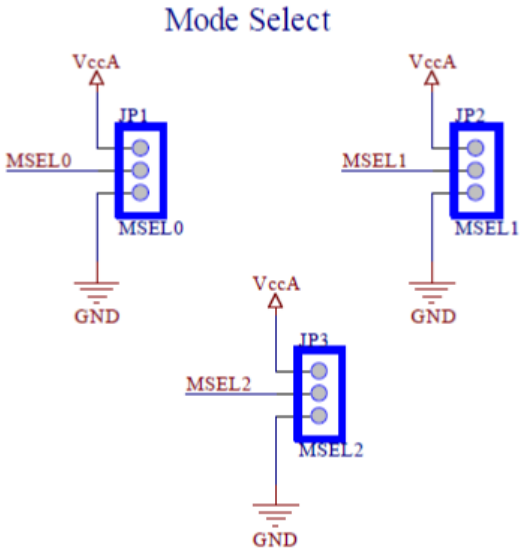


Figure 15: Connection of MSEL lines in schematic to allow for different configurations

1.4 JTAG

A JTAG serial interface (J1) is included to configure and program the FPGA. It is placed on the PCB close to the FPGA to reduce any noise in the signals essential to programming the FPGA. This allows for continual development of the oscillosocpe, as the JTAG can be connected to a USB Blaster to upload new hardware and software designs.

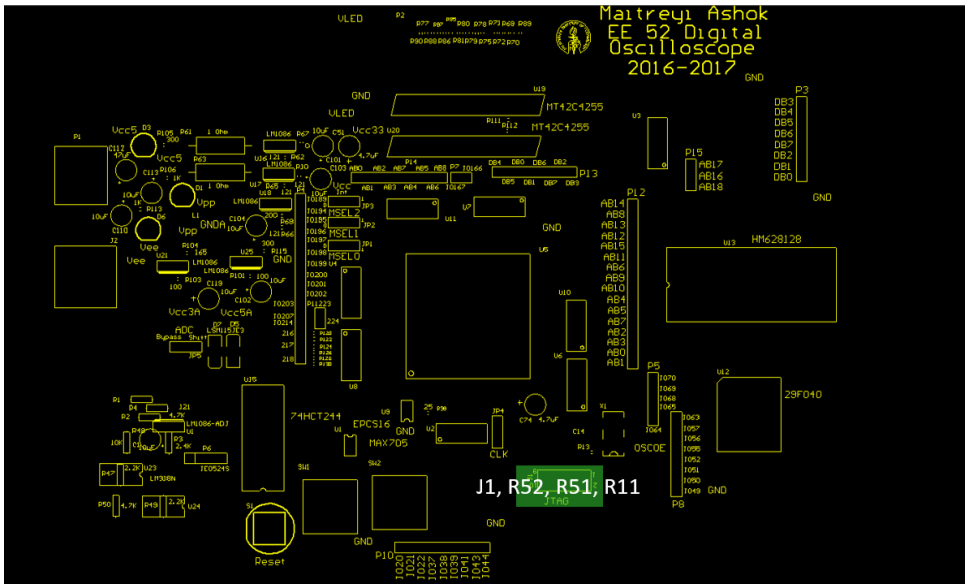


Figure 16: Layout of JTAG component on PCB and designators of all parts in this component

The four signals used from the JTAG are TCK, TDO, TMS, and TDI. TCK is the clock input pin. Every rising edge of TCK, a bit of data is read in from TDI and is written out of TDO. TMS is used to select the next mode for the JTAG every rising edge of TCK. When the oscilloscope is not being programmed, the signals are either pulled high or low (R52, R51, R11), except for TDO which is an output from JTAG.

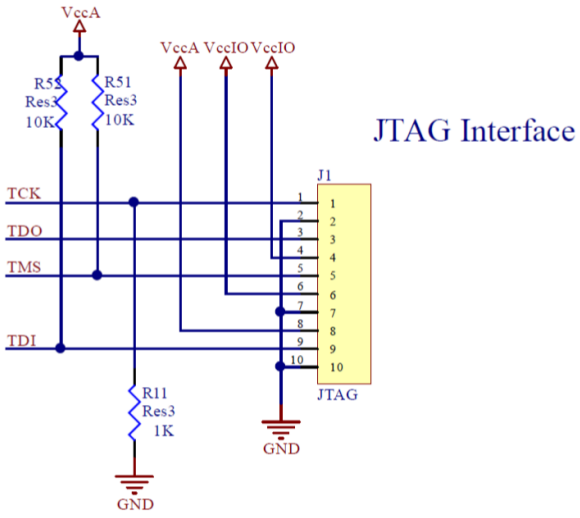


Figure 17: Connection of 4 main signals of JTAG as well as other pins

A level 1 JTAG debug level is used in the NIOS II Processor. This allows for a JTAG target connection,

downloading of software, and software breakpoints. One M9K is also included. The JTAG debug module is included between addresses 0x14_0800 and 0x14_0fff in memory.

In addition, a JTAG Universal asynchronous receiver/transmitter (UART) is included in the CPU configuration, to allow for character data to be streamed serially. The UART buffer is 64 bytes, and the threshold for how many characters remain when the core asserts an interrupt is 8 characters for both the Write FIFO for data to JTAG and the Read FIFO for data from JTAG. The UART is located between address 0x14_1028 and 0x14_102F in memory, and asserts IRQ 0 in the processor when the threshold number of characters is reached.

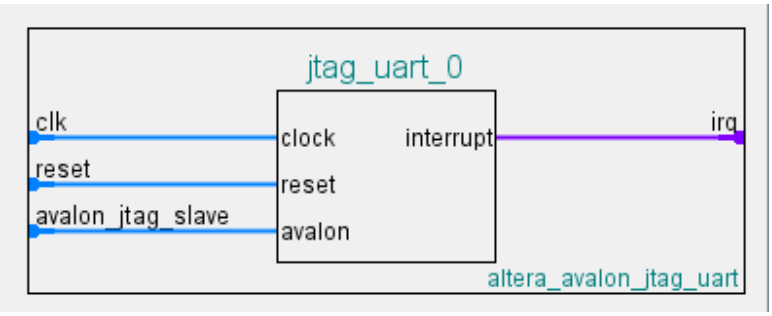


Figure 18: Inputs and outputs of UART in CPU

1.5 Serial ROM

An EPCS16 Serial ROM (U9) is used to store the hardware logic and FPGA CPU configuration when the system is run standalone. The Serial ROM is located near the FPGA Data pins, as seen in the following PCB layout.

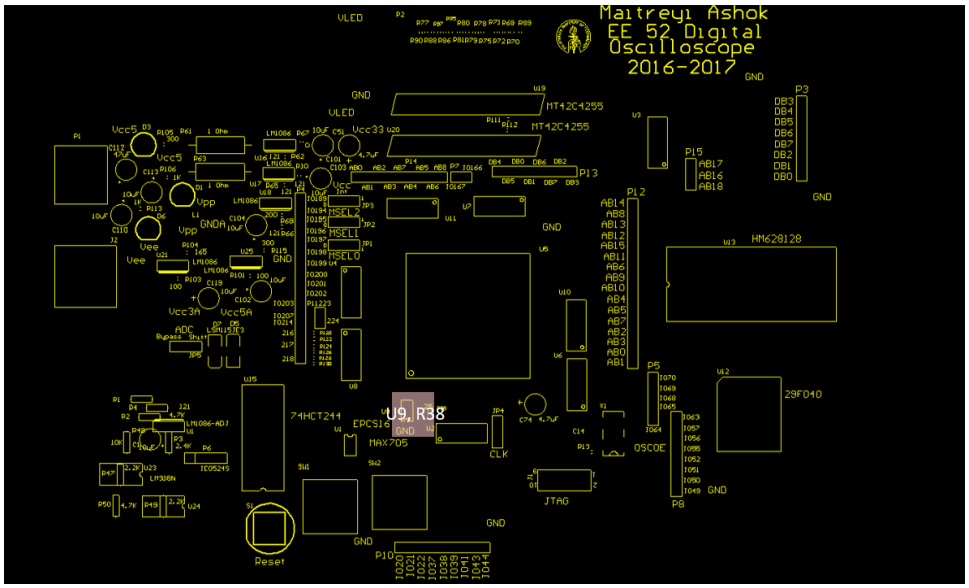


Figure 19: Placement of Serial ROM on PCB and parts that are involved in that component

The SROMnCS signal is connected to Pin 14 on the FPGA, which acts as a control output from the FPGA to the serial configuration device to enable it. Data0 is used to receive bit-wide configuration data. Data1 is also used as a control signal to read configuration data. DCLK is a clock for the serial configuration device.

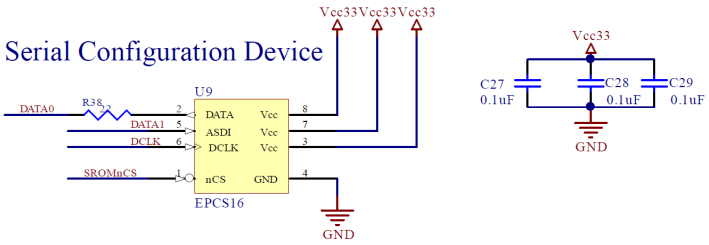


Figure 20: Connection of signals between FPGA and Serial ROM in schematic

The hardware design also includes a serial flash loader to allow for transfer of data to and from the serial configuration device. The noe_in control signal is tied low to enable the Altera Serial Flash Loader IP core.

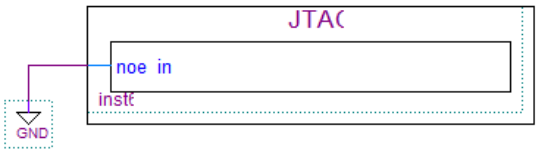


Figure 21: Connection of serial flash loader in hardware logic

1.6 System Reset

A system wide reset is provided using a push button (S1) and a MAX705 reset chip (U22). The MAX705 reset chip is used since it allows for a manual-reset input as well as a reset upon power-up and power-down.

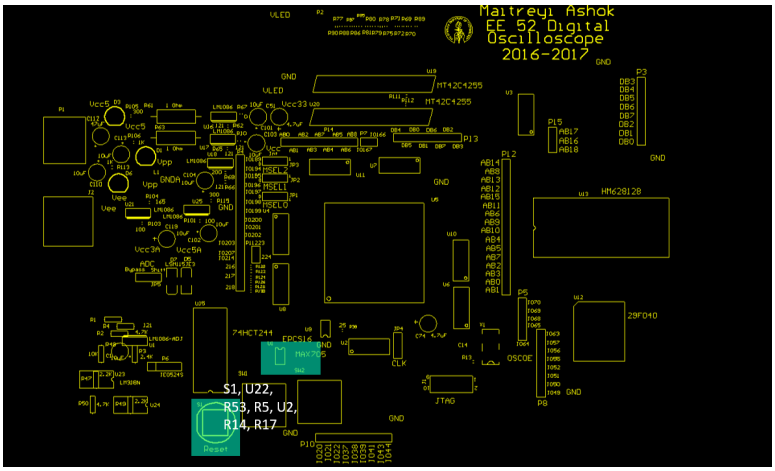


Figure 22: Layout of reset component on PCB and designators of parts involved in this component

An active low reset is used, so that when the push button is pressed, the reset chip receives a low signal on pin 1. The watch dog and reset on power failure features are not used on the oscilloscope. Thus, the watch dog input is left tristated, and the power failure input is pulled high (R5). When the push button is not pressed, the reset input to the chip is pulled high (R53) to avoid the system being reset when not intended.

The reset signal produced by the MAX705 is then input to IO4 of the FPGA (U5) through a buffer configured with an input bank (U2, R14).

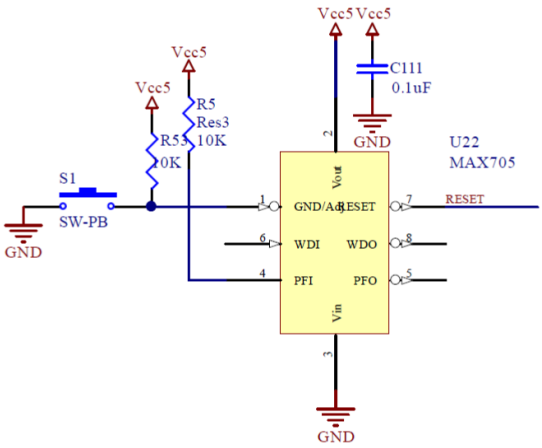


Figure 23: Schematic of reset chip and reset button connections

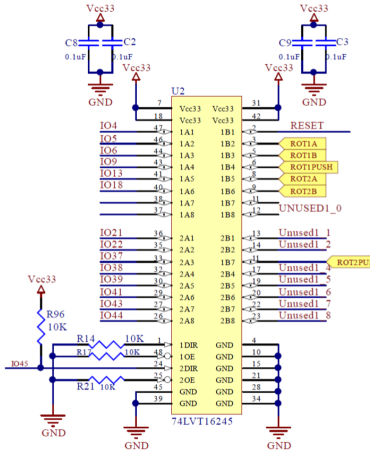


Figure 24: Configuration of buffer to allow reset signal to be input to FPGA

The reset signal is used to reset all hardware logic components, including the CPU, counters, and other storage elements. For components such as the CPU and counters which expect an active high reset signal, the reset signal is inverted before entering the component, as seen for the CPU reset signal.

Figure 25: Input of active low reset signal to active high reset controller in CPU

The CPU reset is controlled by a Merlin reset controller. The controller takes in 1 reset input. The reset is deasserted synchronously but asserted asynchronously. The synchronizer uses 2 register stages to avoid somewhat unstable events propagating. All the CPU components have their reset inputs connected to the output of the Merlin reset controller. The reset vector, containing instructions to execute after the reset event is at address 0x8_0000 to 0x8_0020 in the EEROM.

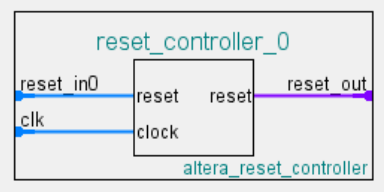


Figure 26: Input and output signals of reset controller in CPU

1.7 Clock

A 30 MHz clock is produced by a OSCOE clockbox (X1).

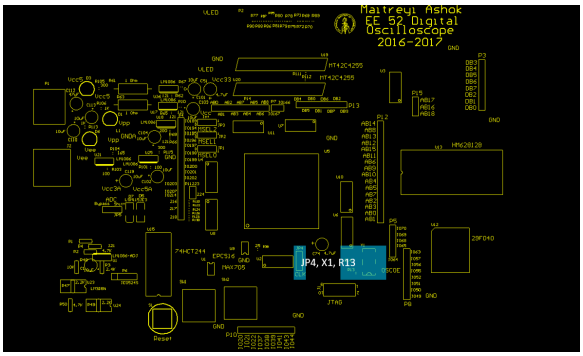


Figure 27: Layout of clock component on PCB and designators of parts in component

The output enable is pulled active (R13), allowing for the clock at 30 MHz frequency to be output. This clock is always input to the FPGA as CLK0. If an additional clock input is necessary to the FPGA, the same clock is input also as CLK1. Otherwise, the CLK1 pin is unused and is tied to ground. This is done through a jumper (JP4) that allows for CLK1 to be shorted to either CLK0 or ground.

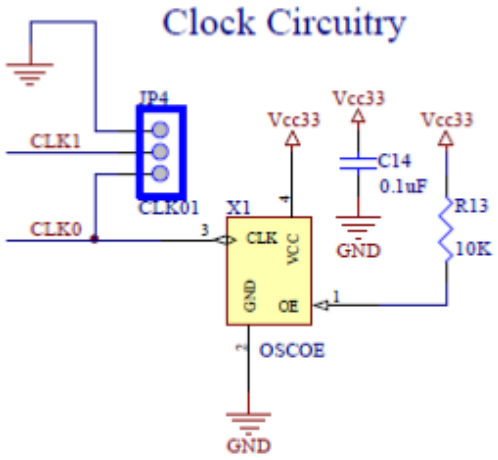


Figure 28: Connecton of signals between clock box and FPGA

CLK0, which is input to pin 31 of the FPGA (U5) is input to the CPU and other hardware logic components that require a clock (counter, flip-flops, etc.). In the CPU, the clock input provides a clock source that is output to all other components in the CPU. The clock also takes the reset controller output and produces a clock reset, which is also used to reset all CPU components (along with the reset chip's output). The clock reset is asserted and deasserted asynchronously.

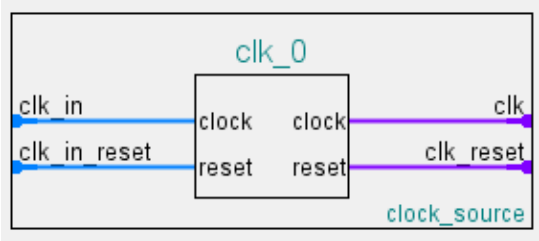


Figure 29: Input and output signals from clock source in CPU

1.8 Buffers

The 74LVT16245 16 bit bus transceiver to drive signals is chosen for its compatibility with most of the ICs used in terms of high and low input and output voltages. In addition, the transceiver allows for the signals to be separated into two banks, which can be controlled separately in terms of output enable and directionality. The buffers are located surrounding the FPGA to allow for signals to travel directly out of the FPGA to the buffer and to the IC, without extra distance to travel which would introduce more noise.

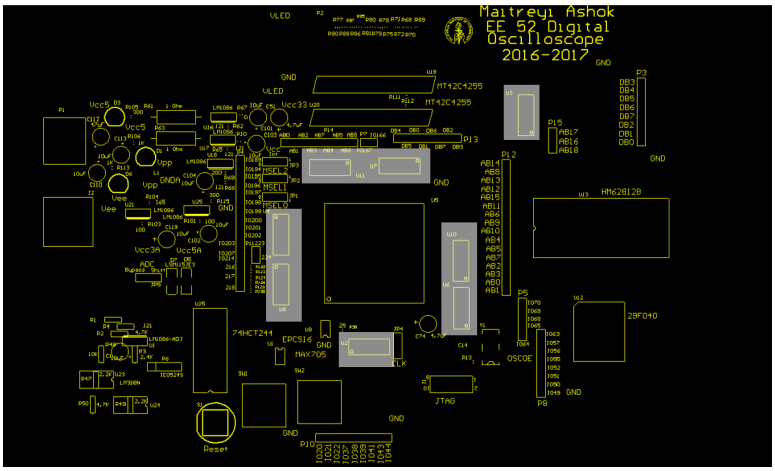


Figure 30: Layout of all buffers in PCB

All buffers have the output enable pulled active (to ground) since there is no need to tristate any of the buffers. Depending on the need of the specific buffer (discussed with the specific components of the scope that they transmit signals for), the direction is pulled high to output from the buffer, pulled low to receive input from the buffer, or controlled by an I/O pin of the FPGA to allow for bidirectionality (data buses). All I/O pins of the FPGA are buffered, to provide enough current at the voltage level the pins are at so that the signals reach the ICs they control without much noise and glitching. The buffers contribute about 3 ns of delay to the transmission of any signal, which is taken into account in the timing of all signals originating from the CPU.

An example of the buffer schematic is shown. To allow for bidirectionality of data to and from the VRAM, an I/O pin of the FPGA controls whether the upper bank is output or input. If the FPGA does not drive the direction of the upper bank, it is pulled low, so that the bank will input data. The lower bank is configured to be output only by pulling the direction high, since all the control signals only need to reach the VRAM and no status is sent back. The output is enabled for both banks by pulling both pins active. All power pins of the buffer are decoupled from AC noise with 0.1 uF capacitors.

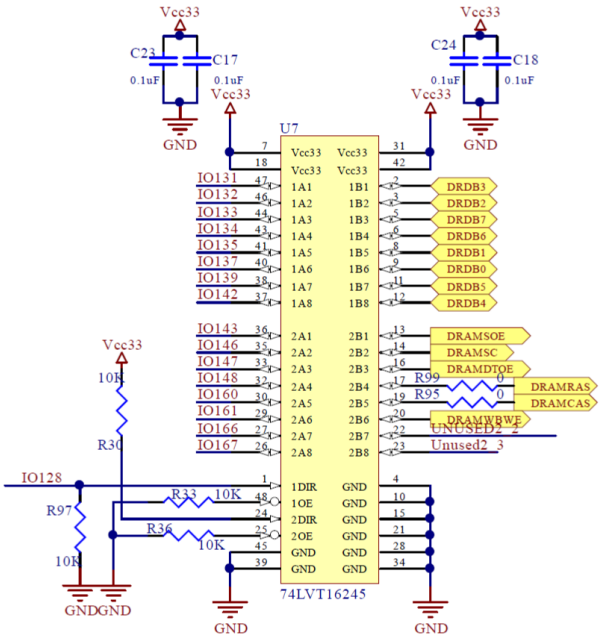


Figure 31: Example of all signals connected to control buffer and input/output data

1.9 Data Storage

To store data and other readable and writable information in the oscilloscope, a 128 KByte HM628128 Hitachi SRAM (U13) is used. This SRAM was chosen for its size, as it is large enough to easily store all the data for the oscilloscope. In addition, it has low standby power dissipation, making the system more efficient.

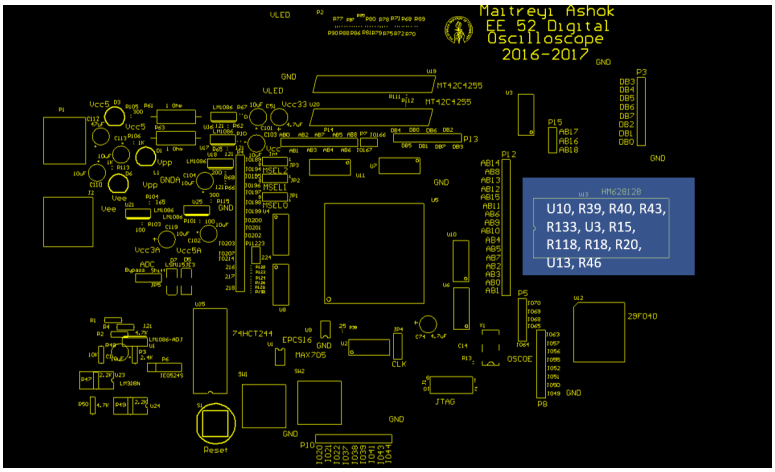


Figure 32: Layout of SRAM on PCB as well as designators of all parts involved in SRAM operation

To store 128 KBytes, there are 17 address bits to access every byte, which are transmitted through the output only buffers (direction pulled high by R40, R39, R15), U10 and U3. The control signals are also transmitted through U3. When output enable (SRAMOE) is active, the SRAM is being read, and when write enable (SRAMWE) is active, the SRAM is being written to.The SRAM uses an active low chip select to access the SRAM (second chip select is unused and pulled permanently active by R46), and is between addresses 0x12_0000 and 0x13_FFFF in the address map.

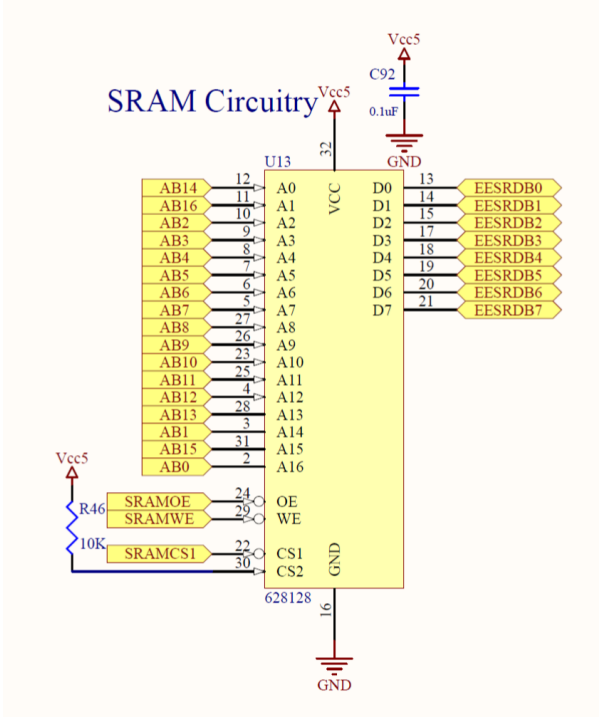


Figure 33: Schematic of all connections of SRAM, including an address bus, data bus, and control signals

The buffers used to transmit signals for the SRAM are pictured below.

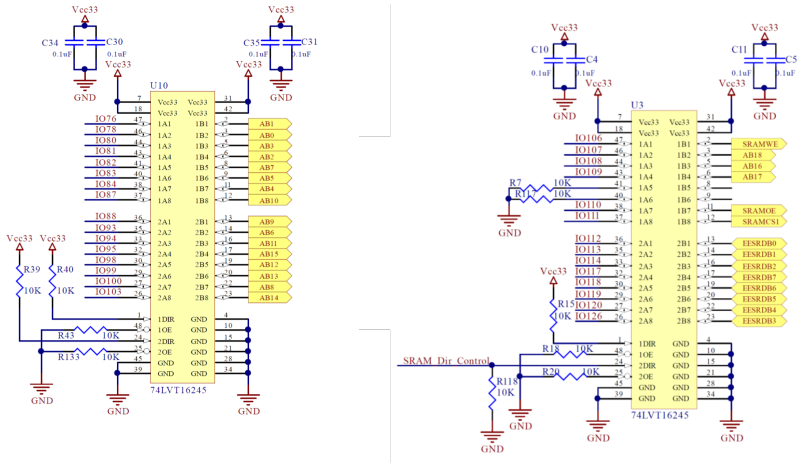


Figure 34: Buffers used to transmit signals between SRAM and FPGA

There are 8 data bits, as each address corresponds to a byte of data. The direction of the bank of U3 that transmits data to and from the SRAM and the FPGA is controlled by SRAM_Dir_Control, an I/O pin of the FPGA. This buffer control was configured in the hardware design to be the inverted write enable. Thus, if write enable was low to allow for a write to SRAM, the buffer direction was pulled high to allow for output of data. If write enable was high to disallow a write to SRAM (but read instead), the buffer direction was pulled low to allow for input of data from SRAM.

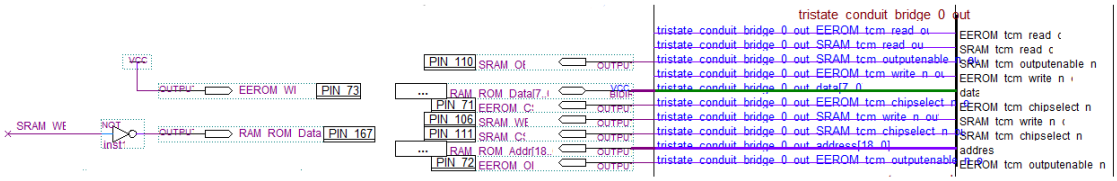


Figure 35: Signals input and output from CPU to pins of FPGA to be sent to SRAM

The SRAM component in the CPU produces output enable and chip select signals which are output directly to the SRAM through the buffer. The address and data busses (shared with EEROM) are also produced in the CPU. The signals and buses are output from the CPU in a tri state conduit bridge. This memory bridge receives input from a pin sharer which combines the signals from both the SRAM and EEROM (with shared buses), and outputs this. These signals are produced according to the following timing cycles.

The read cycle starts when the chip select and output enable signals are active, and the address bus holds the valid address to access. There is a delay of 2 clocks, after which the data becomes valid. The data is valid for 2 clocks, at which time it is read from the data bus. There is a hold time of one clock where the address bus becomes invalid but data must still stay valid. Thus, there are 3 wait states in the read cycle, to ensure that data is valid by the time the CPU tries to read it.

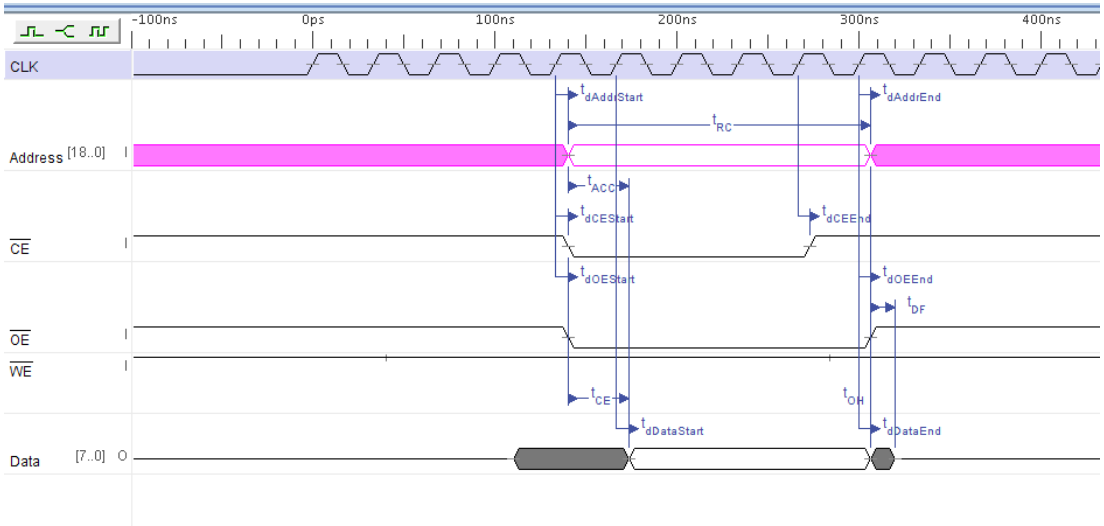


Figure 36: Timing constraints for reading a byte from SRAM

The write cycle starts with the chip select and address becoming valid, while output enable is still enabled, and write enable is not. After one clock, output enable is disabled (as nothing is output during a write) and write enable is active. After one more clock, data becomes valid and stays valid for 2 clocks. There is a hold time of one clock as the data stays valid for 1 clock even after the control signals and address bus become invalid. Thus, there are 2 wait states in the write cycle, after which data becomes valid and is written.

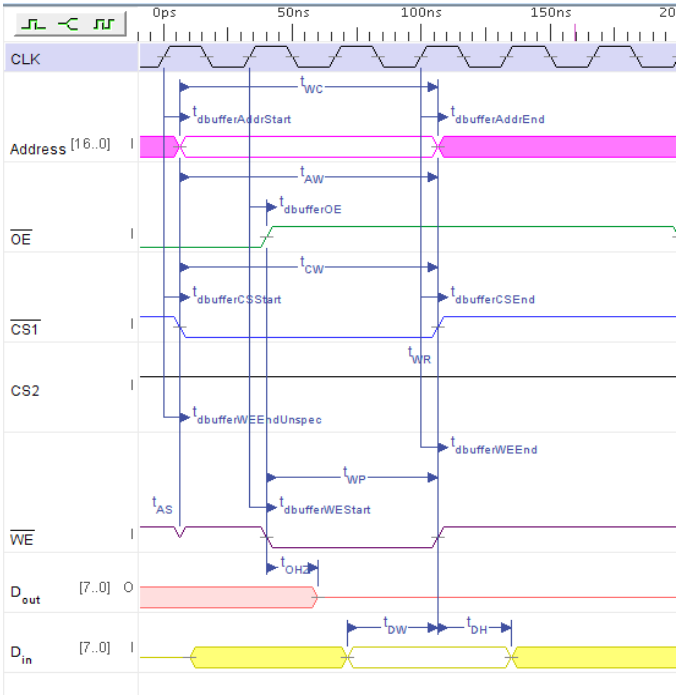


Figure 37: Timing constraints for writing a byte to SRAM

Also, the setup and hold times for the SRAM are 1 clock to ensure that data is valid long enough in advance and stays valid until it is not needed anymore for both the read and write cycles.

In the linker script, the .bss, .heap, .rwdata, and .stack linker sections are stored in the SRAM. Thus, all data that can be read and written to are stored in the SRAM between addresses 0x12_0000 and 0x13_FFFF.

1.10 Code Storage

A 512 KByte EEROM, AMD's AM29F040B, (U12) is used to store code and other read only data for the oscilloscope. This allows for the system to run standalone, with the EEROM programmed with the binary files of all the data and code it needs to store. The 29F040 is chosen for its large size, holding enough bytes to store all current information and for continual development of the oscilloscope. In addition, the fast access times and low power consumption allow for high performance of the scope.

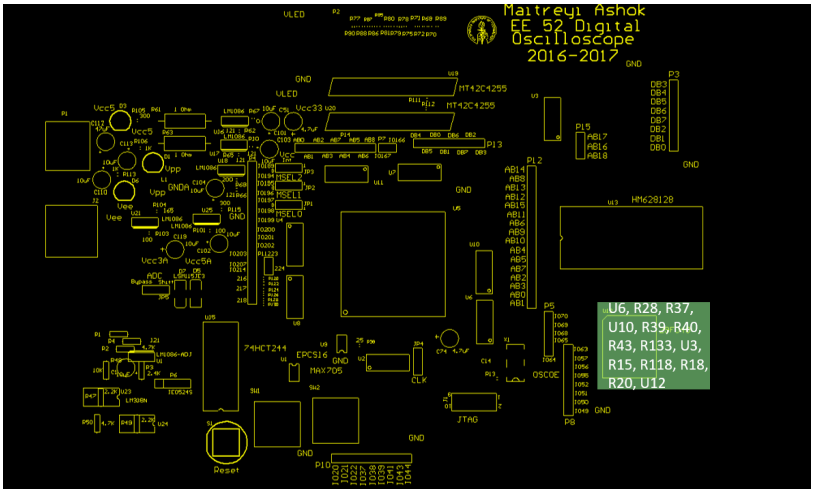


Figure 38: Layout of EEROM on PCB as well as designators of parts that are involved in this component

To access 512 KBytes, 19 address bits are necessary, and each address refers to one byte (thus 8 bit wide data bus). The EEROM is located at addresses 0x8_0000 to 0xF_FFFF in the address map. In addition, there are 3 control signals to control the EEROM. The chip select (EEROMCE) must be active whenever the ROM is being accessed. The output enable signal (EEROMOE) is active when the ROM is being read. The write enable signal (EEROMWE) is permanently inactive since the ROM is read only, and will be tied inactive in the hardware logic for the I/O pin output.

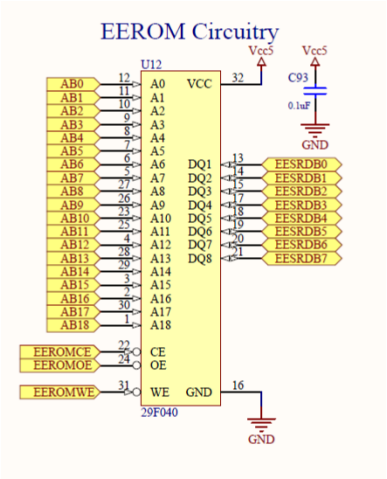


Figure 39: Signals entering and leaving EEROM shown in schematic

The EEROM shares a data and address bus, and thus transmits data and addresses in the same buffers (U3 and U10) as pictured in the previous data storage section. The control signals for the EEROM are transmitted in a separate buffer (U6) which is pictured below.

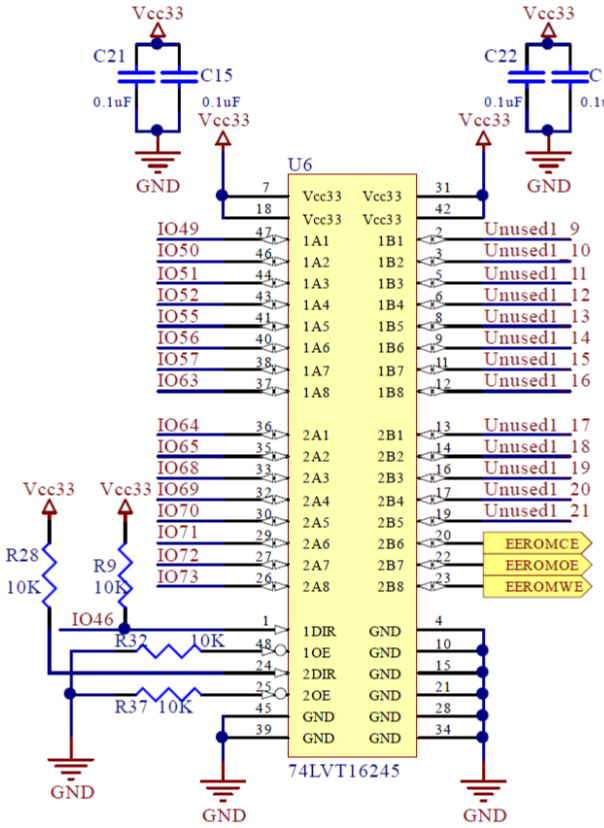


Figure 40: Buffer used to transmit control signals from FPGA to EEROM

As seen in the Quartus configuration, the write enable is tied inactive for the ROM. In addition, the direction of the data buffer is again the inverted write enable of SRAM. If the EEROM is being read, the SRAM_WE is inactive, meaning that the buffer direction pin will be low. Thus, the buffer will be configured for input of data. The data and address buses are shared with the SRAM using a pin sharer as described in the previous section. The output enable and chip select signals are produced by the CPU according to the following timing cycle.

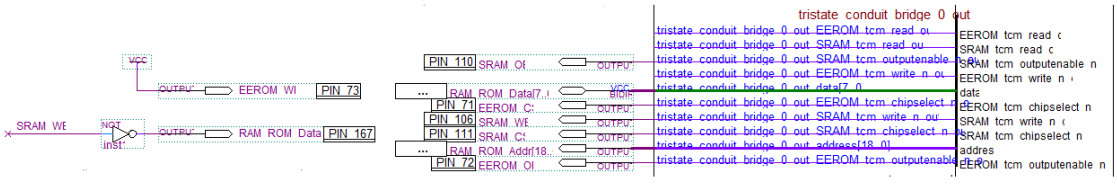


Figure 41: Signals input and output from CPU to pins of FPGA to be sent to EEROM

The read cycle starts when the address bus becomes valid and the chip select and output enable become active. One clock later, the data becomes valid and stays valid for 4 clocks. There is no hold time for the data, since all the control signals and address bus become invalid in the same clock that data becomes invalid. Thus, the read wait time is 3 clocks (1 extra clock to avoid any issues with buffer/FPGA delays).

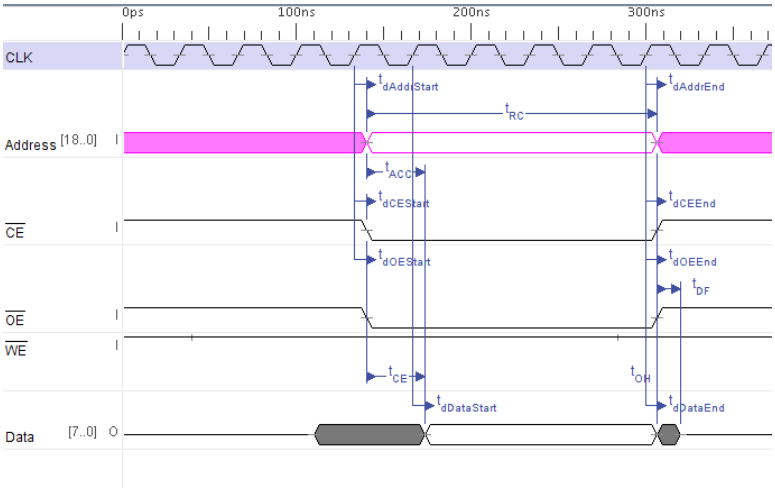


Figure 42: Timing constraints for reading a byte from EEROM

The processor's reset and exception vectors are stored at the start of the EEROM, since these are non-writable addresses the processor must jump to when the system is reset or an exception is handled. In addition, the linker sections .rodata (read only data) and .text (code) are stored in the EEROM.

1.11 Rotary Encoders

There are two rotary encoders (SW1, SW2) with push buttons which are dials used for user input for the oscilloscope. These allow the user to select options for both the display and analog components based on a menu shown on the display.

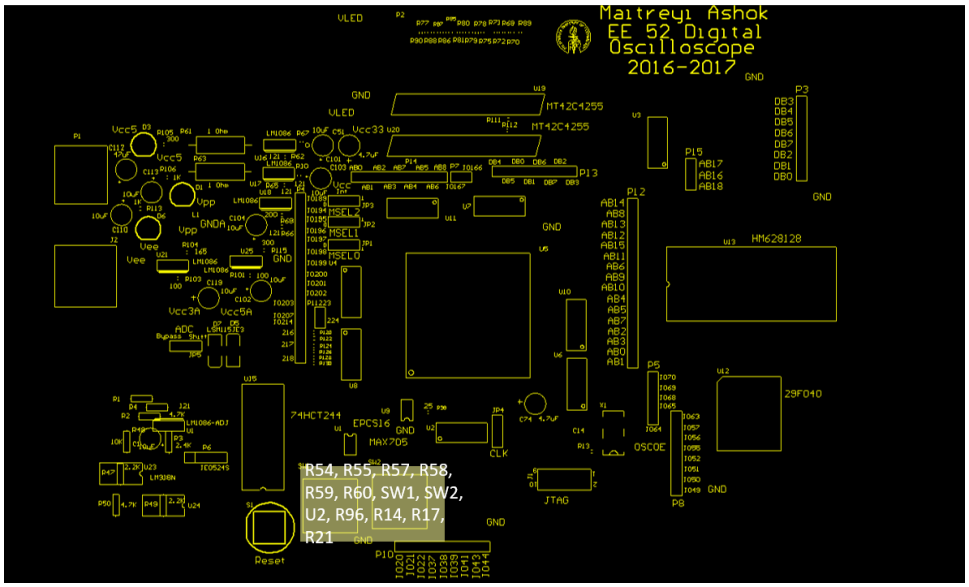


Figure 43: Layout of rotary encoders and associated parts on PCB

The encoders are connected in the configuration shown below. The push buttons are normally pulled high (R59, R60) and are read as a 1. However, when the button is pressed, the push button pin is shorted to ground, causing it to register as a 0. The two pins for determining the direction of turning are pulled high (R54, R57, R55, R58) when the encoder is not turned in any direction. When turned, they follow a pattern of pin A getting shorted to the common pin (tied to ground) before or after pin B at each detent for clockwise or counterclockwise turns, respectively.

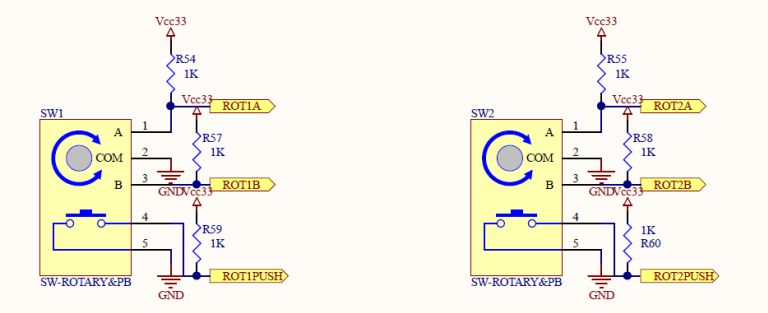


Figure 44: Schematic of rotary encoders with push buttons, and signals output from them

The output of these 3 pins for each encoder then enters the FPGA through a buffer (U2) configured for input. The upper bank has direction tied low, to allow for input. The lower input is controlled by the pin IO45 of the FPGA.

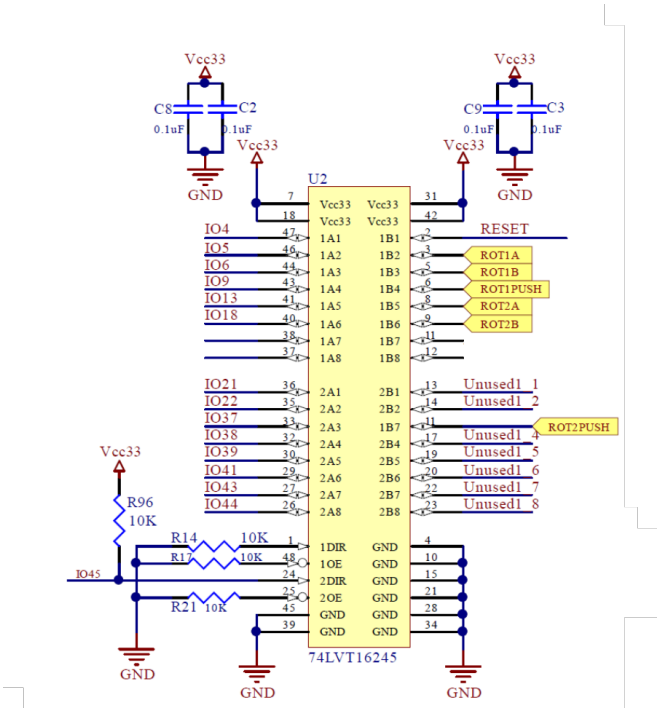


Figure 45: Buffer used to transmit signals from rotary encoder to FPGA

The following Quartus configuration shows how the I/O pin is controlled and the outputs of the rotary encoders are input to the FPGA. More details of the decoding and debouncing logic in the Rotary_Encoders block will be discussed in the next section.

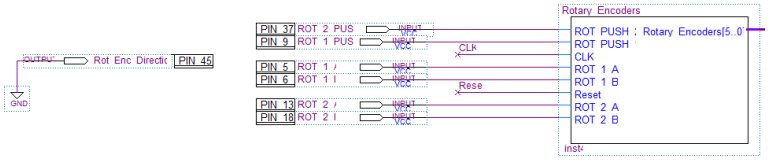


Figure 46: How rotary encoder inputs are sent to rotary encoder block in hardware logic

1.12 Rotary Encoder Logic

The push buttons of the two rotary encoders are debounced to ensure that every press will only be counted once. By only recording a press after it has been pushed down for a certain amount of time, we ensure that glitches between the pressed and unpressed state during the push down and release are not counted as presses. This debounce time is chosen to be 100 ms, since any users will hold the encoder down for longer than one tenth of a second if they mean to press it. This wait for the debounce time is implemented as a counter when the push button is active compared to the threshold debounce time. When the debounce time is reached, the push event is latched using a set-reset flip flop. The flip flop is reset when the push button is released (value of the pin becomes high). Since it is all right to reset the flip flop multiple times as the encoder glitches during the release, this value does not need to be debounced. The following hardware logic is used in Quartus.

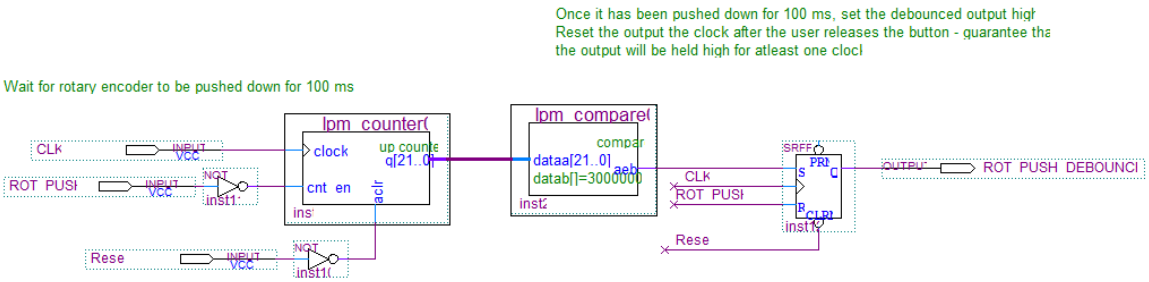


Figure 47: Hardware logic to debounce rotary encoder push buttons

The turns of the encoders are decoded using a finite state machine. Depending on what values are read from pin A and pin B in a succession, the direction of turning is determined. The following state diagram is followed in the transistions between states and corresponding outputs. The outputs are encoded as the lower two state bits themselves: clockwise is the second lowest bit, and counterclockwise is the lowest bit.

Pin A	0				1			
Pin B	0		1		0		1	
Current State	Next State	Output	Next State	Output	Next State	Output	Next State	Output
START	START	-	POSS_CW	-	POSS_CC	-	START	-
POSS_CW	FINISH_CW	-	POSS_CW	-	START	-	START	-
POSS_CC	FINISH_CC	-	START	-	POSS_CC	-	START	-
FINISH_CW	FINISH_CW	-	FINISH_CW	-	FINISH_CW	-	HOLD_CW_1	-
FINISH_CC	FINISH_CC	-	FINISH_CC	-	FINISH_CC	-	HOLD_CC_1	-
HOLD_CW_1	HOLD_CW_2	Clockwise	HOLD_CW_2	Clockwise	HOLD_CW_2	Clockwise	HOLD_CW_2	Clockwise
HOLD_CC_1	HOLD_CC_2	Counterclockwise	HOLD_CC_2	Counterclockwise	HOLD_CC_2	Counterclockwise	HOLD_CC_2	Counterclockwise
HOLD_CW_2	HOLD_CW_3	Clockwise	HOLD_CW_3	Clockwise	HOLD_CW_3	Clockwise	HOLD_CW_3	Clockwise
HOLD_CC_2	HOLD_CC_3	Counterclockwise	HOLD_CC_3	Counterclockwise	HOLD_CC_3	Counterclockwise	HOLD_CC_3	Counterclockwise
HOLD_CW_3	START	Clockwise	START	Clockwise	START	Clockwise	START	Clockwise
HOLD_CC_3	START	Counterclockwise	START	Counterclockwise	START	Counterclockwise	START	Counterclockwise
OTHERS	START	-	START	-	START	-	START	-

Figure 48: State table for hardware decoding of rotary encoder turns

How the state machine works is that both Pin A and B start out as 1 (they are pulled high if not driven). In one turn (click of a detent), pin A and B go through a complete cycle from 1 to 0 back to 1. If Pin A hits 0 (that pin touches common pin first) before Pin B has changed, then the encoder is turning clockwise. If Pin B hits 0 before Pin A, then the encoder is turning counterclockwise. Since one of the pins changing to 0 might just be a glitch, the encoder is only counted as having turned in either direction if the pattern of both pins reaching 0 and then both pins eventually reaching 1 (with possible glitches in the middle) is detected. Once both pins reach their initial state of 1, the clockwise or counterclockwise signal will be held high for 3 clocks to ensure that the CPU receives an interrupt request and can read the value in the edge capture register.

This is implemented in the following VHDL logic

```
--------------------------------------------------------------------------------
--
--  Rotary Encoder Decoding
--
--  This is an implementation of a decoder for the rotary encoders in the
--  digital oscilloscope system.  There are two inputs to the system, the
--  Pin A and Pin B of the rotary encoders. Depending on whether Pin A
--  advances before Pin B, or the other way, the clockwise or counterclockwise
--  rotation is determined. If Pin A advances from 1 to 0 before Pin B does,
--  the encoder is travelling clockwise. If Pin B advances before, then the
--  encoder is travelling counterclockwise. This is subject to the encoders
--  travelling back to both pins being 0 and both pins being 1, due to
--  the previous sequence maybe being a result of bouncing between pins.
--  The two outputs are a clockwise and counterclockwise signal of which
--  direction the rotary encoder is travelling (if either), which is used
--  to generate edge sensitive interrupts in the CPU.
--
--
--  Revision History:
--     13 May 17  Maitreyi Ashok    Initial revision.
--     19 May 17  Maitreyi Ashok    Changed sequence for full debounce
--
--------------------------------------------------------------------------------


-- bring in the necessary packages
library ieee;
use ieee.std_logic_1164.all;

--
--  Rotary Encoder Decoding entity declaration
--

entity rot_decoder is
    port (
            PIN_A       :   in std_logic;       -- pin A of the encoder
            PIN_B       :  in std_logic;        -- pin B of the encoder
            CLK         :  in std_logic;        -- clock
            Reset       :   in std_logic;       -- reset the system

            Clockwise           :   out std_logic;  -- clockwise turn has occurred
            CounterClockwise    :   out std_logic   -- counterclockwise turn has occurred
    );
end rot_decoder;

--
--  Rotary Encoder Decoding Moore State Machine
--     Simple Architecture
--
--  This architecture uses manual assignment of state bits.
--  This minimizes the need to decode the output of a state
--  as the outputs are the lower state bits themselves.
--

architecture assign_dec_statebits   of rot_decoder is
    subtype states is std_logic_vector(4 downto 0); -- state type

    constant    START       : states := "00000";    -- waiting for turn of encoder
    constant    POSS_CW     : states := "00100";    -- pin A started changing before B
    constant    POSS_CC     : states := "01000";    -- pin B started changing before A
    constant    FINISH_CW   : states := "01100";    -- guarantee that travel clockwise
    constant    FINISH_CC   : states := "10000";    -- guarantee that travel clockwise
    constant    HOLD_CW_1   : states := "00110";    -- after finish complete rotation in
    constant    HOLD_CC_1   : states := "00101";    --   either direction, hold the
```

```
    constant    HOLD_CW_2   : states := "01010";    --   clockwise or counterclockwise
    constant    HOLD_CC_2   : states := "01001";    --   signal high for 3 clocks so
    constant    HOLD_CW_3   : states := "01110";    --   it is recognized by the CPU
    constant    HOLD_CC_3   : states := "01101";


    signal  CurrentState    : states;   -- current state
    signal  NextState       : states;   -- next state

begin

    Clockwise   <= CurrentState(1);     -- clockwise rotation is second lowest bit
    CounterClockwise <= CurrentState(0);-- counterclockwise rotation is lowest bit

    -- compute the next state (function of current state and inputs)
    transition: process (PIN_A, PIN_B, Reset)
    begin
            case CurrentState is            -- do the state transition and output

                    when    START =>
                        if (PIN_A = '0' and PIN_B = '1') then   -- if pin A starts changing
                            NextState <= POSS_CW;               -- possibly moving clockwise
                        elsif (PIN_A = '1' and PIN_B = '0') then-- if pin B starts changing
                            NextState <= POSS_CC;               -- possibly moving
                                                                -- counterclockwise
                        else
                            NextState <= START;                 -- turn not possible yet
                        end if;
                    when    POSS_CW =>
                        if (PIN_A = '0' and PIN_B = '1') then   -- if pin A continues changing
                            NextState <= POSS_CW;               -- but not pin B, still
                                                                -- possibly moving clockwise
                        elsif (PIN_A = '0' and PIN_B = '0') then-- if pin A and pin B both
                            NextState <= FINISH_CW;             -- changed then definitely
                                                                -- moved clockwise
                        else                                    -- otherwise encoder just
                            NextState <= START;                 -- bounced move back to
                        end if;                                 -- waiting for a turn
                    when    POSS_CC =>
                        if (PIN_A = '1' and PIN_B = '0') then   -- if pin B continues changing
                            NextState <= POSS_CC;               -- but not pin A, still possibly
                                                                -- moving counterclockwise
                        elsif (PIN_A = '0' and PIN_B = '0') then-- if pin A and B both changed
                            NextState <= FINISH_CC;             -- then encoder definitely
                                                                -- moved counterclockwise
                        else                                    -- otherwise encoder just
                            NextState <= START;                 -- bounced move back to waiting
                        end if;                                 -- for a turn
                    when    FINISH_CW =>
                        if (PIN_A = '0' and PIN_B = '0') then   -- if both pin A and pin B are
                            NextState <= FINISH_CW;             -- at middle value wait for them
                                                                -- to finish rotation before
                                                                -- sending clockwise rotation
                                                               high
                        elsif (PIN_A = '1' and PIN_B = '1') then-- if finished complete
                            NextState <= HOLD_CW_1;             -- rotation, hold clockwise
                                                                -- signal high
                        else                                    -- if travelling to any
                            NextState <= FINISH_CW;             -- intermediate value hold this
                        end if;                                 -- state until rotation completed
                    when    FINISH_CC =>
                        if (PIN_A = '0' and PIN_B = '0') then   -- if both pin A and B are at
                            NextState <= FINISH_CC;             -- middle value wait for end of
                                                                -- rotation before sending
```

```
                                                    -- counterclockwise rotation
                                                    -- high
                    elsif (PIN_A = '1' and PIN_B = '1') then-- if finished complete
                        NextState <= HOLD_CC_1;     -- rotation hold counterclockwise
                                                    -- signal high
                    else                            -- if travelling to any
                        NextState <= FINISH_CC;     -- intermediate value hold this
                    end if;                         -- state until rotation completed

                when HOLD_CW_1 =>               -- hold the clockwise signal for the second
                    NextState <= HOLD_CW_2;     -- clock
                when HOLD_CW_2 =>               -- hold the clockwise signal for the third
                    NextState <= HOLD_CW_3;     -- clock
                when HOLD_CW_3 =>               -- after three clocks go back to waiting
                    NextState <= START;         -- for a turn of the rotary encoder
                when HOLD_CC_1 =>               -- hold the counterclockwise signal for
                    NextState <= HOLD_CC_2;     -- the second clock
                when HOLD_CC_2 =>               -- hold the counterclockwise signal for
                    NextState <= HOLD_CC_3;     -- the third clock
                when HOLD_CC_3 =>               -- after three clocks go back to waiting
                    NextState <= START;         -- for a turn of the rotary encoder
                when    OTHERS =>               -- if any other state, go to waiting
                    NextState <= START;         -- for a turn

        end case;
    end process transition;

    process (CLK)
    begin
        if Reset = '0' then                         -- reset overrides everything
            CurrentState <= START;                  -- if reset go to wait for a turn
                                                    -- of rotary encoder
        elsif CLK = '1' and rising_edge(CLK) then   -- only change states on rising edges
                                                    -- of clock
            CurrentState <= NextState;              -- save the new state information
        end if;
    end process;

end assign_dec_statebits;
```

The decoders and debouncers for both rotary encoders are combined to produce a 6 bit value, as shown in this block diagram.

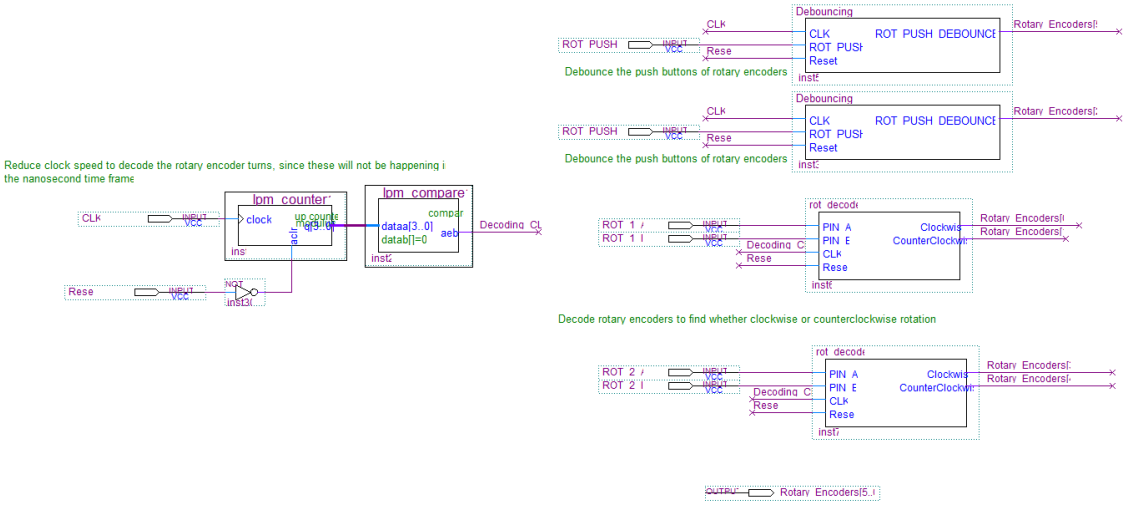


Figure 49: Hardware logic blocks for debouncing and decoding of rotary encoders

The 6 bit value is an input to a peripheral I/O to the CPU. The PIO is located between addresses 0x14_1010 and 0x14_101F in the address map. The edge capture register (with bit-clearing enabled) synchronously captures rising edges, to generate an IRQ when any unmasked bit in the edge-capture register is logic true. This way, any push button or clockwise or counterclockwise turn that is detected will generate an interrupt in the CPU, to be handled in software. The rotary encoder PIO is IRQ 1 in the NIOS II processor's data master.

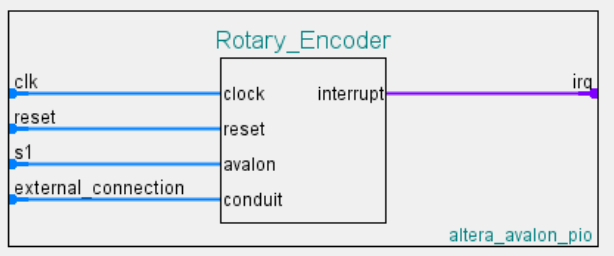


Figure 50: CPU configuration of rotary encoder PIO (with interrupt)

1.13 VRAM

To get color data for each pixel in the graphics LCD, 2 of Micron's MT42C4255 VRAMs (U19, U20) are used. Each VRAM has a 256K x 4 DRAM and a 512 x 4 SAM. Thus, each address points to a nibble of data. Using two VRAMs with the same addressing, we can get 8 bits of data for each pixel, providing enough color for the oscilloscope display. The MT42C4255 is chosen because of its size matching well with that of the display. Since there are 512 rows and columns, the 480 x 272 pixels of the display can be mapped easily into the VRAM with each row of the display being one row in the VRAM (with some unused addresses in the VRAM). In addition, the VRAM has small access times of 80 ns for a random access and 25 ns for serial access. This will allow for changes in the display to happen more seamlessly.

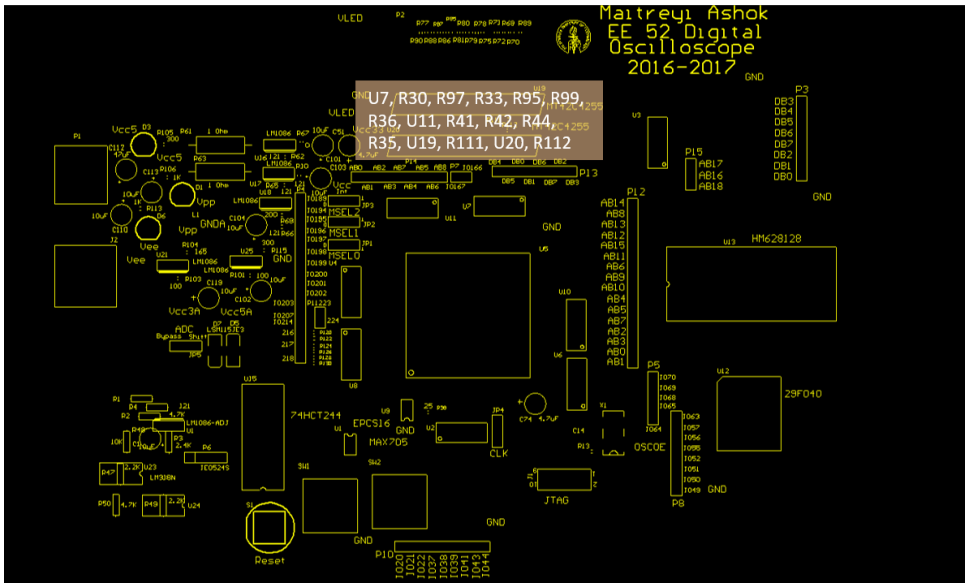


Figure 51: Layout of VRAMs on PCB as well as designators of associated parts

Each VRAM has 9 address bits. These address bits will be mulitplexed between the row and column numbers to allow for access of 256 KBytes (with 18 total address bits). Each VRAM has 4 data bits, so there are 8 data bits total. The control signals for non-serial access are the row address strobe (DRAMRAS) and column address strobe (DRAMCAS) as well as the transfer/output enable (DRAMDTOE) and write enable (DRAMWBWE). RAS and CAS define whether the address provided is for the row or column, and the order of RAS and CAS becoming active defines the current cycle. When TR/OE is active, then an internal transfer is enabled or the DRAM output buffers are enabled. When WE becomes active, a write cycle is selected (rather than a read cycle).

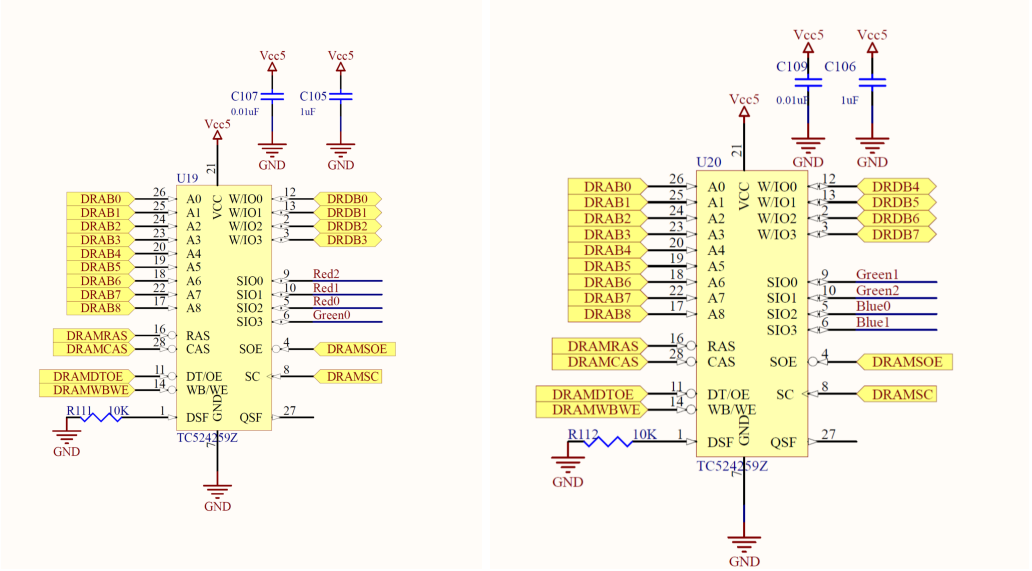


Figure 52: Schematic of signals input/output from both VRAMs between VRAM, FPGA, and display

The special function write cycles are not used, so DSF is pulled low (or inactive) through R111 and R112. There is no use for the split SAM status, so the QSF signal is left floating.

For serial access, the SAM is addressed by nibles, and each VRAM has 4 bits of color that can be read serially from the SRAM. The serial output enable is active through DRAMSOE (made permanently active in hardware logic). The serial clock (DRAMSC) is defined in hardware logic so that one byte of data leaves the two VRAMs to the display every rising edge of the clock. This clock will be matched to the pixel clock for the display, so that one pixel is received by the display during the data valid time in the display and is shown on the display.

Other than read, write, serial output, and row transfer from RAM to SAM, the VRAM is also made of passive elements rather than transistors. Thus, it must be continually refreshed so the capacitors don't discharge and the high voltages drop to low voltages.

Two buffers (U7, U11) are used to transmit all the signals and busses for the DRAM. The data bus is made bidirectional through IO128 from the FPGA controlling whether the buffer is output or input. Like with the SRAM, the directionality is controlled by the inverted write enable signal. If the write enable is inactive, then the directionality is pulled low so that the buffer inputs data. If write enable is active, the directionality is pulled high so the buffer outputs data. If the direction is not driven, then the upper bank inputs data by default (R97). The lower bank of the buffer has its directionality pulled high (R30) so it always outputs the control signals to the VRAM. The address bus is in U11, and is configured for output only (R41, R45) since there is never any need to input an address.

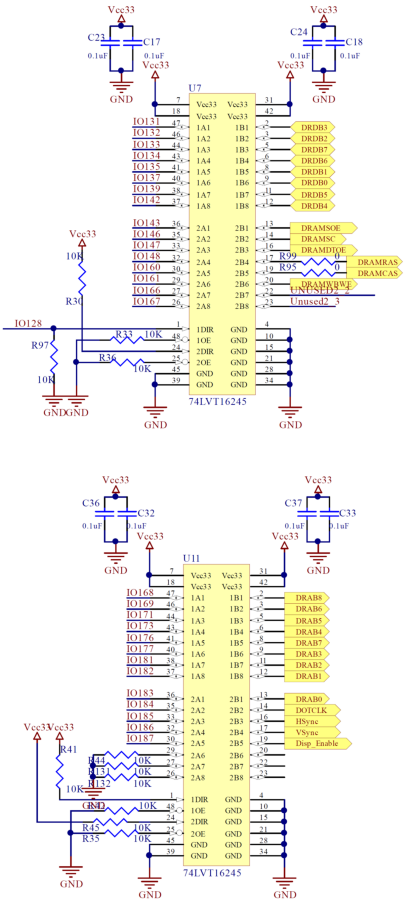


Figure 53: Buffers used to transmit signals between FPGA and VRAMs

There is no chip select for the VRAM as the RAS and CAS signals become active to signify an access of the VRAM. The VRAM is located between addresss 0x4_0000 and 0x7_FFFF in the address map.

The VRAM interacts with the controller, CPU, display, and display controller as in the following block diagram, with the following key for signals:

Blue - Address
Green - Data
Purple - Control

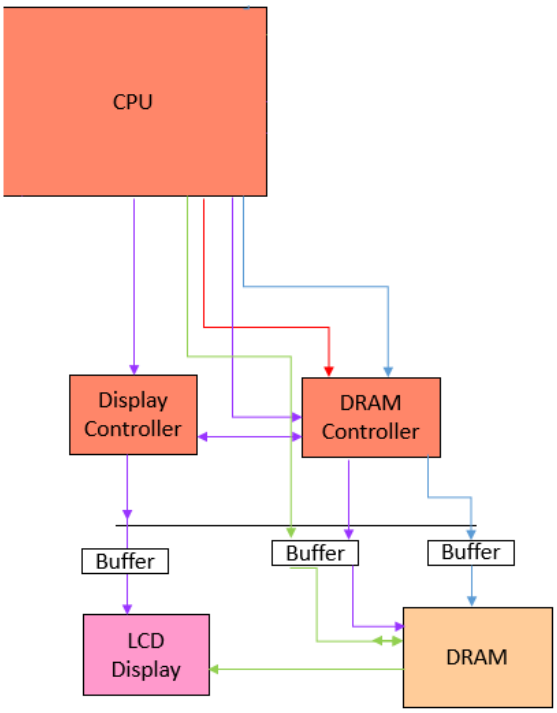


Figure 54: Interaction of signals between CPU, VRAM blocks, and display blocks

The Quartus configuration of the signals to the VRAM is shown. The CPU generates the 18 bit address, data, write enable, and internal chip select signal. These are input to a controller described in the next section, which will generate the signals necessary for the VRAM (9 bit address, RAS, CAS, WE, and OE). The signals are output from the CPU through a tri-state conduit bridge, which converts the tri-state encoded signals into bidirectional ones.

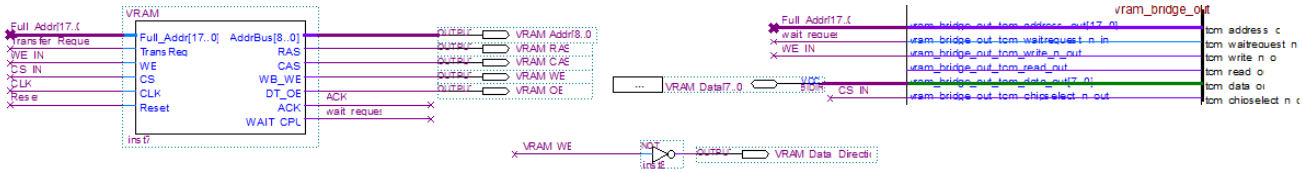


Figure 55: Configuration of VRAM and CPU blocks and signals in Quartus

1.14 VRAM Controller

The VRAM controller is responsible for driving the read, write, row transfer, and refresh cycles of the VRAM by generating the necessary signals for the VRAM. The timings for these cycles is described below.

1.14.1 VRAM Read

The read cycle starts out with the 9 row address bits clocked in when RAS becomes active. Then, when CAS becomes low the column address bits are clocked in the same address bus. The write enable is inactive the entire cycle since write operations can not be performed when reading data. The output enable must be inactive when RAS becomes active to indicate that a read/write operation must happen (and not a row transfer). The data becomes valid one clock after the column address is clocked into the address bus.

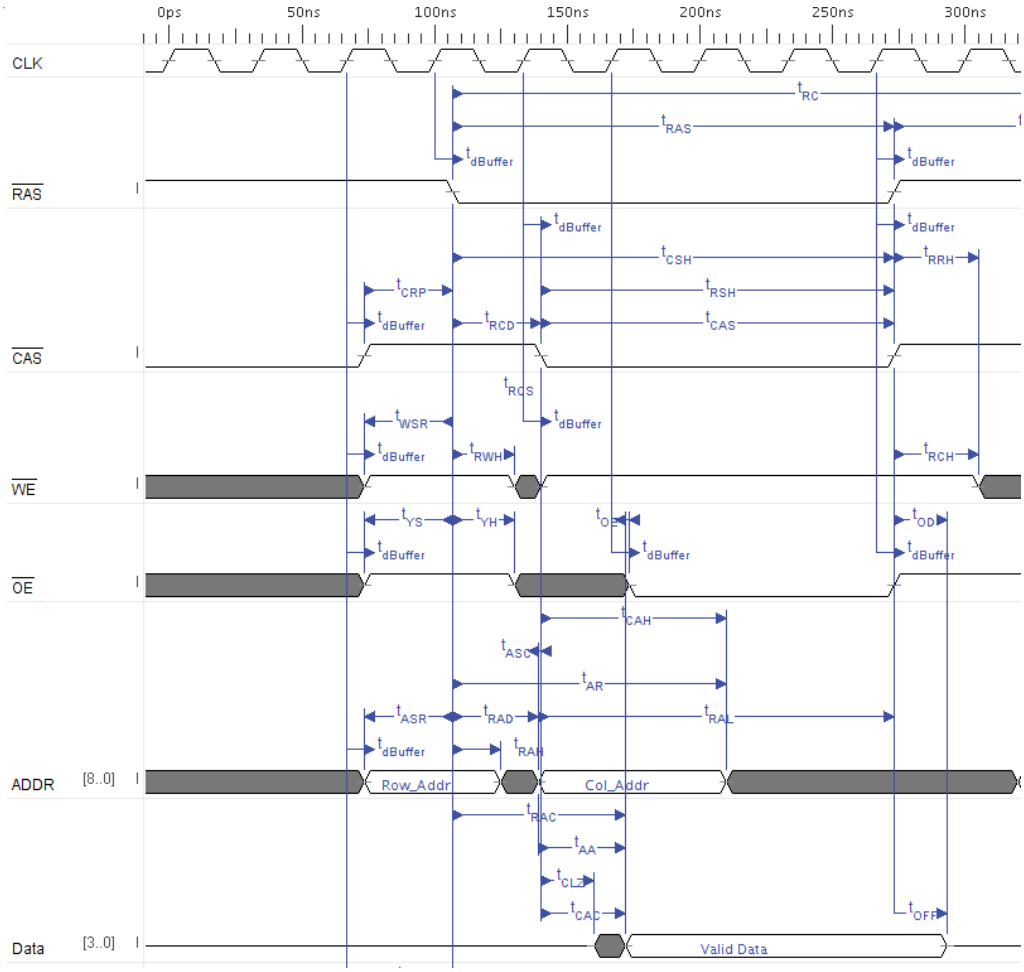


Figure 56: Timing constraints for reading a byte from DRAM portion of VRAM

This is summarized in the following table that is used to build the read cycle portion of the controller state machine. The Row/Col signal is 1 when the row address should be in the address bus, and 0 when the column addresss should be in the address bus. The RdWr/Tr signifies that a read or write cycle is taking place and not a transfer cycle. Finally, the wait signal is used to signify to the CPU when the data should become valid. The signal is active low, so for every clock other than the one before data is made valid, the wait request is active. When wait request is inactive, the CPU can become ready to read data from the bus. Thus, there are no wait states in the read cycle. Also, there is a data hold time of one clock since the data must be valid for one clock after RAS and CAS become invalid and the cycle ends.

	RAS	CAS	WE	OE	RdWr/Tr	ACK	Wait	Row/Col
Read1	1	1	1	1	1	0	0	1
Read2	0	1	1	1	1	0	0	1
Read3	0	0	1	0	1	0	1	0
Read4	0	0	1	0	1	0	0	0
Read5	0	0	1	0	1	0	0	0
Read6	0	0	1	0	1	0	0	0
Read7	1	1	1	1	1	0	0	0

Figure 57: States in VRAM read cycle with signal values at each state

1.14.2 VRAM Write

The RAS and CAS cycle in a write cycle is very similar to that of a read cycle. However, output enable is permanently inactive throughout the entire cycle (also during RAS low time to ensure that a transfer cycle does not take place). This is since no data is output when writing to the VRAM. The write enable must be inactive when RAS transitions to active. To perform an early-write cycle, the write enable becomes active before CAS becomes active. The row address is clocked in when RAS becomes active and column address is clocked in when CAS becomes active. The data must be valid before the column address is clocked in, and it must stay valid for 2 clocks after the column address is clocked.

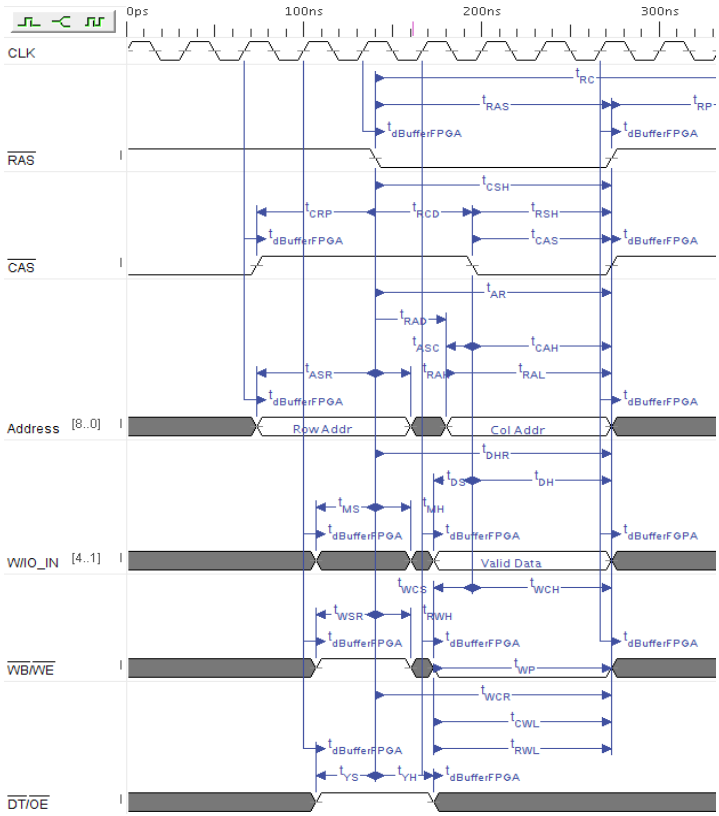


Figure 58: Timing constraints for writing a byte to DRAM portion of VRAM

The write cycle is summarized in the following table used to build the state machine. As before, the RdWr/Tr signal is always 1, since a read/write cycle is taking place. The wait signal signifies to the CPU that there is no data to wait for, and is thus pulsed inactive on the 3rd clock. When waitrequest is inactive, the CPU can place valid data on the data bus. For this reason, there are no wait states in the write cycle, as data is made valid the clock after the CPU leaves its waiting state. For the first three clocks, the row address is on the bus, so the Row/Col signal is 1. After that, when the column address is on the address bus, the Row/Col signal is 0.

	RAS	CAS	WE	OE	RdWr/Tr	ACK	Wait	Row/Col
Write1	1	1	1	1	1	0	0	1
Write2	1	1	1	1	1	0	0	1
Write3	0	1	1	1	1	0	1	1
Write4	0	1	0	1	1	0	0	0
Write5	0	0	0	1	1	0	0	0
Write6	0	0	0	1	1	0	0	0
Write7	1	1	1	1	1	0	0	0

Figure 59: States in VRAM write cycle with signal values at each state

1.14.3 Row Transfer

The transfer cycle starts when the transfer/output enable signal goes low and then RAS becomes active. Since the write enable is high during this transition of RAS, the direction of the transfer is from DRAM to SAM. Since DSF is always tied low, a normal Read transfer takes place. CAS goes low during the transfer when the column address is clocked in. The column address is the start address in the row to transfer to the SAM. This address is made to always be 0 in hardware logic, so the data is transferred starting at the first (0th) bit. Since the transfer is not synchronized with the serial clock, the transfer/output enable becomes inactive before CAS becomes active. Since there is no data from/to the CPU in this cycle, the data bus is tristated.

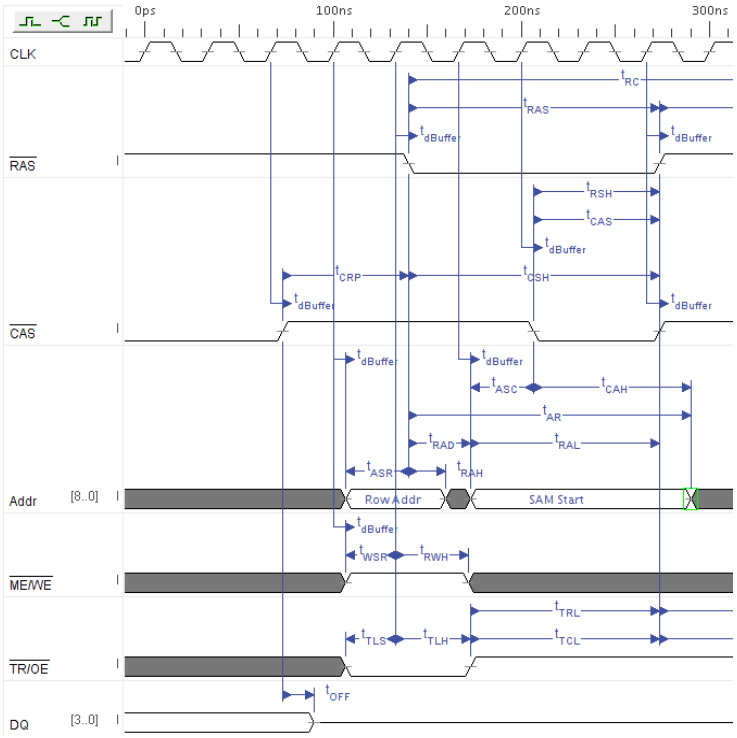


Figure 60: Timing constraints for transferring a row from DRAM to SAM

This is summarized in the following table used for the row transfer state machine. Since no data is written to the DRAM, the write enable signal is permanently high. The RdWr/Tr signal is always low to indicate that a transfer operation (and not a read or write) is taking place. Since no data must be made valid or read, the wait request signal to the CPU is permanently active. The address bus holds the row address of which row to transfer until that is clocked by CAS. After that, the address bus holds the SAM start address of which column to start from in the specified row. The acknowledge signal is used to signify to the display controller that a complete row has been transferred to the SAM. Based on the pulsing of this signal, the display controller can remove its request for a row transfer and start displaying the next row of pixels when possible.

	RAS	CAS	WE	OE	RdWr/Tr	ACK	Wait	Row/Col
Transfer1	1	1	1	0	0	0	0	1
Transfer2	1	1	1	0	0	0	0	1
Transfer3	0	1	1	0	0	0	0	1
Transfer4	0	1	1	1	0	0	0	0
Transfer5	0	0	1	1	0	0	0	0
Transfer6	0	0	1	1	0	0	0	0
Transfer7	1	1	1	1	0	1	0	0

Figure 61: States in VRAM transfer cycle with signal values at each state

1.14.4 Refresh

Every byte of the DRAM must be refreshed to ensure that the capacitors retain their charge (rather than discharging through the resistors). For this refresh, all 512 rows are accessed within 8 milliseconds. CAS before RAS refresh is used, in which the CPU does not need to provide the row addresses to refresh (done internally in VRAM). In this cycle, CAS must become active before RAS does, signifying to the VRAM that a refresh cycle must happen. If CAS becomes active while RAS is still inactive, the DRAM does not consider any of the other signals. Thus, output enable and write enable are made inactive to not drive them unnecessarily.

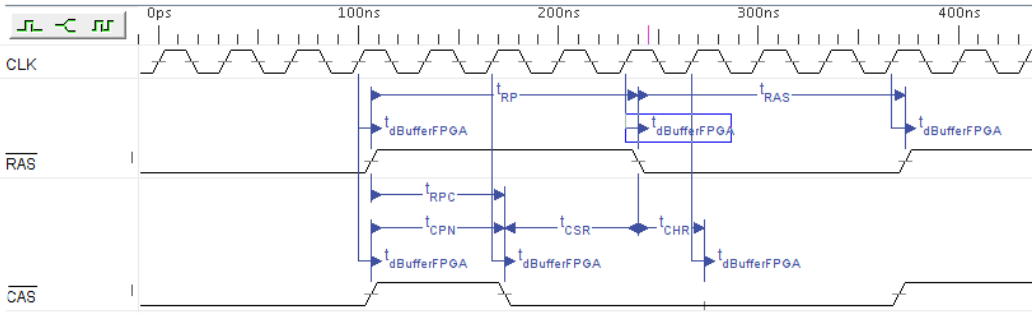


Figure 62: Timing constraints for refreshing a row of VRAM

The following table summarizes the states the refresh cycle includes in the state machine. The main two signals are only RAS and CAS. To allow for the address bus to hold all 0s without separate logic, the RdWr/Tr is 0 (signifying transfer) and Row/Col is 0 (signifying column). This allows for the SAM start address which is constantly 0 to be used in the address bus. This is done to avoid having to drive the address lines at a high voltage. Also, since this is a refresh operation, there is no acknowledge signal and no data that must become valid (waitrequest always active).

	RAS	CAS	WE	OE	RdWr/Tr	ACK	Wait	Row/Col
Refresh1	1	1	1	1	0	0	0	0
Refresh2	1	1	1	1	0	0	0	0
Refresh3	1	0	1	1	0	0	0	0
Refresh4	1	0	1	1	0	0	0	0
Refresh5	0	0	1	1	0	0	0	0
Refresh6	0	0	1	1	0	0	0	0
Refresh7	0	0	1	1	0	0	0	0
Refresh8	0	0	1	1	0	0	0	0
Refresh9	1	1	1	1	0	0	0	0

Figure 63: States in VRAM refresh cycle with signal values at each state

1.14.5 State Machine

A finite state machine is used based on the above 4 state tables for the cycles controlled by the CPU: read, write, row transfer, and refresh. Other than from the idle state, the states follow a linear progression, so that inputs do not change which state the machine progresses to. In the idle state, the machine decides which cycle to perform based on inputs. Since row transfers have highest priority (must happen within a certain time for display to work), if there is a transfer request, a row transfer cycle takes place. Otherwise, if the chip select is active, then either a read or write must be performed since the CPU is trying to access the VRAM. If write enable is active, a write cycle takes place. Otherwise, a read cycle takes place. If there is neither a chip select nor a transfer request, then a refresh cycle takes place. The outputs described in the previous state tables are encoded as state bits themselves.

This is seen in the following VHDL logic for the VRAM controller state machine.

```
--------------------------------------------------------------------------------
--
--  VRAM Controller
--
--  This is an implementation of a VRAM controller for a digital oscilloscope in
--  VHDL.
--  The VRAM controller receives control signals from the CPU and display controller
--  and using these provides the appropriate signals for the read, write, row
--  transfer, and refresh cycles of the VRAM. In addition, the controller provides
--  status output to the CPU and Display controller when a cycle is in progress
--  or if the transfer is completed, respectively.
--
--
--  Revision History:
--     18 Feb 17  Maitreyi Ashok    Initial revision.
--     21 May 17  Maitreyi Ashok    Updated with timing cycles for new VRAM
--     16 Jun 17  Maitreyi Ashok    Changed when column address valid
--     25 Jun 17  Maitreyi Ashok    Updated comments
--
--
--------------------------------------------------------------------------------


-- bring in the necessary packages
library  ieee;
use  ieee.std_logic_1164.all;


--
--  Oscilloscope VRAM Controller entity declaration
--

entity  vramController  is
    port (
        WE          :  in  std_logic;       -- Read/Write signal from CPU
        CS          :  in  std_logic;       -- Chip Select from CPU
        TransReq    :  in  std_logic;       -- Row transfer request from Display
        clk         :  in  std_logic;       -- clock
        Reset       :  in  std_logic;       -- reset the system
        RAS         :  out std_logic;       -- Row address strobe signal
        CAS         :  out std_logic;       -- Column address strobe signal
        WB_WE       :  out std_logic;       -- Write enable signal
        DT_OE       :  out std_logic;       -- Data transfer, output enable signal
        RdWr_Tr     :  out std_logic;       -- Serial output enable
        ACK         :  out std_logic;       -- Acknowledgement to Display controller
                                            -- that transfer completed
        WaitCPU     :  out std_logic;       -- Signal to CPU to wait
        ROW_COL     :  out std_logic        -- Row or column address in address bus
    );
end  vramController;


--
--  Oscilloscope VRAM Controller Moore State Machine
--     State Assignment Architecture
--
--  State assignments are made manually.  This is useful for minimizing output
--  decoding logic and avoiding glitches in the output (due to the decoding
--  logic). State bits are assigned as output bits, with 4 additional state bits
--  to distinguish between states with identical outputs
--

architecture  assign_statebits  of  vramController  is
```

```
    subtype  states  is  std_logic_vector(10 downto 0);     -- state type

    -- define the actual states as constants
    constant  IDLE      : states := "11110000100";  -- waiting to perform some VRAM action

     constant  REFRESH1  : states := "11110000110";  -- first step of refresh cycle
     constant  REFRESH2  : states := "11110000011";  -- second step of refresh cycle
     constant  REFRESH3  : states := "10110000001";  -- third step of refresh cycle
     constant  REFRESH4  : states := "10110000000"; -- fourth step of refresh cycle
     constant  REFRESH5  : states := "00110000011"; -- fifth step of refresh cycle
     constant  REFRESH6  : states := "00110000110"; -- sixth step of refresh cycle
     constant  REFRESH7  : states := "00110000101"; -- seventh step of refresh cycle
     constant  REFRESH8  : states := "00110000100"; -- eighth step of refresh cycle
     constant  REFRESH9  : states := "11110000010"; -- ninth step of refresh cycle

     constant  READ1        : states := "11111001010";  -- first step of read cycle
     constant  READ2        : states := "01111001000";  -- second step of read cycle
     constant  READ3        : states := "00101010000";  -- third step of read cycle
     constant  READ4        : states := "00101000010";  -- fourth step of read cycle
     constant  READ5        : states := "00101000001";  -- fifth step of read cycle
     constant  READ6        : states := "00101000011";  -- sixth step of read cycle
     constant  READ7        : states := "11111000001";  -- seventh step of read cycle

     constant  WRITE1    : states := "11111001001";  -- first step of write cycle
     constant  WRITE2    : states := "11111001000";  -- second step of write cycle
     constant  WRITE3    : states := "01111011001";  -- third step of write cycle
     constant  WRITE4    : states := "01011000000";  -- fourth step of write cycle
     constant  WRITE5    : states := "00011000001";  -- fifth step of write cycle
     constant  WRITE6    : states := "00011000000";  -- sixth step of write cycle
     constant  WRITE7    : states := "11111000000";  -- eighth step of write cycle

     constant  TRANSFER1     : states := "11100001001";  -- first step of row transfer cycle
     constant  TRANSFER2     : states := "11100001000";  -- second step of row transfer cycle
     constant  TRANSFER3     : states := "01100001000";  -- third step of row transfer cycle
     constant  TRANSFER4     : states := "01110000010";  -- fourth step of row transfer cycle
     constant  TRANSFER5     : states := "00110000001";  -- fifth step of row transfer cycle
     constant  TRANSFER6     : states := "00110000000";  -- sixth step of row transfer cycle
     constant  TRANSFER7     : states := "11110100000";  -- seventh step of row transfer cycle

     signal  CurrentState  :  states;    -- current state
     signal  NextState     :  states;    -- next state

begin


    -- Define output bits (which are just state bits)
     RAS        <= CurrentState(10);    -- RAS is the high (10th) bit
     CAS        <= CurrentState(9);     -- CAS is the second highest (9th) bit
     WB_WE   <= CurrentState(8);        -- WB/WE is the third highest (8th) bit
     DT_OE   <= CurrentState(7);        -- DT/OE is the fourth highest (7th) bit
     RdWr_Tr     <= CurrentState(6);    -- RD/Wr vs transfer is the fifth highest(6th) bit
     ACK        <= CurrentState(5);     -- ACK is the sixth highest (5th) bit
     WAITCPU <= CurrentState(4);        -- WAITCPU is the seventh highest (4th) bit
     ROW_COL <= CurrentState(3);        -- Row/Col is the eighth highest (3rd) bit


    -- compute the next state (function of current state and inputs)

    transition:  process (Reset, TransReq, CS, WE, CurrentState)
    begin

        case  CurrentState  is          -- do the state transition/output

            when  IDLE =>               -- in idle state, do transition
```

```
            if  (TransReq = '1')  then
                NextState <= TRANSFER1;     -- transfer request from display controller
            elsif  (CS = '0' and WE = '1')  then
                NextState <= READ1;         -- chip select from CPU and read request
            elsif (CS = '0' and WE = '0') then
                NextState <= WRITE1;        -- chip select from CPU and write request
                 else
                      NextState <= REFRESH1;        -- if nothing else, refresh
            end if;

        when  REFRESH1 =>                   -- Move from each refresh state to next
            NextState <= REFRESH2;          -- consecutive refresh state every clock
            when  REFRESH2 =>
            NextState <= REFRESH3;
            when  REFRESH3 =>
            NextState <= REFRESH4;
            when  REFRESH4 =>
            NextState <= REFRESH5;
            when  REFRESH5 =>
            NextState <= REFRESH6;
            when  REFRESH6 =>
            NextState <= REFRESH7;
            when  REFRESH7 =>
            NextState <= REFRESH8;
            when  REFRESH8 =>
            NextState <= REFRESH9;
            when  REFRESH9 =>
            NextState <= IDLE;

            when  READ1 =>                  -- Move from each read state to next
                 NextState <= READ2;        -- consecutive read state every clock
            when  READ2 =>
                 NextState <= READ3;
            when  READ3 =>
                 NextState <= READ4;
            when  READ4 =>
                 NextState <= READ5;
            when  READ5 =>
                 NextState <= READ6;
            when  READ6 =>
                 NextState <= READ7;
            when  READ7 =>
                 NextState <= IDLE;

            when  WRITE1 =>                 -- Move from each write state to next
                 NextState <= WRITE2;       -- consecutive write state every clock
            when  WRITE2 =>
                 NextState <= WRITE3;
            when  WRITE3    =>
                 NextState <= WRITE4;
            when  WRITE4 =>
                 NextState <= WRITE5;
            when  WRITE5 =>
                 NextState <= WRITE6;
            when  WRITE6 =>
                 NextState <= WRITE7;
            when  WRITE7 =>
                 NextState <= IDLE;

            when  TRANSFER1 =>              -- Move from each row transfer state to next
                 NextState <= TRANSFER2;    -- consecutive row transfer state every clock
            when  TRANSFER2 =>
                 NextState <= TRANSFER3;
            when  TRANSFER3 =>
```

```
                NextState <= TRANSFER4;
            when  TRANSFER4 =>
                NextState <= TRANSFER5;
            when  TRANSFER5 =>
                NextState <= TRANSFER6;
            when  TRANSFER6 =>
                NextState <= TRANSFER7;
            when  TRANSFER7 =>
                NextState <= IDLE;

            when  OTHERS =>             -- in any other state
            NextState <= IDLE;          -- always go back to idle

        end case;




    end process transition;


    -- storage of current state (loads the next state on the clock)

    process (clk)
    begin

        if  Reset = '0'  then           -- reset overrides everything
            CurrentState <= IDLE;       -- go to idle on reset
        elsif  clk = '1'  and rising_edge(clk) then -- only change on rising edge of clock
            CurrentState <= NextState;              -- save the new state information
        end if;

    end process;


end  assign_statebits;
```

1.14.6 Address Multiplexing

The 9 bit row and column addresses are multiplexed on the 9 bit address bus. This is done in the following manner in conjunction with the VRAM controller. If a read/write is taking place (RdWr_Tr = 1), then the complete address output by the CPU is split into two parts. The upper 9 bits of the address refer to the row address (output on bus when Row_Col = 1). The lower 9 bits of the address refer to the column address (output on bus when Row_Col = 0). If a row transfer operation is taking place (RdWr_Tr = 0), the address of the row to transfer or the SAM start address is output. If the row address must be output (Row_Col = 1), then a counter is used to determine which row. Every transfer request, this counter is incremented and it cycles through rows 0 to 271. This is because the display only contains 272 rows, and so only these rows need to be output to the display serially. If the SAM start address must be output (Row_Col = 0), then the address 0 is always output. This ensures that the row is transferred starting at bit 0 of the SAM.

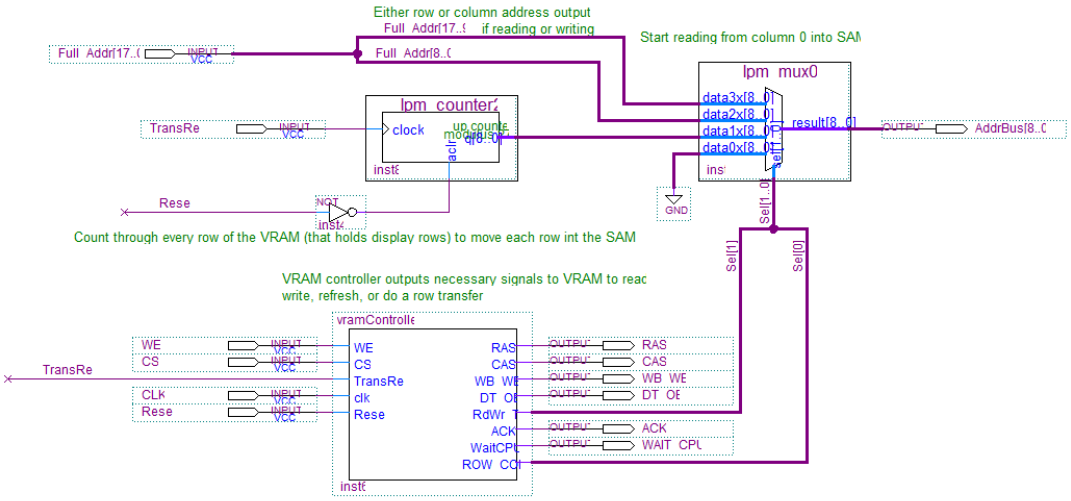


Figure 64: Selection of address to send to VRAM based on cycle to perform

1.14.7 Serial Output

The serial output is not controlled by the VRAM controller, but the timing cycle for it is explained here. The timing does not involve any of the control signals except for the serial clock and serial enable. The serial enable is tied permanently active in hardware logic. Thus, for every rising edge of serial clock, a nibble of data is output serially from each VRAM (where it is read by the display). Due to the constraints of the display, a 10 MHz clock is used for the serial clock. In regions of the display cycle when no data should be ouput from the VRAM there will be no rising edges in the serial clock, and the serial data output will be tristated.

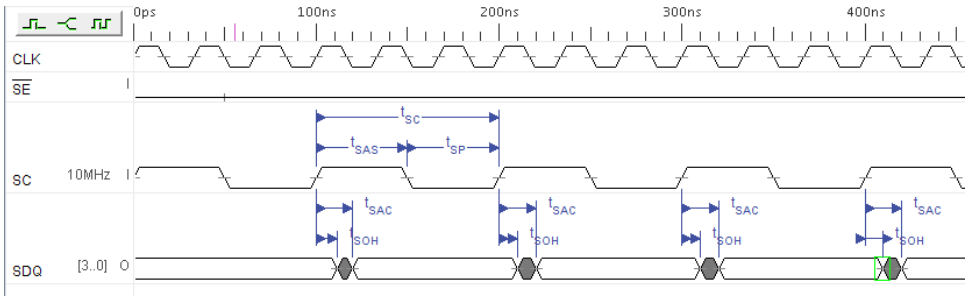


Figure 65: Timing constraints for serial output form SAM of VRAM

1.15 Display

A 480 x 272 pixel graphics thin film transistor (TFT) LCD display with backlight, ER-TFT043-3, is used to display the waveforms measured by the oscilloscope and a menu of options to select. This display was chosen for its RGB interface. In addition, the medium size allows for easy viewing of signals but also a reasonable size to have a portable oscilloscope.

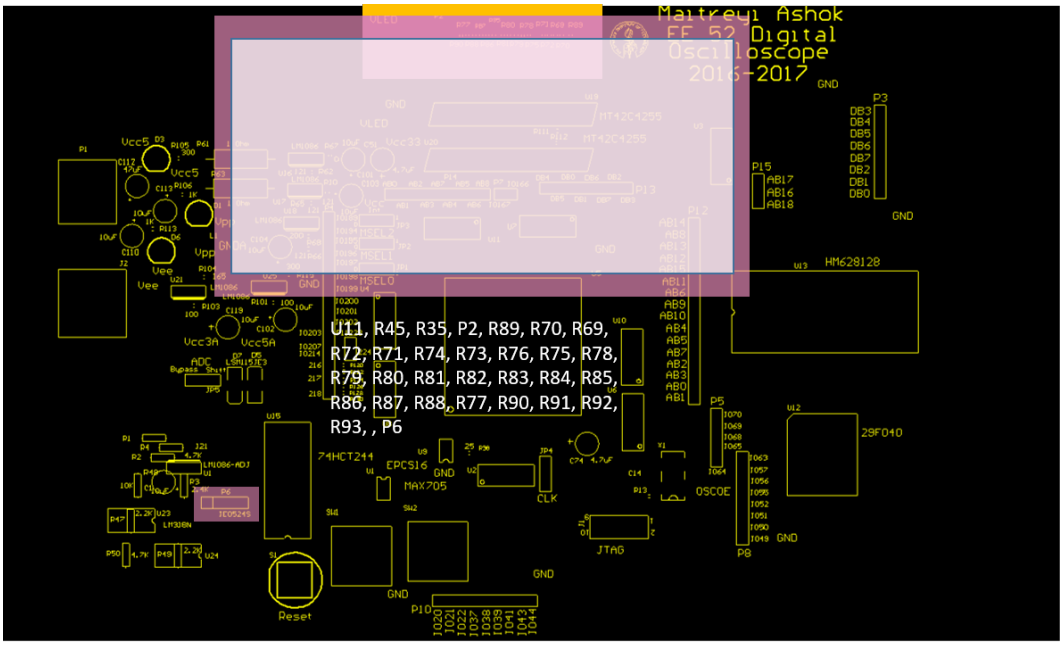


Figure 66: Layout of display component on PCB with designators of associated parts

The backlight of the LCD is powered by a 24 V DC voltage. This is generated by a DC to DC converter from 5 V to 24 V (P6), which takes in an input of 5 V referenced to ground, and outputs 24 V, also referenced to ground.

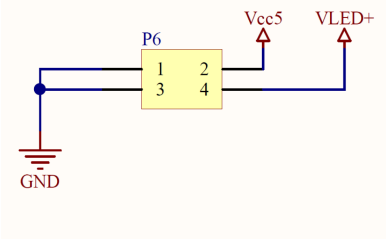


Figure 67: Schematic of DC converter for LCD Backlight voltage

The display is connected to the PCB through a 40 pin zero insertion force (ZIF) connector (P2), which allows for easy removal of the display from the oscilloscope. All the color bits input to the LCD go through a 10 KOhm resistor to limit current entering the display (R89, R70, R69, R72, R71, R74, R73, R76, R75, R78, R79, R80, R81, R82, R83, R84, R85, R86, R87, R88, R77, R90, R91, R92). Only 8 of the bits are changeable, and the other bits of color are pulled high to be 1. Thus, a true white color can be displayed (0xFFFFFF) for the background color of the oscilloscope if all the controllable color bits are pulled high as well. However, a true black color can never be displayed, and the darkest color that can be output is a dark blue. The controllable color bits are the high bits of each color, to allow for a difference in a color bit to cause a large difference in the shade of the color. Since the display is on whenever the oscilloscope is turned on, pin 31 (DISP) is pulled high at all times. The resistive touch panel is not used, and thus the last 4 pins are unconnected. HSync, VSync, DOTCLK and Disp_Enable are control signals used to synchronize the timing

of displaying pixels and rows. HSync is the horizontal sync that synchronizes the start of a horizontal row to display. VSync is the vertical sync that synchronizes the start of displaying the first row at the top of the display. DOTCLK is a pixel clock that clocks the color data for each pixel. Disp_Enable enables when data to display is read from the VRAM serial output. If Disp_Enable (active high) is active, then the serial output data is read at every pixel clock. Otherwise, data from the VRAM is not read into the display. The succession of these signals will be discussed in greater depth in the next section.

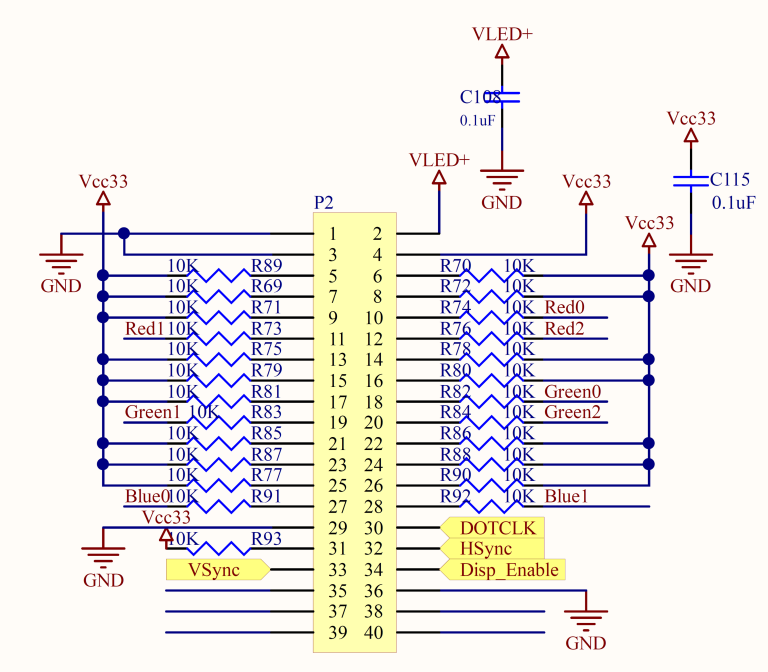


Figure 68: Signals input to display through connector for color and control signals

The control signals for the display are output from an output only bank in a buffer (U11, R45). This is since no status is sent back from the display to the CPU, so input is not needed.

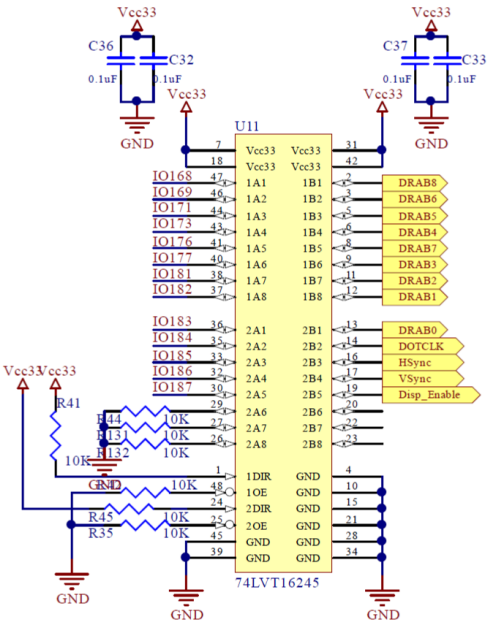


Figure 69: Schematic of buffer used to transmit control signals from FPGA to display

The display interacts with the Display Controller, VRAM, and VRAM Controller with the movement of signals as shown in the following block diagram.

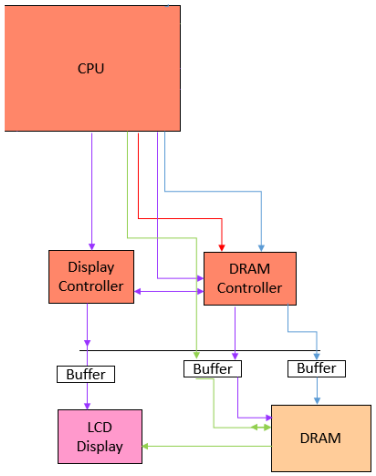


Figure 70: Interaction of signals between CPU, display components, and VRAM components

1.16 Display Controller

The pixel clock used is a 10 MHz clock. This is because the display expects a clock of frequency about 9 MHz and 10 MHz is a convenient divisor of the 30 MHz system clock.

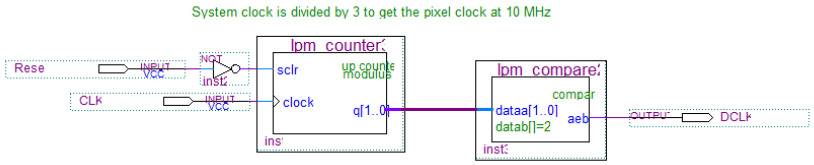


Figure 71: Creation of slower pixel clock for LCD display from system clock

The control of the display can be split into two main parts: the cycling through the pixels in a row and the cycling through all the rows in the display.

Cycling through all the pixels in a row happens in a HSync cycle as follows. Every row cycle starts with HSync pulsing low for 41 pixel clocks. For the rest of the cycle, HSync is high. On either side of the data being valid (and 480 pixels being clocked in), there is a porch where HSync is high but data is invalid. The back porch before data becomes valid is 47 pixel clocks, and the front porch after data is all clocked in is 8 pixel clocks.

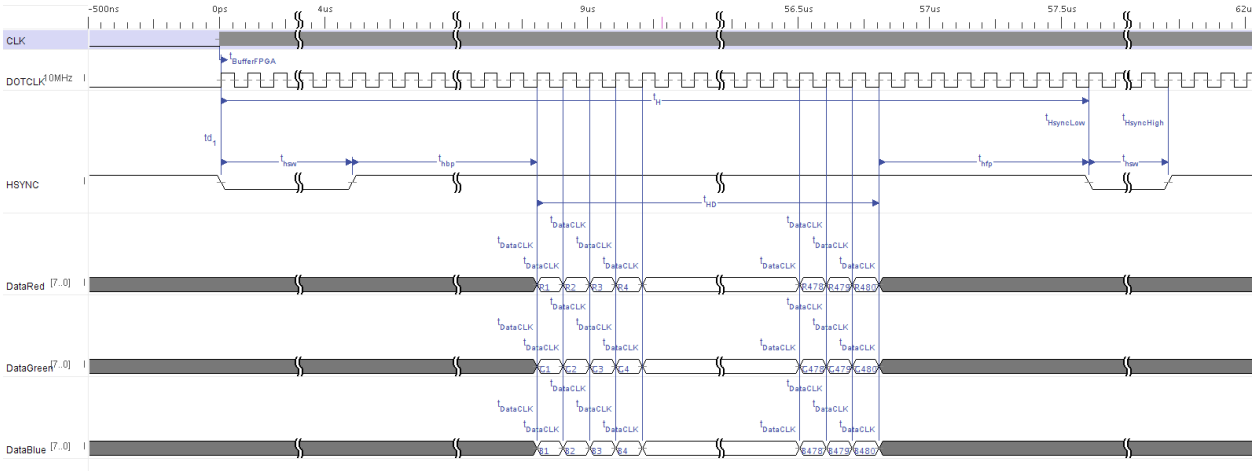


Figure 72: Timing constraints for one cycle of HSync

The serial clock only has rising edges in the region between the back and front porches, when data is valid. The display data enable signal is also only active in this region, though it becomes active one clock after the serial clock starts pulsing. This is to ensure that there is valid data clocked from the VRAM to the display by the time the display tries to read it.

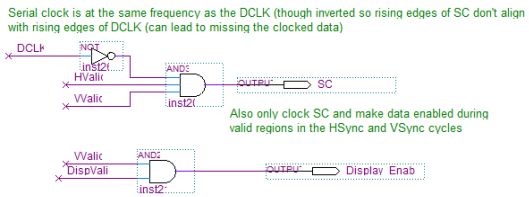


Figure 73: Creation of serial clock and display enable signals based on timing constraints

At the end of each HSync cycle, a transfer request is made to the VRAM for the next row to be moved from DRAM to SAM. This row transfer will take place within the next HSync low time, when no data is read from the SAM. The transfer request is latched by a set-reset flip flop and is only reset when the VRAM controller acknowledges that the transfer request has taken place. Transfer requests also only take place during the valid vertical sync regions of the display, once for each row in the display.

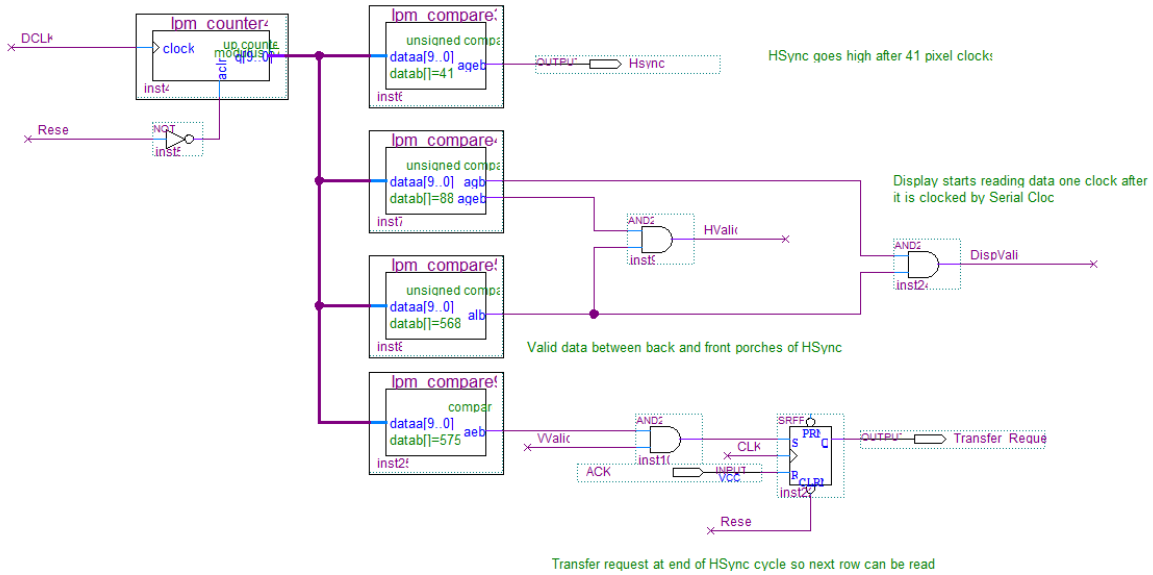


Figure 74: Generation of HSync and Transfer Request signals

Cycling through all the rows in the display happens in a VSync cycle as folllows (defined in terms of HSync cycles). Every cycle through all the rows starts with VSync being low for 10 HSync cycles. Again, there is a back porch before data becomes valid for 2 HSync cycles, and a front porch after data has become invalid again for 3 HSync cycles. In between the porches, there is valid data (contingent on the HSync cycle being valid) as the HSync cycles go through all 272 rows of the display.

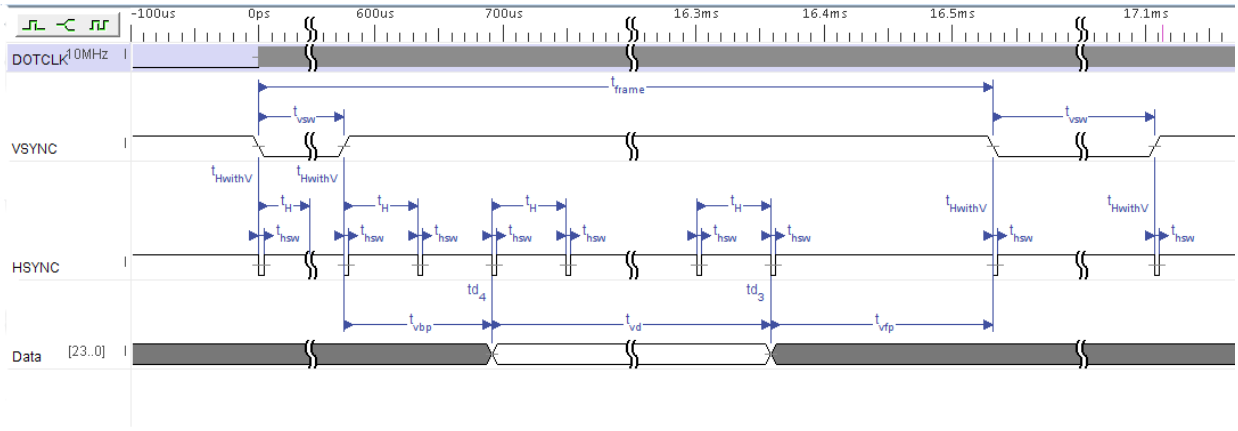


Figure 75: Timing constraints for one cycle of VSync

To avoid glitches propagating, the clock for the VSync cycle logic is based on the DCLK and not HSync. Thus, the timing diagram is defined as follows in hardware logic, where each HSync cycle is 576 DCLKs.

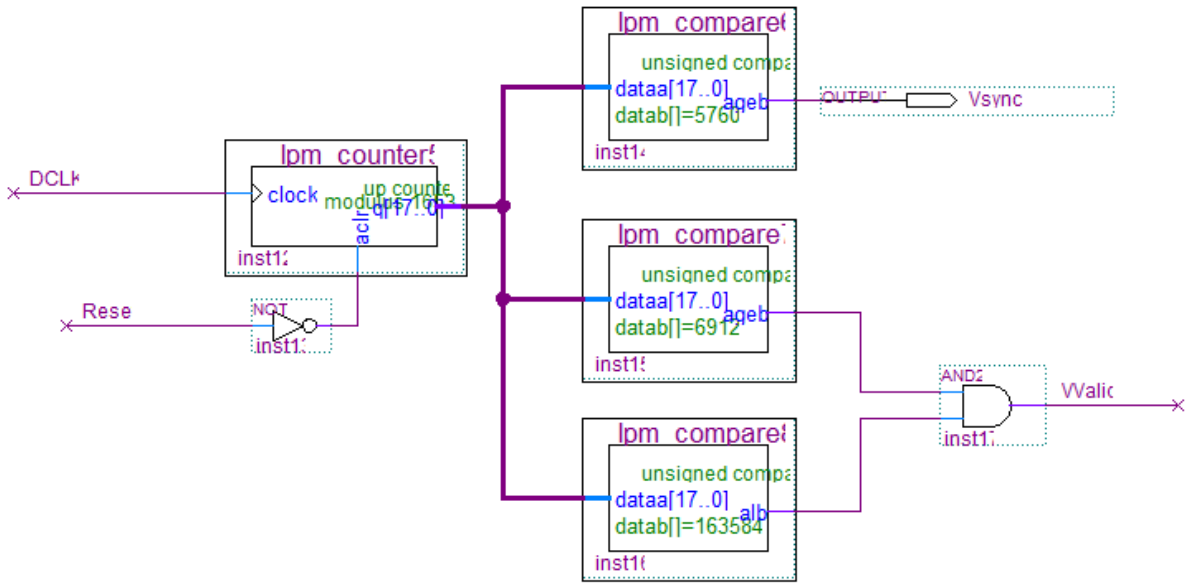


Figure 76: Generation of VSync signal

This display controller interfaces with the CPU and VRAM controller in the following configuration.

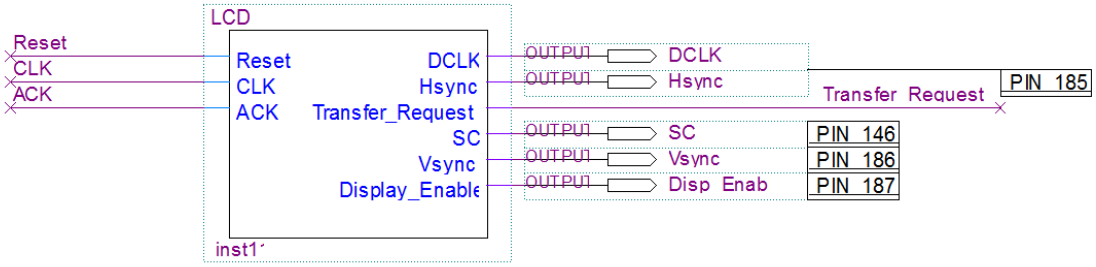


Figure 77: Input and output signals to display controller

1.17 Analog Input

The analog section of the oscilloscope consists of many components, as seen in the layout on the PCB below. While all parts of the analog section were designed and built, they were not all tested. Thus, the analog section is contingent on nonidealities present in a real oscilloscope system.

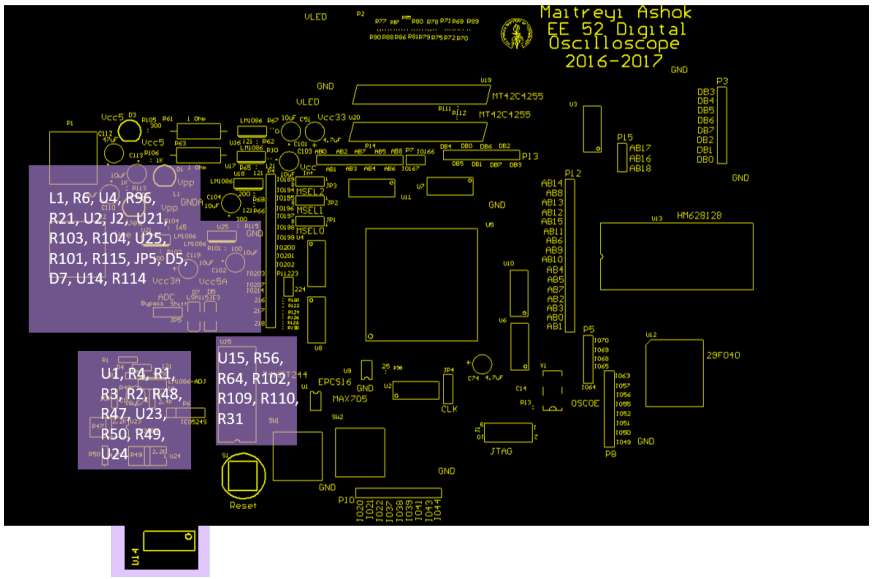


Figure 78: Layout of components for analog section and designators for associated parts

1.17.1 Analog Power

The power for the analog section is supplied separately to reduce the amount of noise introduced into the analog components. This will allow for more accurate reading of the analog data. The ground that all the analog components reference is connected to the digital ground by a ferrite bead (L1), which kills high frequency noise. The ferrite bead was chosen since its impedance at DC is 200 mOhm, while it is 600 Ohm at 100 MHz, causing high frequency signals to be filtered out effectively.

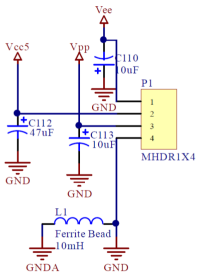


Figure 79: +/-12 V used from power line for analog, and analog ground is connected to digital ground

Analog voltages are also created by regulating the +12 V down to the necessary voltages using a LM1086-ADJ regulator for the same reasons as in the main power section (see figure there for generic schematic for adjustible regulator). This is done rather than using the existing voltages which are regulated down from 5 V to avoid interference with the rest of the system (which uses the digital voltages). For the Analog to Digital converter (ADC), an analog 5 V supply is needed, as well as a 3.3 V analog voltage for the top reference of values.

For a 5 V regulator (U25), R_1 (R101) is a 100 Ohm resistor and R_2 (R115) is a 300 Ohm resistor to allow for

$$V_{out} = V_{ref}(1 + 3) = 5V$$

For a 3.3 V regulator (U21), R_1 (R103) is a 100 Ohm resistor and R_2 (R104) is a 165 Ohm resistor to allow for

$$V_{out} = V_{ref}(1 + 1.65) = 3.3V$$

For another regulator to be used as a reference for the operational amplifiers in the front end, R_1 (R4) is a 121 Ohm resistor and R_2 (R1) is a 1 KOhm potentiometer to allow for a range of

$$V_{out} = V_{ref}(1) \; to \; V_{ref}(1 + 8.26) = 1.25V \; to \; 11.58V$$

. This adjustible reference will allow for adjusting the range of input signal voltages that can be scaled down to the range allowed by the ADC. Since this entire range is not necessary for the scaling down, but reference voltages below 1.25 V may be necessary, the voltage is reduced by a voltage drop across a resistor (R2).

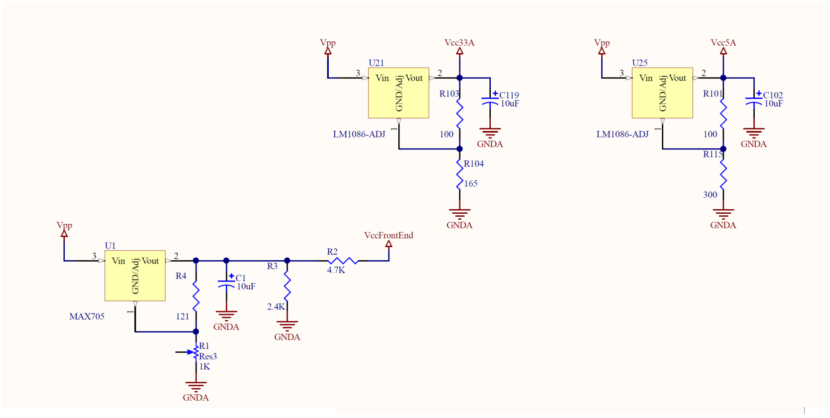


Figure 80: Regulation of separate voltage lines for analog components

1.17.2 Front End

The following schematic is used to scale signals in the range -12 V to +12 V down to a voltage between 0 V and 3.3 V.

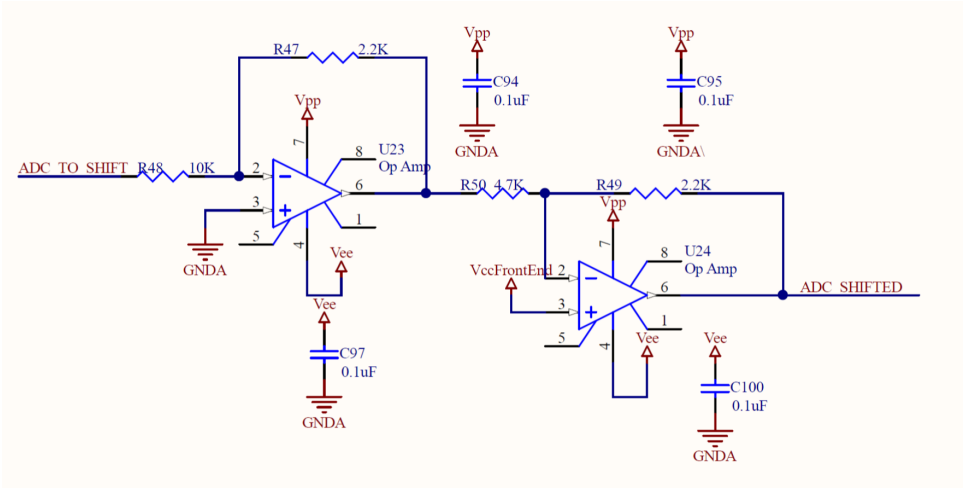


Figure 81: Level shifter using op amps to scale and shift input voltage

Assuming the op amps are ideal: U23 is connected in the inverting op amp configuration. Thus, the gain of U23 is -R47/R48 = -0.22. U24 is also in the inverting configuration, so its gain is -R49/R50 = -0.47. Since the input to the positive terminal of U24 is a voltage other than ground, the output of U24 will be shifted up by VccFrontEnd. If a voltage between -12 V and 12 V is input to R48 (ADC_TO_SHIFT), then the output at pin 6 of U23 will range between +2.64 V and -2.64 V. The voltage at the output of pin 6 of U24 (ADC_SHIFTED) will range between VccFrontEnd -1.25 V and VccFrontEnd + 1.25 V. If VccFrontEnd is adjusted to be around 1.8 V, then the output of the front end will range between about 0.5 V and 3.1 V, so that both ends of the range are well within the top and bottom reference voltage ranges in the ADC. Even if voltages slightly outside of the range of -12 V to 12 V are input, the ADC will not be broken by a negative voltage or too large voltage.

The operational amplifier used was the LM318N. This op amp was used mainly due to its wide bandwidth of 15 MHz. This allows for the highest sampling frequency supported by the oscilloscope (when sampling at 10 MHz to measure 5 MHz signals) to be passed through the op amps without the gain being reduced. In addition, the input offset and bias current are low with a maximum of 200 nA and 500 nA, respectively. Due to this, the op amps can be treated effectively as ideal op amps, allowing the circuit design to be used with minimal changes.

This front end was tested separately from the rest of the PCB in a prototyping area. When tested, a voltage between -12 V and +12 V did output a voltage in the range of 0 V to 3.3 V. The actual front end built was slightly different from the design above (VccFrontEnd was different), but this version was lost in the debugging of a different issue that led to all unnecessary components being removed from the board. This front end was never connected with the ADC and the rest of the circuit to scale down signals for the oscilloscope.

1.17.3 Main Analog Conversion

Signals are input to the oscilloscope through a BNC connector (J2) on the PCB. The signals can be sourced from a probe or function generator.

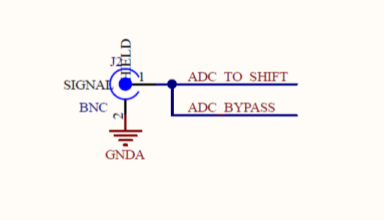


Figure 82: Signal input through BNC connector

The signals can be input straight from this connector to the ADC (U14), if the signals input are within the range of 0 V to 3.3 V (the maximum range the ADC will accept). If a larger range is desired, signals scaled down from the front end can be selected. This done by a 3 pin jumper (JP5) so the user can switch between the two modes. In addition, to ensure that the input voltage to the ADC stays within the maximum range, two Schottky diodes (D5, D7) which have 220 mV of forward voltage drop limit the range. If the input voltage to the ADC is too large, D5 will become an almost short, causing the input voltage to rail at 3.3 V. If the input voltage is too small, D7 will become an almost short, causing the input voltage to rail at 0 V. The diode used was the Microsemi LSM115JE3. This was used due to the low forward voltage at 1 A of current as well as the low reverse leakage current. Thus, when the voltage is within the allowed range, less current is pulled by the diodes, and most of it enters the ADC. In addition, the actual range allowed by the diodes (including 200 mV extra on each side due to the forward voltage drop) is very close to the intended range of 0 V to 3.3 V.

The analog to digital converter used is the TLC5510A. This is due to it having a 4 V full scale, as well as a high maximum conversion rate of 20 megasamples per second. In addition, the integral and differential linearity errors are low at +/- 0.75 LSB and +/- 0.5 LSB maximum, respectively, at room temperature.

The ADC will be clocked at a regular clock (ADCCLK) which depends on the sampling frequency set by the user. Every clock, the analog voltage will be converted to 8 bits representing the voltage in digital form.

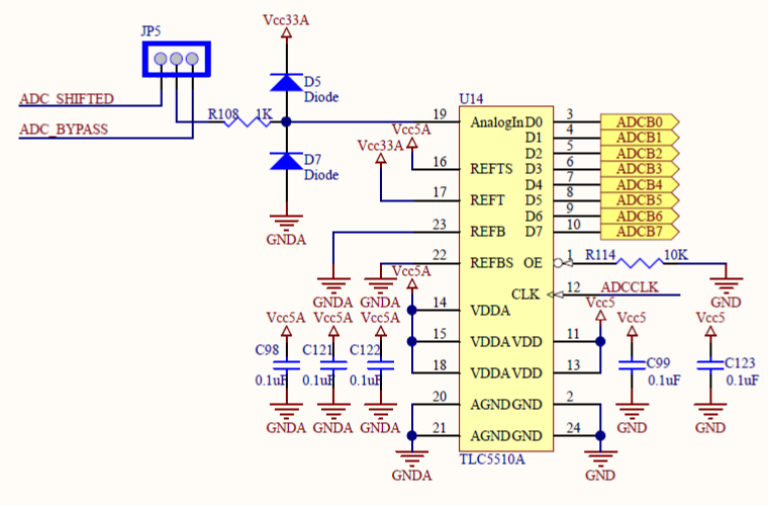


Figure 83: Connection of signals and components to ADC in schematic

The top reference for the sampling is the 3.3 V analog voltage generated, while the bottom reference is the analog ground. The output of the ADC is alays enabled, with pin 1 pulled active (R114). This is since the oscilloscope will always be measuring signals whenever powered and trigger is enabled. The signals will be displayed depending on whether there is a trigger or not. These 8 bits of data are wired to the CPU to be displayed on the LCD as well as used to generate a trigger through two separate bidirectional buffer banks (U2, U4). For both of these buffers, the corresponding direction pin controlled by the FPGA is pulled low so the banks are configured for input (IO45, IO219).

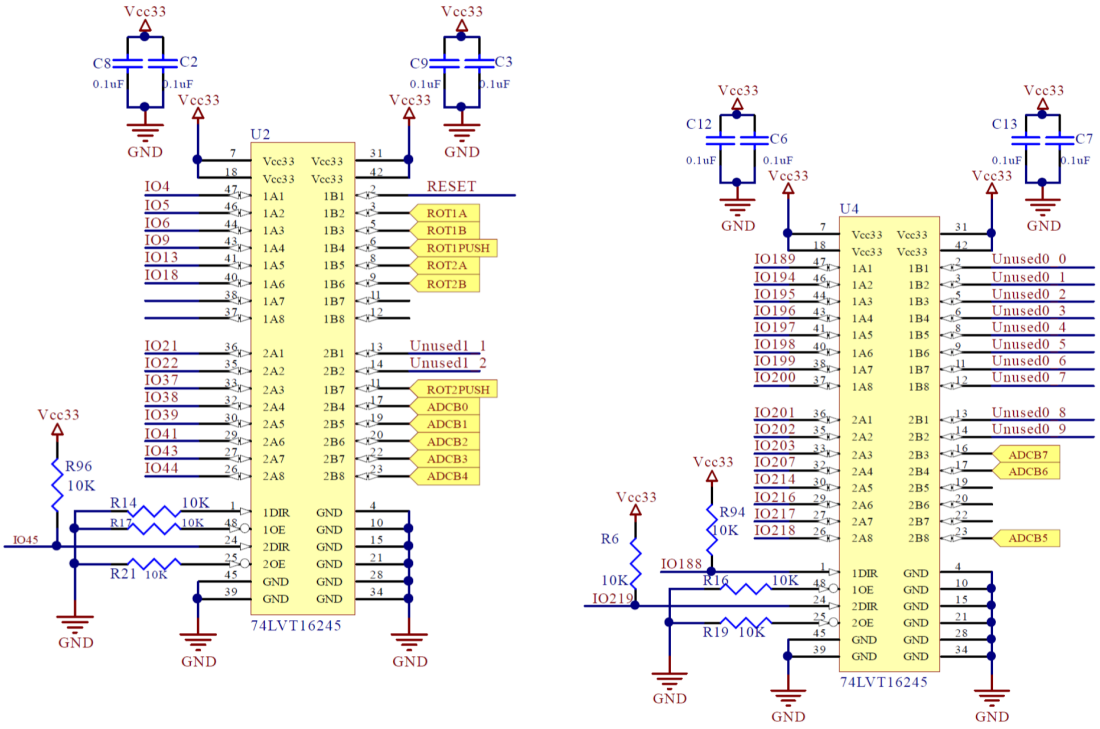


Figure 84: Buffers for input of converted analog signals to FPGA from ADC

Since the input voltage levels of the ADC are not compatible with the buffers used elsewhere in the system, a TI SN74HCT244 octal buffer (U15) was used. They are not compatible since the 74LVT162245 has a V_{OH} of V_{cc} - 0.2 V, which is 3.1 V. However, the V_{IH} of the ADC is 4 V. Thus, voltages output by the buffer which are considered a '1' might not be considered as a '1' by the ADC. Thus, the different line driver was used, for its V_{OH} of 4.4 V as well as the high current output to drive the load. The ADCCLK signal is passed through one of the inputs in this buffer. These buffers only allow for output, so the output is enabled for the upper bank by pin 1 (pulled low by R110). Since the lower bank is unused, all the input pins are tied low but are not actually output anywhere.

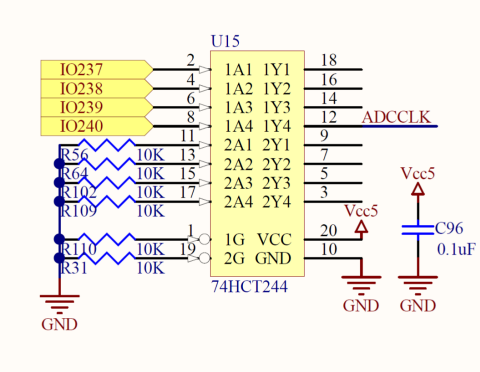


Figure 85: Separate buffer to send control signal to ADC

The physical components of the analog section interface with the trigger controller and FIFO in the following way.

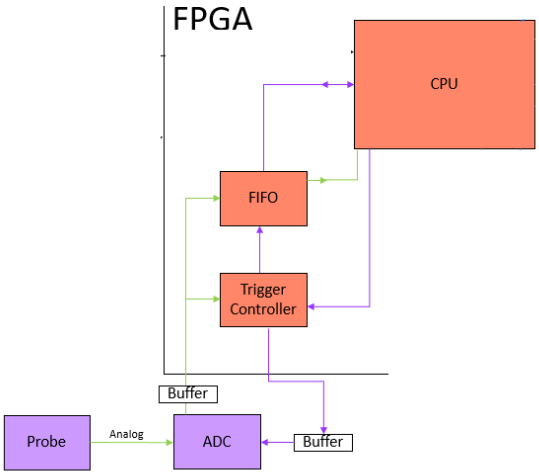


Figure 86: Interaction of signals and blocks in analog section

1.18 Trigger Controller

The trigger controller is responsible for deciding when a trigger should be generated and then creating the trigger event. This is done in multiple stages.

First, a sampling clock is generated. The sampling frequency is passed in from software as a 25 bit value of the amount of time between samples in units of 100s of nanoseconds. Since the system clock period is 33.33 ns, multiplying the sampling frequency by 3 gives us the time between samples in terms of the system clock. When compared to the value from a counter based on the system clock, a sample clock rising edge can be generated every time the counter equals the number of clocks to wait for. When there is a rising edge in the sampling clock, the counter is reset to start waiting again for the next rising edge. This counter is active whenever the trigger is enabled, and the system is not being reset, as seen below. This clock is also output from the trigger controller to the ADC as the clock to use when sampling analog data.

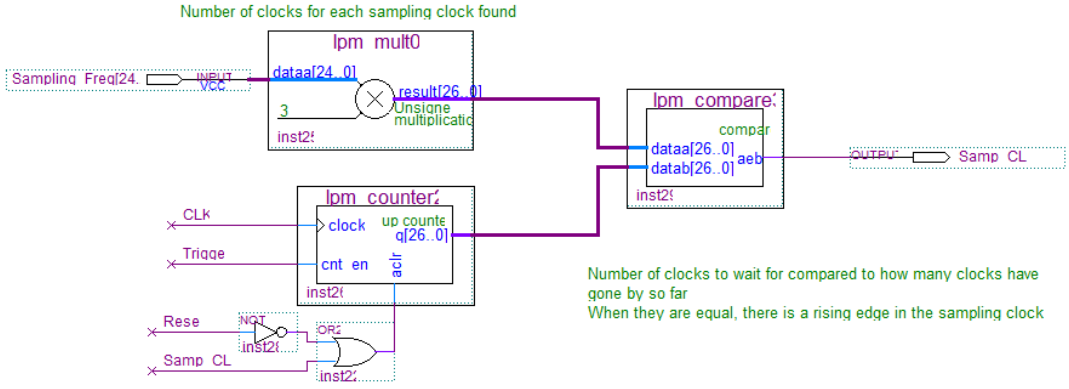


Figure 87: Generation of sample clock by dividing system clock based on sampling frequency

Generating a trigger event is first done by checking whether a manual trigger should be generated depending on the parameters generated by the user. This includes the trigger slope, level, and trigger enable signals, as well as the data itself. The trigger slope is a 1 bit value where 0 is a negative slope, and 1 is a positive slope. The trigger enable is also a flag, where true is trigger events being enabled while false is trigger events being disabled. The trigger level is a 7 bit value, for reasons explained below. This is mostly done by a state machine implemented in VHDL on the next page. In essence, if the trigger slope is positive and the analog signal transistions from below the trigger level to above it, then a trigger is generated. The

opposite is true for a negative trigger slope: if the signal transitions from above the trigger level to below it, a trigger event is generated. The only output of the state machine is a trigger event.

As an input to this state machine, the relation of the signal to the trigger level must be calculated (to know if it is less, equal, or greater). This is done by a comparator. To avoid the trigger being affected by glitches in the signal, only the top 7 bits of the analog signal are used to generate a trigger. This avoids small changes in the LSB creating triggers repeatedly. The trigger is also reset whenever the system is reset or the trigger is disabled. This logic is pictured below.

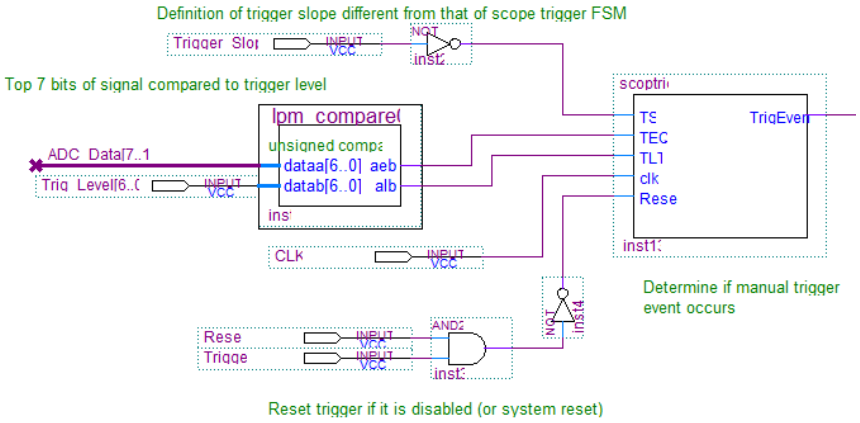


Figure 88: How to generate a manual trigger based on parameters set by user

```
--------------------------------------------------------------------------------
--
--  Oscilloscope Digital Trigger
--
--  This is an implementation of a trigger for a digital oscilloscope in
--  VHDL.  There are three inputs to the system, one selects the trigger
--  slope and the other two determine the relationship between the trigger
--  level and the signal level.  The only output is a trigger signal which
--  indicates a trigger event has occurred.
--
--  The file contains multiple architectures for a Moore state machine
--  implementation to demonstrate the different ways of building a state
--  machine.
--
--
--  Revision History:
--     13 Apr 04  Glen George       Initial revision.
--      4 Nov 05  Glen George       Updated comments.
--     17 Nov 07  Glen George       Updated comments.
--     13 Feb 10  Glen George       Added more example architectures.
--       11 Feb 17  Maitreyi Ashok      Fixed compilation errors
--
--------------------------------------------------------------------------------


-- bring in the necessary packages
library  ieee;
use  ieee.std_logic_1164.all;


--
--  Oscilloscope Digital Trigger entity declaration
--

entity  scoptrig  is
    port (
        TS         :  in  std_logic;      -- trigger slope (1 -> negative, 0 -> positive)
        TEQ        :  in  std_logic;      -- signal and trigger levels equal
        TLT        :  in  std_logic;      -- signal level < trigger level
        clk        :  in  std_logic;      -- clock
        Reset      :  in  std_logic;      -- reset the system
        TrigEvent  :  out  std_logic      -- a trigger event has occurred
    );
end  scoptrig;


--
--  Oscilloscope Digital Trigger Moore State Machine
--     State Assignment Architecture
--
--  This architecture just shows the basic state machine syntax when the state
--  assignments are made manually.  This is useful for minimizing output
--  decoding logic and avoiding glitches in the output (due to the decoding
--  logic).
--

architecture  assign_statebits  of  scoptrig  is

    subtype  states  is  std_logic_vector(2 downto 0);     -- state type

    -- define the actual states as constants
    constant  IDLE      : states := "000";  -- waiting for start of trigger event
    constant  WAIT_POS  : states := "001";  -- waiting for positive slope trigger
    constant  WAIT_NEG  : states := "010";  -- waiting for negative slope trigger
```

```
    constant  TRIGGER   : states := "100";  -- got a trigger event


    signal  CurrentState  :  states;    -- current state
    signal  NextState     :  states;    -- next state

begin


    -- the output is always the high bit of the state encoding
    TrigEvent <= CurrentState(2);


    -- compute the next state (function of current state and inputs)

    transition:  process (Reset, TS, TEQ, TLT, CurrentState)
    begin

        case  CurrentState  is          -- do the state transition/output

            when  IDLE =>               -- in idle state, do transition
                if  (TS = '0' and TLT = '1' and TEQ = '0')  then
                    NextState <= WAIT_POS;      -- below trigger and + slope
                elsif  (TS = '1' and TLT = '0' and TEQ = '0')  then
                    NextState <= WAIT_NEG;      -- above trigger and - slope
                else
                    NextState <= IDLE;          -- trigger not possible yet
                end if;

            when  WAIT_POS =>           -- waiting for positive slope trigger
                if  (TS = '0' and TLT = '1')  then
                    NextState <= WAIT_POS;      -- no trigger yet
                elsif  (TS = '0' and TLT = '0')  then
                    NextState <= TRIGGER;       -- got a trigger
                else
                    NextState <= IDLE;          -- trigger slope changed
                end if;

            when  WAIT_NEG =>           -- waiting for negative slope trigger
                if  (TS = '1' and TLT = '0' and TEQ = '0')  then
                    NextState <= WAIT_NEG;      -- no trigger yet
                elsif  (TS = '1' and (TLT = '1' or TEQ = '1'))  then
                    NextState <= TRIGGER;       -- got a trigger
                else
                    NextState <= IDLE;          -- trigger slope changed
                end if;

            when  TRIGGER =>            -- in the trigger state
                NextState <= IDLE;      -- always go back to idle
                when  OTHERS =>            -- in any other state
                NextState <= IDLE;      -- always go back to idle

        end case;


        if  Reset = '1'  then           -- reset overrides everything
            NextState <= IDLE;          --   go to idle on reset
        end if;

    end process transition;


    -- storage of current state (loads the next state on the clock)
```

```
    process (clk)
    begin

        if  clk = '1'  then             -- only change on rising edge of clock
            CurrentState <= NextState;  -- save the new state information
        end if;

    end process;


end  assign_statebits;
```

In addition, if an auto trigger is enabled, a trigger event will be automatically generated after 5 ms regardless of whether the trigger level was passed at the correct slope. This is done by a counter using the system clock where the count is enabled if auto trigger and the general trigger itself are enabled. When 5 ms has been reached, a trigger event is generated. The auto trigger counter is reset if the system is reset or whenever an auto trigger is generated.

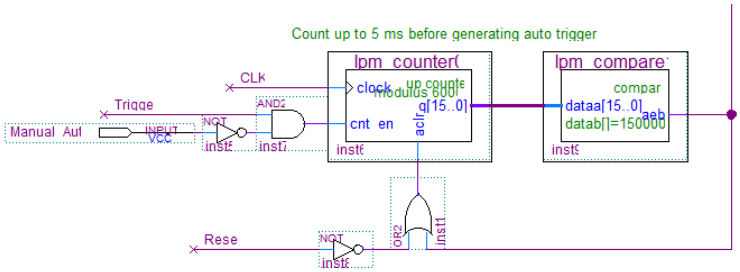


Figure 89: Auto trigger timeout if auto trigger is enabled

Once either an auto or manual trigger is generated, the next step is to latch the trigger signal as well as wait for the appropriate amount of trigger delay. The signal is latched since when the signal value changes in the next clock, the trigger signal will be lost. However, the the trigger event must be present until the FIFO is written to. This is done by latching the value to a set-reset flip flop when trigger is enabled, which only be reset if trigger is disabled or the system is reset. This is since the reset pin of the flip flop is tied to ground, so it will never be reset.

This trigger event is used to enable a counter to wait for the trigger delay before writing the value of the signal to the FIFO. The counter counts every sampling clock until the trigger delay value is reached. At that point, the write request is latched, as the FIFO must be written to until it becomes full (480 data points), even though the counter will only be equal to the trigger delay value for one clock. To allow for a maximum trigger delay of 50,000 times the sampling frequency, the trigger delay is a 16 bit value of how many sampling clocks to wait for before recording the analog to digital converted signals. This is depicted in the below hardware logic.

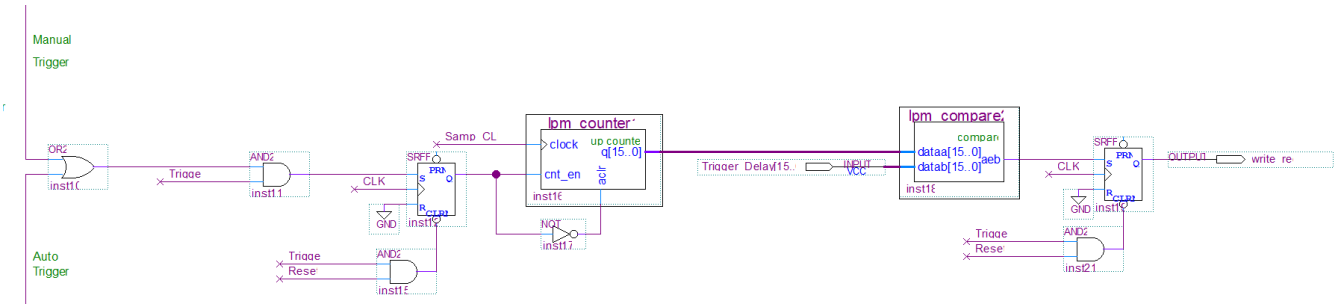


Figure 90: Hardware logic to start writing data after a trigger delay set by user

1.18.1 PIO Setup

Each parameter that the user can set is output from the CPU as a peripheral I/O. It is output to the trigger controller as a paramter to determine when/if a trigger event should happen. The sampling rate is stored as a 25 bit value that is output to the trigger as the number of 100s of nanoseconds between trigger events. It is located between addresses 0x0000_0130 and 0x0000_013F in the address map.

The trigger slope is a 1 bit value that is output to the trigger controller. A 1 indicates that a positive trigger slope is set, and a 0 indicates that a negative trigger slope is set. It is located between addresses 0x0000_0120 and 0x0000_012F in the address map.

The trigger delay is a 16 bit value output to the trigger controller, storing the number of sampling clocks to wait between a trigger event happening and analog to digital converted signals being recorded in the FIFO. It is located between addresses 0x0000_0110 and 0x0000_011F in the address map.

The trigger level is a 7 bit resolution value that indicates the level the top 7 bits of the ADC data must pass through for a trigger event to be generated. It is located between addresses 0x0000_0100 and 0x0000_010F in the address map.

The trigger enable signal is a 1 bit flag indicating whether the trigger is enabled (1) or disabled (0). This is output to the trigger controller to determine whether a trigger event should be generated at all. This is located between addresses 0x0000_00F0 and 0x0000_00FF in the address map.

The manual/auto trigger signal is a 1 bit flag indicating whether a manual trigger (1) or auto trigger (0) should be used to generate a trigger event. If auto trigger is disabled, the trigger event will only be generated if the signal passes the trigger level with the correct slope. This may lead to situations where no trigger is generated, so no signal is displayed. However, if auto trigger is enabled, if this is the case, a trigger will be forced after a certain amount of time, causing a signal to be recorded and displayed. If auto trigger is enabled, but a manual trigger occurs before the auto trigger timeout period, the manual trigger will be used. This PIO is located between addresses 0x0000_00D0 and 0x0000_00DF in the address map.

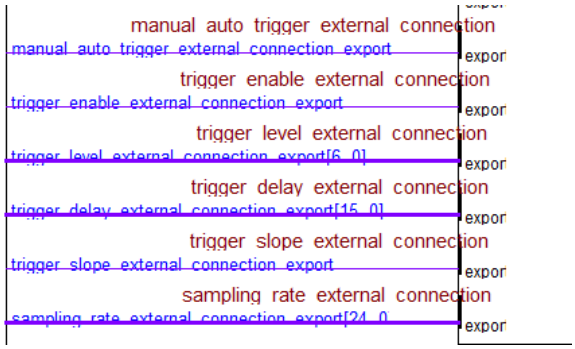


Figure 91: PIOs for trigger controller output from CPU

1.18.2 Trigger Controller Simulation

While the trigger controller was not tested in the real oscilloscope, the results were simulated with the following results.

The settings are set to a positive trigger slope, trigger level of 0b0000100, manual trigger, and sampling rate of 100 ns. When this is the case, there is a sampling clock once every three system clocks. Values read from the ADC are 0b00000101, 0b00000110, 0b00001001, and other larger signals. When the trigger level is passed (with signal 0b00001001), the write request signal is generated 1 sample clock later as the trigger delay is one sampling clock. The write request signal is enabled until the trigger enable is pulsed low (which it will be during the read of the FIFO in software at the end of the trigger cycle).

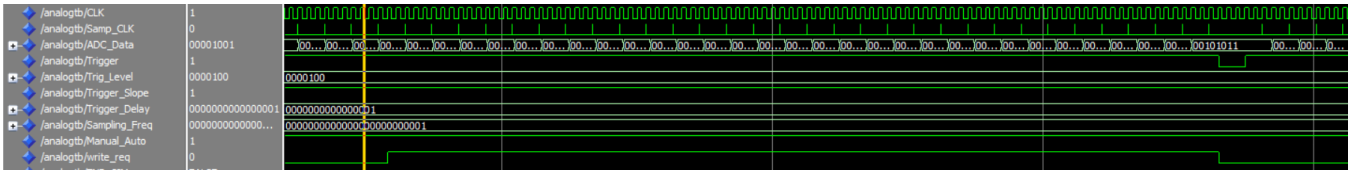


Figure 92: Simulation when signal has positive slope and user sets positive trigger slope

When the same parameters are used, except that the ADC data passes the trigger level with a negative slope, no write request is generated since the trigger slope set by the user is a positive slope.

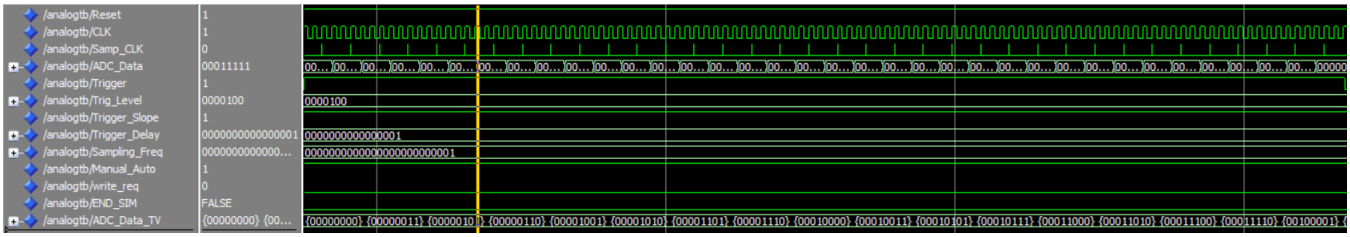


Figure 93: Simulation when signal has negative slope and user sets positive trigger slope

The opposite happens when there is a negative slope. Thus, when the trigger delay and level are the same, a write request signal is only generated when the trigger level is passed by the ADC signals moving from a value higher than the trigger level to one lower than it. Thus when the ADC signal is lowered from 0b00101000 to 0b00000011, the write request signal is generated after the data passes 0b00001001, with a delay of 1 sampling clock.

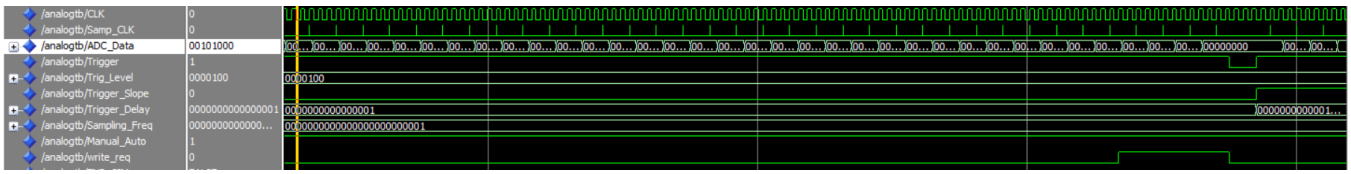


Figure 94: Simulation when signal has negative slope and user sets negative trigger slope

When the trigger delay is increased to 8 sampling times (0b1000), the write request is generated 8 sample clocks after the data passes the trigger level, as seen below. Thus, the data passes the trigger level when it is 0b1001, but 8 sampling clocks pass before write request becomes active.

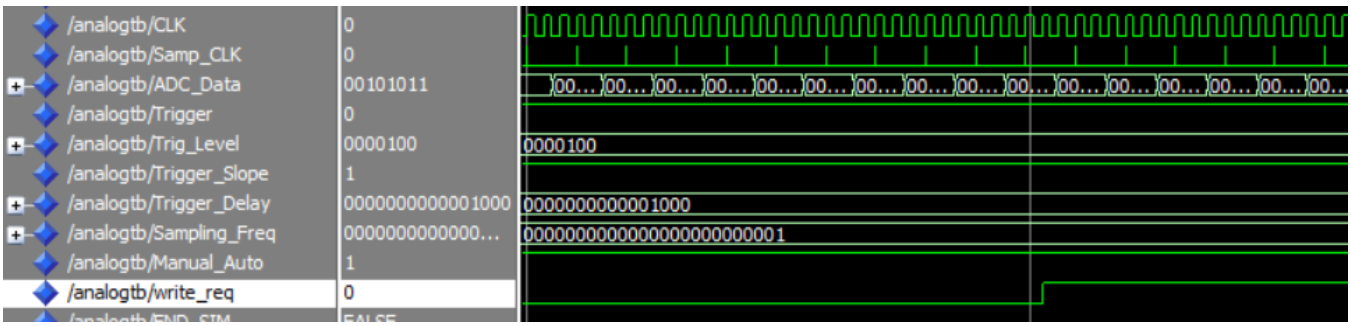


Figure 95: Simulation when trigger delay is increased

When the trigger level is increased to 0b1011 (11), the write request only becomes active the trigger delay number of clocks after the signal passes the higher trigger level. In comparison to the first simulation case (Figure 92) considered with all same parameters except trigger level, it is obvious that the write request becomes active later.

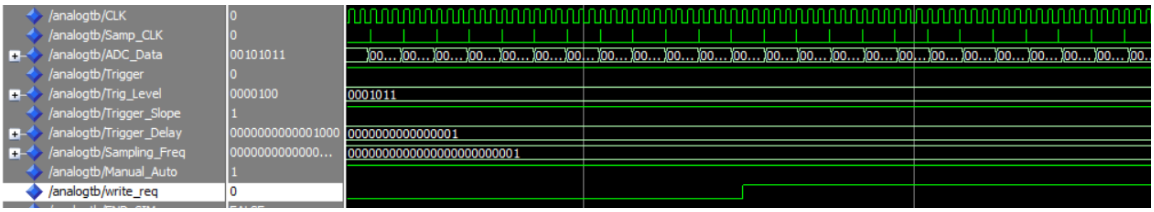


Figure 96: Simulation when trigger level is raised

When the sampling frequency is decreased, or amount of time between samples is increased, the sampling clock pulses less often compared to the system clock. This is seen in the following simulation when the sampling clock is set to 1 us. Thus, after the trigger level is passed, the write request becomes active after the trigger delay number of times of rising edges in the sampling clock, leading to the following measurements.

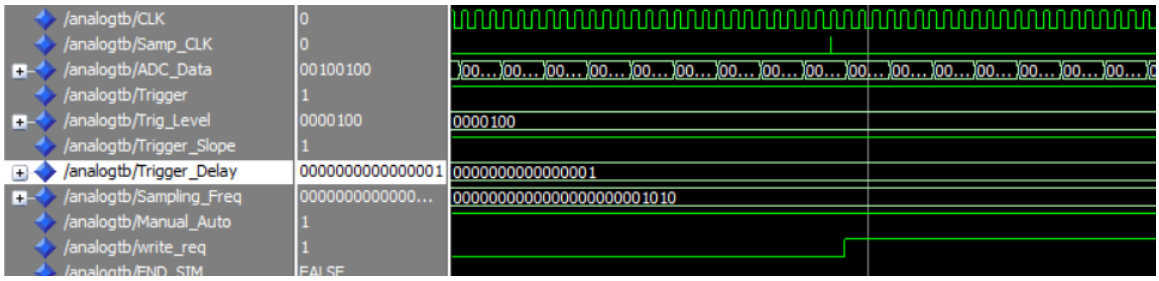


Figure 97: Simulation when sampling frequency is increased

Based on the results of these simulations, we can be fairly confident that the trigger controller works as expects in a real life oscilloscope.

1.19 FIFO

The analog data sampled by the ADC must then be stored. For this, a hardware FIFO is used. When a write request signal is generated by the trigger controller, the FIFO will start recording 8 bits of data from the ADC every sampling clock. When the FIFO becomes full (it can store 480 bytes of data), an interrupt will be generated by the Qsys peripheral I/O it inputs into the CPU. This will cause the software to read every sample measured and stored in the FIFO one byte at a time using the Data_Read signal after disabling triggers while it does this. After the FIFO has been completely read, the trigger enable will be re-enabled, and more signals will be captured from the ADC and stored in the FIFO.

This FIFO is configured with the following logic.

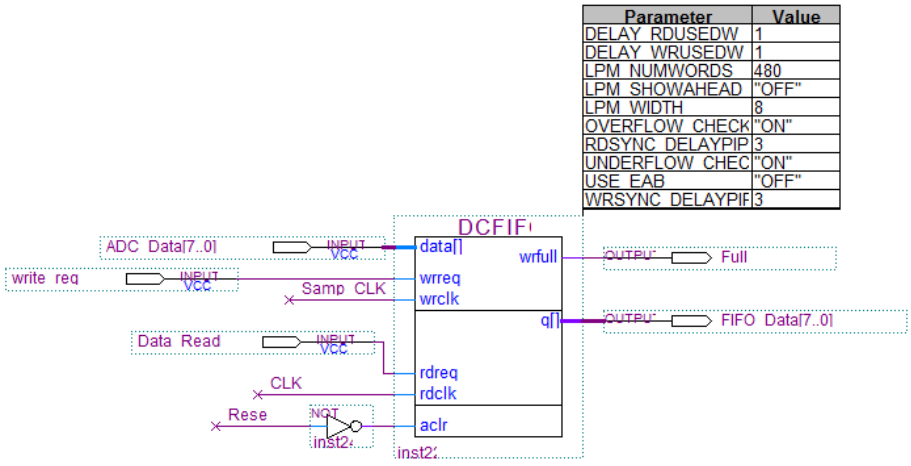


Figure 98: FIFO block and signals input and output from it

The FIFO signals are configured in the CPU as follows. The FIFO_Full signal from the FIFO generates an interrupt in the CPU, indicating that the event handler must read the data from the CPU before it is overwritten by new analog data. This is a 1 bit signal input to the CPU. The edge capture register (which has bit-clearing enabled) will synchronously capture rising edges, and generate an edge IRQ, when any unmasked bit in the edge-capture register is logic true. The peripheral I/O is located between addresses 0x0000_00E0 and 0x0000_00EF in the address map.

The signal the CPU uses to respond to reading the FIFO is the Data_Read signal which causes the FIFO to output the next byte of data stored in it. The CPU will read this data and store it in a buffer in the SRAM. This signal is pulsed after the CPU reads each byte. It is a 1 bit signal output, and is located between addresses 0x0000_00C0 and 0x0000_00CF in the address map.

The last peripheral I/O used by the FIFO is the FIFO data itself, which is a 8 bit value holding a byte of data sampled from the ADC that was stored in the FIFO. This input to the CPU will be read by the software and stored in a buffer in SRAM. The PIO is located between addresses 0x0000_00A0 and 0x0000_00AF in the address map.

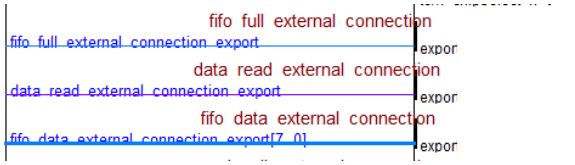


Figure 99: PIO connections between FIFO and CPU

1.20 Appendix - Fixes

This is a description of changes to the oscilloscope made between design and implementation.

A 25 KOhm resistor was incorrectly used in series with the Data output pin in the Serial configuration device. Instead, as specified in the connection guidelines for the FPGA, a 22 Ohm resistor was used instead. The old and then new schematics for the Serial ROM are shown below.

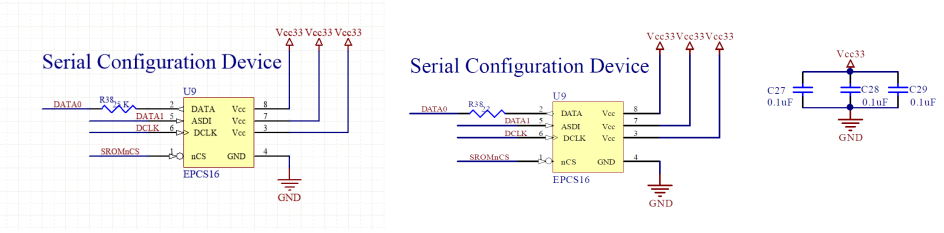


Figure 100: Old (left) and new (right) schematics for Serial ROM

Initially, the second push button was designed to be input to I/O pin 19 of the FPGA. However, pin 19 is not an I/O pin on the FPGA, so the push button pin was wired to be input to pin 37 of the FPGA instead. This change is reflected in the following schematics of the first old and then new version.

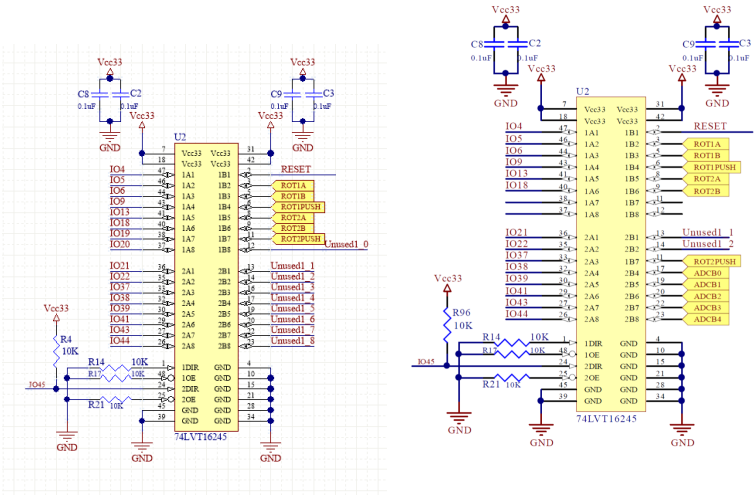


Figure 101: Old (left) and new (right) schematics for buffers for rotary encoder

Since the original VRAM the oscilloscope was designed for, the MSM514252A, was not available, the MT42C4255 VRAM was used instead. The schematic was preserved since the two VRAMs have the same pinout, but the timing cycles and controller state machine changed slightly based on new timing constraints.

The switching regulator initially used to provide a 25 V supply for the backlight of the LCD display from the 5 V power line was not compatible with the circuit. On multiple tries, after a couple of hours of use, the regulator encountered an internal short between pins 4 and 5, the ground pin and collector of the internal power transistor, respectively. To be able to power the backlight, a DC to DC converter from 5 V to 24 V was used instead.

The old schematic of the switching regulator is shown below.

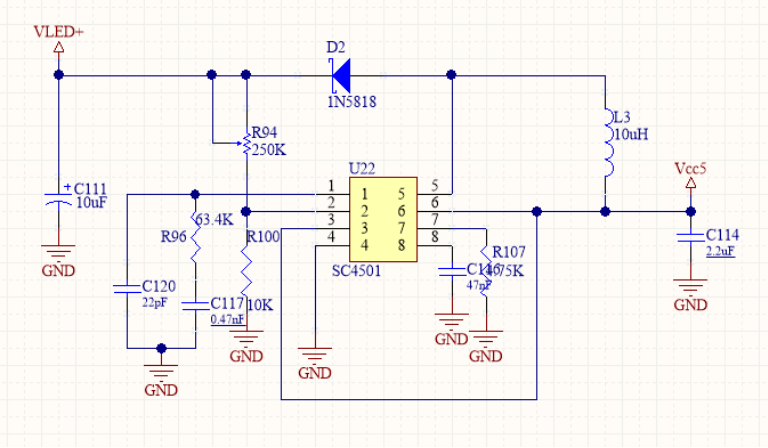


Figure 102: Switching regulator connections initially used for LCD backlight power

Instead, the following DC to DC converter schematic is used.

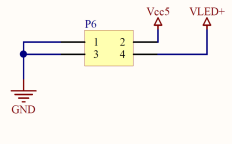


Figure 103: Direct converter from 5 to 24 V used instead of switching regulator

The display schematic was also incorrect, in that two pins (3 and 29) were connected to the supply voltage rather than ground. This was fixed by cutting the traces to these two pins and wiring them to ground instead. This change is reflected in the old and then new schematic for the 40 pin ZIF connector to the display.

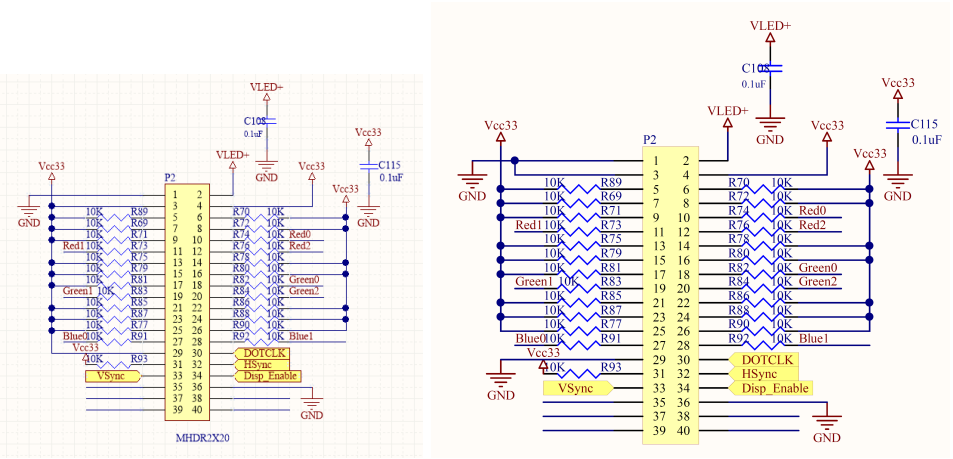


Figure 104: Old schematic (left) and new schematic (right) for display connections

The footprint for the Analog to Digital Converter was incorrect initially, so an adapter board was used that was wired to the main oscilloscope board. However, there were no changes in the schematic itself.

The last change in the design of the oscilloscope was in the front end. Initially, the front end was designed to only have a range of -10 V to 10 V. In addition, the op amps the front end was designed with had a limited bandwidth that did not meet the desired requirements for the maximum sampling frequency of the oscilloscope. Thus, the front end was changed from the first schematic to the next one.

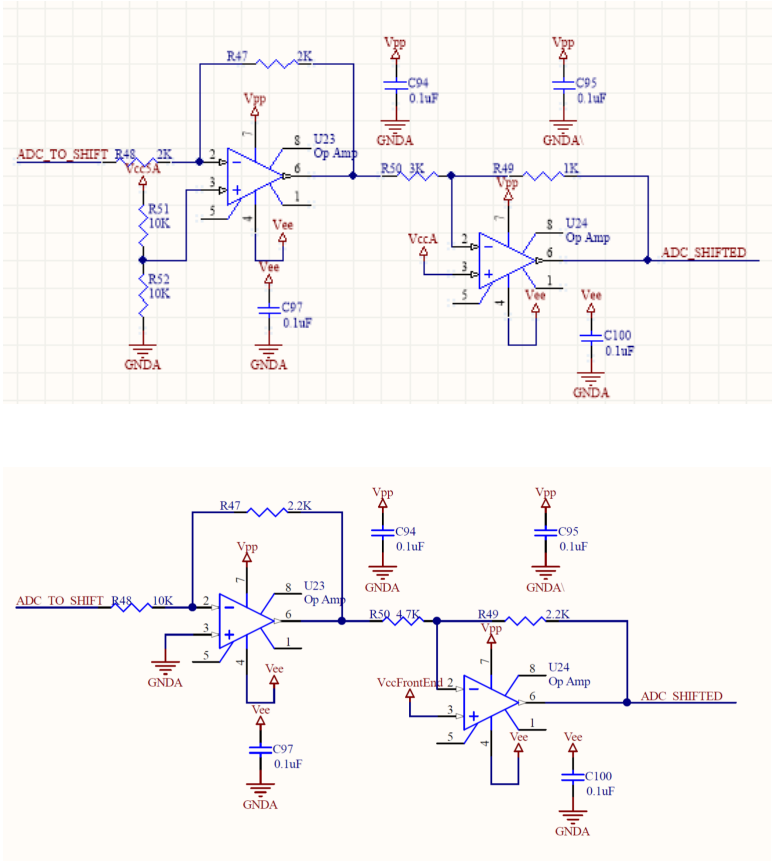


Figure 105: Schematics of old (top) and new (bottom) front ends

2 Oscilloscope Software

2.1 Overview

The other major component of the digital oscilloscope is the software that interfaces with the hardware to allow for control of the input and output of the system. The main components that interface with each other are the dial input, the display output, and analog input. The interactions between the parts of the software is summarized in the following block diagram.

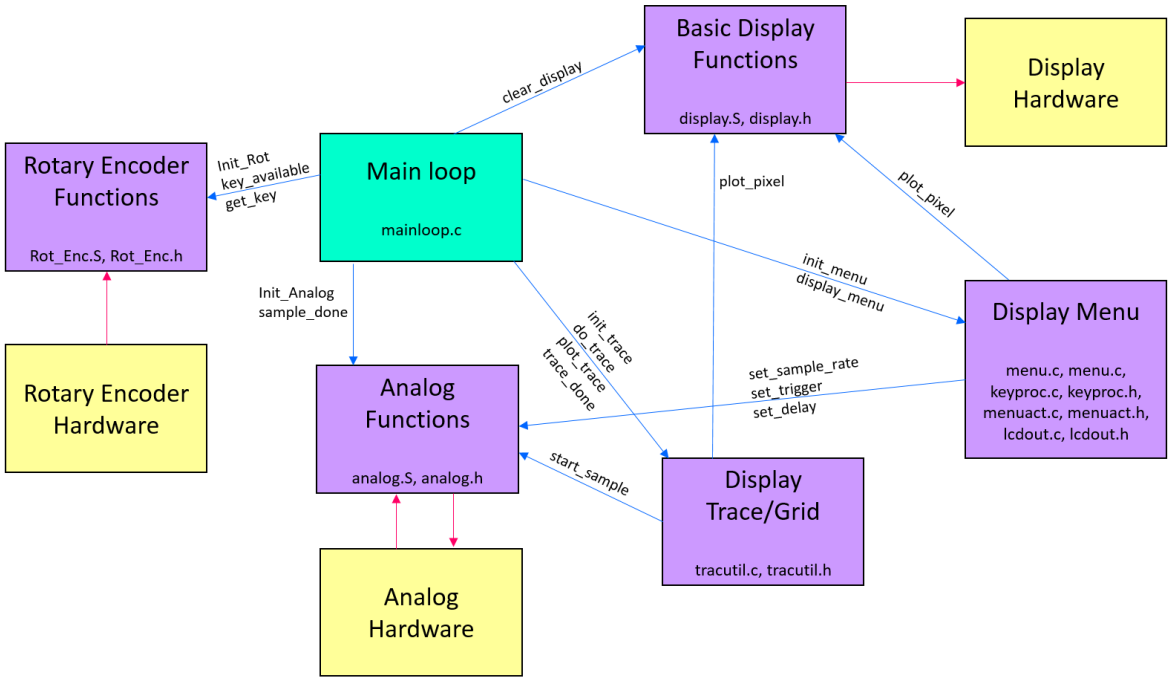


Figure 106: Block diagram of interactions between major components of the software and the functions linking them

The main interactions are summarized in this overview.

Rising edges in the input for the push buttons and clockwise/counterclockwise turns of the Peripheral I/Os from the rotary encoders cause interrupts in the CPU. These are handled by a rotary encoder event handler, which stores the key events in an internal variable to signify which event occurred (Menu, Left, Right, Up, Down, or Other). This event handler is initialized during the start up section of the main loop. The main loop then regularly checks if a key is available, and if so, retrieves the key press event. This is then stored in a key buffer within the main loop.

These key press events are then used to change how the menu and grid are displayed. The Menu button will cause the menu to turn on and off. The Up and Down events will move up and down in the menu to change different features. The Left and Right events will rotate through the options for a feature backward and forward, respectively.

Depending on what features are being changed in the menu, these functions may change the following analog settings: sample rate, trigger slope, trigger delay, and trigger level. These are done through mutator functions which change the value of the setting in the peripheral I/O set up in the CPU for these values. This will then interface with the analog hardware to change how the trigger is generated for the data, or what data is captured. This new data will then be read from the hardware FIFO by the analog functions, and passed back to the main loop through the sample_done function. The new data is read every time the FIFO becomes full, which causes an interrupt in the CPU. The event handler (which is initialized in the

startup portion of the main loop) reads every byte from the FIFO and stores it in an internal buffer. After it has read all the data stored in the FIFO, it re-enables triggers so that a new analog signal can be captured and the trace for it displayed.

After the new sample is captured, it is displayed on the LCD through the trace displaying functions. The trace displaying functions start a sample in the analog functions, so that the trace of it can be displayed. These functions (as well as the menu display functions) also use the basic display functions to facilitate plotting specific pixels on the display. The basic display functions interface with the VRAM hardware to write to pixels in the VRAM. These new values for the pixels will then be output from the SAM serially and transferred to the LCD, and eventually displayed. The main loop also interfaces with the basic display functions to clear the display when the system is powered or reset. This is done by setting all the pixels in the VRAM to the background color.

For the most part, the menu and trace display functions are treated as black boxes, and their interactions with external components are studied. In addition, while the analog functions were designed, they were not tested in the oscilloscope. Thus, for the current version of the oscilloscope, the trace displayed is for a ringing square wave. While the analog settings can be changed on the display, the results of these changes are not reflected on the trace (no analog signal is actually measured).

2.2 Dial Input

User input for the oscilloscope is through the two rotary encoders, or dials. These dials can each be pushed down or turned clockise or counterclockwise. The turns of the encoders are decoded in hardware, and the push buttons are also debounced in hardware. The results of this hardware logic is a 6 bit value:
Bit 0 (LSB) : Clockwise Turn of Right Rotary Encoder
Bit 1 : Counterclockwise Turn of Right Rotary Encoder
Bit 2 : Push of Right Rotary Encoder
Bit 3 : Clockwise Turn of Left Rotary Encoder
Bit 4 : Counterclockwise Turn of Left Rotary Encoder
Bit 5 (MSB) : Push of Left Rotary Encoder

Whenever there is a rising edge in any bit of this value, an interrupt occurs in the CPU. The Init_Rot function, which is called by the main loop during startup, sets up the interrupt handling process. Using the IRQ number for the rotary encoders as well as the interrupt controller ID, the event handler function is set up to be called whenever there is an edge using the alt_ic_isr_register function. This function sets up an interrupt service routine, and writes the event handler function and its arguments (none in this case) to the interrupt vector table. This function also clears any pending interrupts so any new actions by the user will be recorded and processed.

When an interrupt actually happens, the event handler is reached. The value of the PIO with the rising edge for a bit is stored in the edge capture register of the PIO core. This can be read by the event handler to know which bit caused the interrupt. After this, the value is checked for validity (two key events at same time disallowed), and the associated key value is stored in the keyCodeStored shared variable. This can be used by other status and access functions to know when a key press has been recorded. Interrupts are also enabled again and the current edge cleared from the register.

The main loop regularly queries the key_available function, which checks whether the key code stored in the shared variable is KEY_ILLEGAL (no key has been pressed yet, or an illegal key combination). If it is not, then a valid key is available, and that information is returned to the caller.

If the main loop finds that there is a key event available, then it will use the get_key function to get the actual value of the key code. This function checks again that there is a key available (if not, the function blocks until there is one). After the value of the key code is read from the shared variable, a KEY_ILLEGAL value is writen to the shared variable. This makes sure that each key event is only recorded once. The block diagram of interactions between components of the rotary encoder software is below.

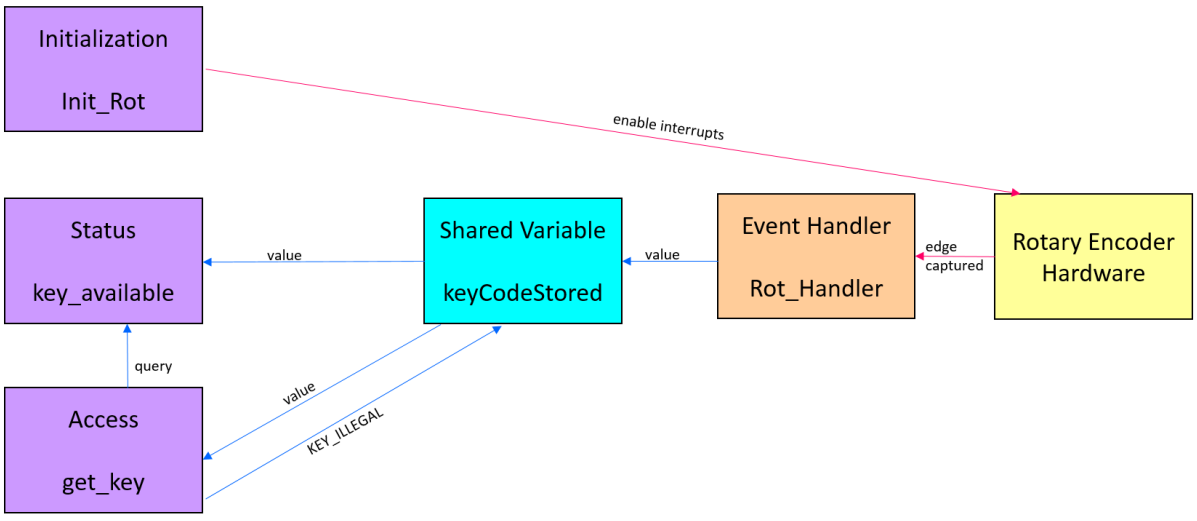


Figure 107: Block diagram of interactions between functions and variables in rotary encoder component

The code for this is included in the next pages (Rot_Enc.S, Rot_Enc.h)

```
////////////////////////////////////////////////////////////////////////////////
//                                                                            //
//                                   ROT_ENC                                  //
//                            Rotary Encoder Handling                         //
//                         Digital Oscilloscope Project                       //
//                                   EE/CS 52                                 //
//                                                                            //
////////////////////////////////////////////////////////////////////////////////


// This file contains the routines to initialize rotary encoder driven interrupts
// as well as to handle these events. In addition, this provides an interface
// with the user interface code with a function that checks whether an event
// has occurred and key retrieved as well as returns the key press in a form
// usable by the front end software.
//
// Table of contents
// Init_Rot - Initializes edge sensitive interrupts from the bits of the
//  rotary encoder PIO
// Rot_Handler - Handles interrupts from any of the bits of the PIO by
//  matching up the correct key to the event
// key_available - Checks whether there is a valid key available from the
//  rotary encoders
// get_key - If there is a valid key returns the key code to the front end
//  does not exit until there is a valid key
//
// Revision History:
//    05/14/17  Maitreyi Ashok      initial revision
//    05/15/17  Maitreyi Ashok      cleaned up code
//    05/17/17  Maitreyi Ashok      debugged inconsistent rotary encoders
//    05/20/17  Maitreyi Ashok      updated comments


// contains key code definitions
#include "interfac.h"
// contains general constant definitions
#include "scopedef.h"
// contains rotary encoder PIO definitions
#include "system.h"
// contains masks to interface with PIO core register
#include "Rot_Enc.h"
// contains macros to interface with the stack
#include "macros.m"


.section .text


// Init_Rot
//
// Description:      This function initializes interrupts from the ROTARY_ENCODER
//                   PIO as well as installs the handler for it. This interrupt
//                   contains the push button as well as the clockwise and
//                   counterclockwise turns of both rotary encoders. All
//                   possibilities of events are handled by the same interrupt
//                   handler and are part of the same PIO.
//
// Operation:        Installs the handler using the IRQ constants of the
//                   ROTARY_ENCODER PIO in the alt_ic_isr_register function which
//                   enables hardware interrupts so that the software can use an
//                   interrupt service routine. Then, the interrupts for the used
//                   bits of this PIO are enabled and the bits used in the edge
//                   capture register for this PIO are cleared so that any pending
//                   interrupts will be turned off.
```

```
//
// Arguments:        None
// Return Value:     None
//
// Local Variables:  Rot_Enc_PIO [r9] - contains address of the Rotary Encoder
//                          PIO register
// Shared Variables: None
// Global Variables: None
//
// Input:            None
// Output:           None
//
// Error Handling:   The function does not return until the installing of the
//                   handler using alt_ic_isr_register succeeds
//
// Algorithms:       None
// Data Structures:  None
//
// Registers Changed:r2, r4, r5, r6, r7, r8, r9
// Stack Depth:      2 words
//
// Author:           Maitreyi Ashok
// Last Modified:    05/14/17   Maitreyi Ashok  Initial revision
//                   05/17/17   Maitreyi Ashok  Changed ldw/stw to ldwio/stwio
//                   05/20/17   Maitreyi Ashok  Updated comments

.global Init_Rot
.align 4
.type Init_Rot, @function

Init_Rot:
Set_up_handler:
    movui   r4, ROTARY_ENCODER_IRQ_INTERRUPT_CONTROLLER_ID
    movui   r5, ROTARY_ENCODER_IRQ      // move the rotary encoder interrupt
                                        // id and IRQ into registers as arguments
                                        // to the installer function
    movia   r6, Rot_Handler             // function pointer to event handler is
                                        // another argument
    mov     r7, zero                    // pass a null pointer as isr_context
                                        // argument since it is unused
    PUSH    ra                          // store return address on stack before
                                        // calling another function
    PUSH    zero                        // store a null pointer for the flags
                                        // argument since it is also unnecessary
Call_handler_setup:
    call    alt_ic_isr_register         // install the event handler for the rotary
                                        // encoder interrupt
    bne     r2, zero, Call_handler_setup    // if installing failed, try again
Renable_interrupts:
    POP                                 // remove the argument from the stack
    POP_VAL ra                          // restore the return address
    movia   r9, ROTARY_ENCODER_BASE     // get the rotary encoder PIO register address
    ldwio   r8, 8(r9)                   // and the value in the interruptmask register
    ori     r8, r8, ENABLE_INT          // enable interrupts and store the enabled
    stwio   r8, 8(r9)                   // value to the interruptmask register

    ldwio   r8, 12(r9)                  // get the value in the edgecapture register
    ori     r8, r8, ENABLE_INT          // clear any pending interrupts and store
    stwio   r8, 12(r9)                  // the cleared value into the edgecapture
                                        // register
    ret

// Rot_Handler
//
```

```
// Description:       This function handles interrupts due to the rotary encoders.
//                    Hardware interrupts by turning either of the rotary encoders
//                    clockwise or counterclockwise or pushing them will
//                    be registered in the ROTARY_ENCODER PIO, causing this handler
//                    function to be called. This function determines the key
//                    code that the hardware event corresponds to and stores this
//                    code to be used by the main loop software. If an illegal key
//                    is received, no value is stored.
//
// Operation:         This function first disables interrupts during the handling
//                    of the event to avoid any extra events happening. Then,
//                    the edge captured for the event is stored in a register to
//                    determine what the event signified. If there is still an old
//                    key code stored (a valid one), this current event is ignored
//                    to allow the old key code to be retrieved. In addition, if
//                    the edge captured is somehow illegal, the edge is ignored and
//                    no key code is stored. Otherwise, the key code is stored to be
//                    later retrieved by the main loop. The key code options are
//                    MENU (push button 1), UP (rotary encoder 1 counter clockwise),
//                    DOWN (rotary encoder 1 clockwise), LEFT (rotary encoder 2
//                    counter clockwise), RIGHT (otary encoder 2 clockwise), and
//                    OTHER (push button 2). After storing the key code, interrupts
//                    are enabled again and the edge capture register is cleared to
//                    allow for future hardware interrupts to be registered.
//
// Arguments:         None
// Return Value:      None
//
// Local Variables:   Rot_Enc_PIO [r9] - contains address of the Rotary Encoder
//                           PIO register
//                    edge_captured [r8] - contains the value stored in the edge
//                           capture register that caused the interrupt
// Shared Variables: keyCodeStored - stores value of keycode corresponding to user
//                           input. Either Left, Right, Up, Down, Menu, or Illegal.
// Global Variables: None
//
// Input:             Rotary encoders are turned clockwise or counterclockwise
//                    or the pushbutton in one of the encoders is pushed
// Output:            None
//
// Error Handling:    If the interrupt is due to an invalid turn or push (either
//                    nothing is registered or a combination of turns is registered)
//                    then ignore the value retrieved
//
// Algorithms:        None
// Data Structures:   None
//
// Registers Changed:r8, r9, r10, r12, r14, r15
// Stack Depth:       0 words
//
// Author:            Maitreyi Ashok
// Last Modified:     05/14/17   Maitreyi Ashok  Initial revision
//                    05/17/17   Maitreyi Ashok  Changed ldb/stb to ldbio/stbio
//                    05/20/17   Maitreyi Ashok  Updated comments

.global Rot_Handler
.align 4
.type Rot_Handler, @function

Rot_Handler:
    movia   r9, ROTARY_ENCODER_BASE     // Get the address of the PIO register
    ldbuio  r8, 8(r9)                   // Retrieve the current value of the
    andi    r8, r8, DISABLE_INT         // interruptmask register and disable
    stbio   r8, 8(r9)                   // interrupts during the handling routine
```

```
    ldbuio  r8, 12(r9)                  // Get the value stored in the edge capture
                                        // register as the interrupt that is registered
    movia   r10, keyCodeStored          // Get the address of the key code variable
    movi    r14, TRUE

Check_Validity:
    ldb     r15, 0(r10)                 // Load the old key code
    cmpeqi  r15, r15, KEY_ILLEGAL       // If it is not KEY_ILLEGAL, then the old
    bne     r15, r14, Error             // value has not been retrieved so ignore
                                        // the current one
    movi    r12, KEY_LEFT               // If the current event is a combination of
    ori     r12, r12, KEY_RIGHT         // both left and right turns, then it is
    cmpeq   r12, r8, r12                // an illegal turn and should not be stored
    beq     r12, r14, Error
    movi    r12, KEY_DOWN               // If the current event is a combination of
    ori     r12, r12, KEY_UP            // both up and down turns, then it is an
    cmpeq   r12, r8, r12                // illegal turn and should not be stored
    beq     r12, r14, Error
    beq     r8, zero, Error             // If the current event has no value, it
                                        // is not stored
Store_Value:
    stb     r8, 0(r10)                  // If there is no error in the value
    jmpi    Done_Handler                // store it in the shared variable key code
Error:
Done_Handler:
    ldbuio  r8, 8(r9)                   // Get the current value of the interrupt
    ori     r8, r8, ENABLE_INT          // mask register and enable interrupts
    stbio   r8, 8(r9)                   // due to the rotary encoder PIO
    movi    r8, ENABLE_INT              // Clear the edge capture register for all
    stwio   r8, 12(r9)                  // interrupts in this PIO for any pending
                                        // interrupts
    ret

// key_available
//
// Description:      This function checks if a key event is available. This would
//                   be a result of an edge being captured in the peripheral I/O,
//                   causing the event handler to record the key event in the
//                   keyCodeStored variable. This function checks if the variable
//                   has a valid key value, and if so returns true. Otherwise,
//                   the function returns that no key is avaiable for retrieval.
//
// Operation:        The function checks if a key is available by checking the
//                   value of the shared variable for the key code. This key code
//                   is then compared to the key code for an illegal key. If the
//                   key codes match, the function returns false since a valid key
//                   is not available. This happens when either an illegal key
//                   combination is recorded, or if no key event has been recorded
//                   at all. If the key code does not match that for an illegal key,
//                   then the function returns true since a key is available to be
//                   retrieved from the shared variable.
//
// Arguments:        None
// Return Value:     available [r2] - returns whether a valid key code is stored
//                          due to a hardware interrupt (TRUE) or not (FALSE)
//
// Local Variables:  None
// Shared Variables: keyCodeStored - stores value of keycode corresponding to user
//                          input. Either Left, Right, Up, Down, Menu, or Illegal.
// Global Variables: None
//
// Input:            Rotary encoders are turned clockwise or counterclockwise
//                   or the pushbutton in one of the encoders is pushed
```

```
// Output:           None
//
// Error Handling:   None
//
// Algorithms:       None
// Data Structures:  None
//
// Registers Changed:r8, r9, r2
// Stack Depth:      0 words
//
// Author:           Maitreyi Ashok
// Last Modified:    05/14/17   Maitreyi Ashok  Initial revision
//                   05/20/17   Maitreyi Ashok  Updated comments

.global key_available
.align 4
.type key_available, @function

key_available:
    movia   r8, keyCodeStored       // Retrieves the address of the store key code
    ldbu    r9, 0(r8)               // and the value
    cmpeqi  r9, r9, KEY_ILLEGAL     // Check if the key code is key illegal
    bne     r9, zero, illegal_key   // If it is return that key not available
legal_key:
    movi    r2, TRUE                // If key code is not illegal, return
    jmpi    end_key_available       // that key is available
illegal_key:
    movi    r2, FALSE               // If key code is illegal return that key
                                    // is not available
end_key_available:
    ret

// get_key
//
// Description:      This function returns the key code of the key event that has
//                   taken place. The function blocks if no valid key event is
//                   available, and it will only return when a valid key is
//                   present. It also resets the value of the key code shared
//                   variable to hold an illegal key value. This ensures that
//                   that each user input key event is only recorded and handled
//                   once. Then, the next time the main loop checks if a key is
//                   available, there will not be a valid key until the user
//                   interacts with the system again.
//
// Operation:        This function checks if there a valid key available using the
//                   key_available function. If there is, the actual value of the
//                   key code is retrieved from the shared variable. If not, the
//                   function repeatedly checks for an available key until there
//                   eventually is one. Thus, the function blocks if called before
//                   checking if there is a key event. Once the key code is read,
//                   it is stored as the return value. The shared variable key
//                   code is reset to an illegal key code value. This is to ensure
//                   that the specific key event will only be handled once.
//
// Arguments:        None
// Return Value:     keyCode [r2] - returns the key code if its valid. Does not
//                          return until has valid key code
//
// Local Variables:  None
// Shared Variables: keyCodeStored - stores value of keycode corresponding to user
//                          input. Either Left, Right, Up, Down, Menu, or Illegal.
// Global Variables: None
//
// Input:            Rotary encoders are turned clockwise or counterclockwise
```

```
//                   or the pushbutton in one of the encoders is pushed
// Output:           None
//
// Error Handling:   None
//
// Algorithms:       None
// Data Structures:  None
//
// Registers Changed:r8, r9, r2
// Stack Depth:      1 word
//
// Author:           Maitreyi Ashok
// Last Modified:    05/14/17   Maitreyi Ashok  Initial revision
//                   05/19/17   Maitreyi Ashok  Set key code back to illegal after
//                                              reading it
//                   05/20/17   Maitreyi Ashok  Updated comments

.global get_key
.align  4
.type   get_key, @function

get_key:
    PUSH    ra                      // store the return address before calling
                                    // key available
    call    key_available           // Check if there is a key available
    movi    r8, FALSE               // If there is a key available
    bne     r2, r8, valid_key       // then return that key
    jmpi    get_key                 // Otherwise keep trying until there is a key
valid_key:
    movia   r8, keyCodeStored       // Get the key code actually stored in the
    ldbu    r2, 0(r8)               // shared variable
    stw     r0, 0(r8)               // Reset the shared variable to an illegal
    movui   r9, KEY_ILLEGAL         // key code byte after clearing out the entire
    stb     r9, 0(r8)               // word

    POP_VAL ra                      // restore the return address
    ret


.section .data
.align 4
keyCodeStored: .byte 6          // Contains the key code of the user input event
               .skip 1          // Can be KEY_MENU, KEY_LEFT, KEY_RIGHT, KEY_UP,
                                // KEY_DOWN, or KEY_ILLEGAL
```

```
/*****************************************************************************/
/*                                                                           */
/*                                 ROT_ENC.H                                 */
/*                          Rotary Encoder Definitions                       */
/*                                Include File                               */
/*                        Digital Oscilloscope Project                       */
/*                                 EE/CS 52                                  */
/*                                                                           */
/*****************************************************************************/

/*
 * This file contains the masks for the enabling, disabling, and handling
 * of interrupts from user input via the rotary encoders for the Digital
 * Oscilloscope project. These masks are used to clear and set bits in both
 * interruptmask and edgecapture registers of the PIO core.


 * Revision History:
 *  5/14/17  Maitreyi Ashok  Initial Revision
 *  6/03/17  Maitreyi Ashok  Changed constants for 6 bit PIO value
*/



#ifndef  __ROT_ENC_H__
    #define  __ROT_ENC_H__

/* library include files */
  /* none */

/* local include files */
  /* none */


/* constants */


#define DISABLE_INT 0xFFC0  /* Mask to disable rotary encoder interrupts */
#define ENABLE_INT  0x003F  /* Mask to enable rotary encoder interrupts and
                               clear the rotary encoder edges in the edge
                               capture register */

#endif
```

2.3 Display

The output of the oscilloscope is the display, where the measured waveforms are shown to the user. In addition, the menu that users interact with to choose options for the display and analog sampling is through the display.

The first interaction with the display is when the main loop clears the display during start up. This allows for any garbage stored in the VRAM to not be displayed, and for a clear screen to be shown instead. The clear_display function uses plot_pixel to plot a white pixel at every position in the display. The plot_pixel function writes a pixel of any color to a position in the display, given row and column.

As the oscilloscope is used, the main loop changes the display in two ways: the trace and menu options. The main loop first retrieves the sample to display from the analog functions (described later). Then, this trace is plotted on the display one pixel at a time using the plot_pixel function. If the position the trace must be plotted at should display a grid line or axis instead, then the trace is not plotted. Instead, a horizontal or vertical line is plotted for all the grid and axis lines. These have been configured so that all horizontal scale lines are blue. (The vertical lines were not due to a bug discovered after demonstration of the scope where plot_vline does not take a color as an argument). Thus, the grid lines are distinguished from the trace by a different pixel color.

In addition, the plot_pixel function is used to display the menu in black. Strings of characters are plotted to display the options. When a menu option is chosen, it is shaded in black with the text in white. Otherwise, the menu option is shaded white with text in black.

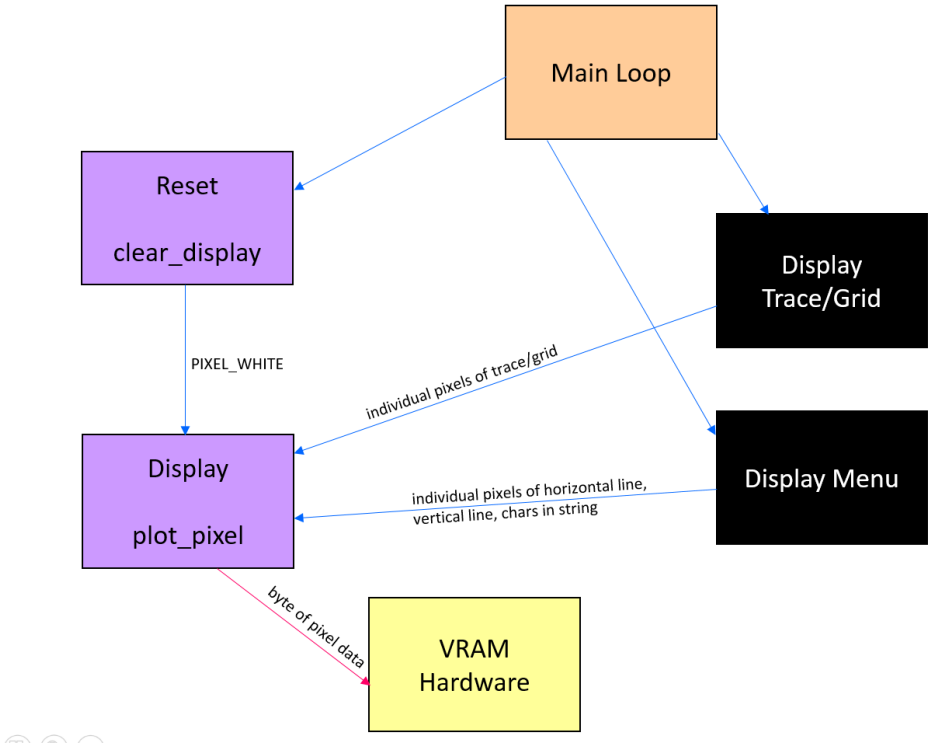


Figure 108: Block diagram of interactions between functions in the display components. Display Trace/Grid and Menu components are treated as black boxes

A sample display as a result of these functions is shown below.

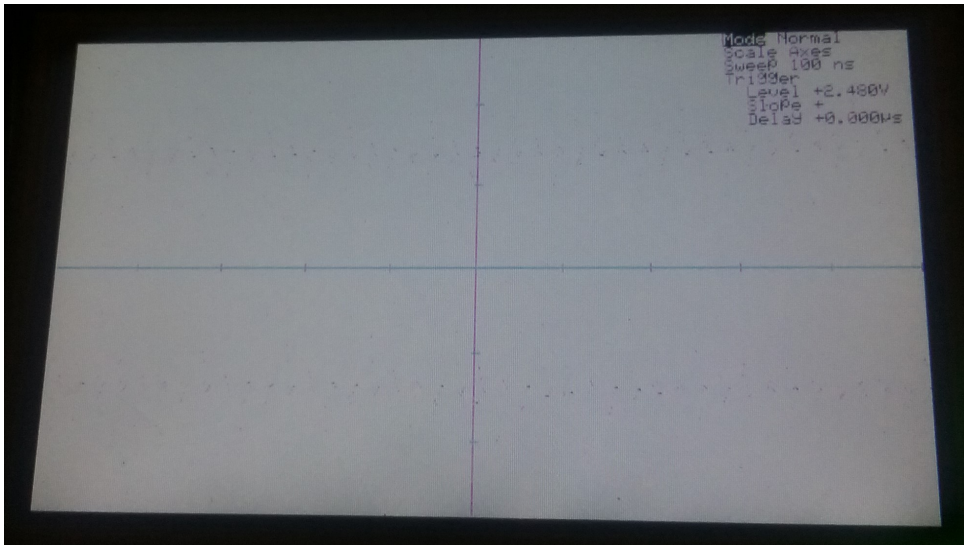


Figure 109: Menu, trace, and axes shown in default display upon power

Due to a bug in either the display or VRAM controller in hardware, there is a shift in which row and column indices of the display correspond to what row and column number in VRAM. Since this bug was not found, a software fix is used instead. This fix shifts all pixels to be displayed in the first column of a row to be at the end of the previous row in the VRAM. In addition, every other column on the display refers to the column to the left in the VRAM.

For example, a pixel in row 5 and column 10 in the display, would be stored at row 5 and column 9 in the VRAM. Also, a pixel in row 19 and column 0 in the display would be stored at row 18 and column 480 in the VRAM. This is done through a mapping from the display row and column system to the VRAM row and column system whenever finding the address of a pixel in the VRAM to plot it.

The following pages contain the following files, which have the described functions
display.S/disp.h - Basic display functions to plot pixel/clear display (written by me)
tracutil.c/tracutil.h - Handle capturing and displaying data (one function updated by me)
lcdout.c/lcdout.h - Doing output to the LCD screen
menuact.c/menuact.h - Carrying out menu actions, display current selection
menu.c/menu.h - Maintaining the menus and handling menu updates
keyproc.c/keyproc.h - Key processing for menu
char57.c - Dot matrix patterns for characters

```
////////////////////////////////////////////////////////////////////////////////
//                                                                            //
//                                  DISPLAY                                   //
//                            BASIC DISPLAY FUNCTIONS                         //
//                          Digital Oscilloscope Project                      //
//                                  EE/CS 52                                  //
//                                                                            //
////////////////////////////////////////////////////////////////////////////////

// This file contains the routines to plot pixels to the display as well as clear
// the display. These functions interface between the high level user interface
// functions and the oscilloscope hardware. This is done by writing bytes directly
// to the VRAM to be output into the LCD as pixels on the display. This allows for
// more complicated images to be shown by plotting one pixel at a time. In addition,
// the display can be reset with the display clearing function, to allow for
// new pixels to be displayed without the old ones still on the LCD.
//
// Table of contents
// get_addr - Local function to perform a software fix for a hardware issue and
//      get address in VRAM corresponding to pixel position on display
// clear_display - Resets display to be all white pixels in all positions
// plot_pixel - Plots a pixel of a certain color in the specific position (based
//      on row and column)
//
// Revision History:
//      5/28/17  Maitreyi Ashok  Initial Revision
//      5/31/17  Maitreyi Ashok  Fixed clear_display
//      6/01/17  Maitreyi Ashok  Added software fix of display
//
// Contains VRAM PIO definitions
#include "system.h"
// Contains display and VRAM constants
#include "disp.h"
// contains macros to interface with the stack
#include "macros.m"
// contains color and display size definitions
#include "interfac.h"


.section .text

// get_addr
//
// Description:      This function calculates the address in the VRAM
//                   corresponding to a row and column of a pixel on the display.
//                   The row and column number are with reference to the top left
//                   corner pixel, which is row 0 and column 0 in the VRAM. The row
//                   number increases moving down in the display, and column
//                   number increases moving right in the display. In addition,
//                   a fix is implemented due to an issue in the VRAM or display
//                   controller to adjust what row/column in the display
//                   corresponds to what row/column in the VRAM.
//
// Operation:        This function finds the address in the VRAM for a specified
//                   row and column in the display. Since the first byte of the
//                   VRAM stores the top left pixel of the display, the address
//                   is found by first finding the starting address of the row
//                   to plot the pixel in the VRAM by multiplying the row number
//                   to the number of columns in the display per row. Then, the
//                   column number is added to include the offset from the start
//                   of the row, and this offset from the start of the VRAM is
//                   added to the starting address of the VRAM itself.
//                   The row and column numbers used for this calculation are
//                   adjusted based on a software fix for a hardware issue that
```

```
//                   could not be debugged. Due to an indexing issue, pixels in
//                   the first column of the display are actually the first pixel
//                   in the row above of the VRAM that is outside of the display's
//                   range. In addition, all other column indices for pixels are
//                   shifted left, so the second column of the display is stored
//                   in the first column of the display for the same row. This
//                   fix is used, and then the adjusted row and column are used
//                   to find the exact address in the VRAM to write the pixel to.
//
// Arguments:        x [r4] - Column number of the pixel to plot
//                   y [r5] - Row number of pixel to plot
// Return Value:     addr [r2] - address in VRAM to write pixel to
//
// Local Variables:  adjusted_x [r12] - Adjusted column number in VRAM
//                   adjusted_y [r11] - Adjusted row number in VRAM
// Shared Variables: None
// Global Variables: None
//
// Input:            None
// Output:           None
//
// Error Handling:   None
//
// Algorithms:       addr = VRAM_BASE + NUM_COLS_VRAM * adjusted_y + adjusted_x
// Data Structures:  None
//
// Registers Changed:r12, r11, r2
// Stack Depth:      0 words
//
// Author:           Maitreyi Ashok
// Last Modified:    06/01/17   Maitreyi Ashok  Initial revision
//                   06/29/17   Maitreyi Ashok  Updated comments

.align 4
.type get_addr, @function

get_addr:
GetArgs:
    mov     r11, r5                 // Save the values of the row (y) and column
    mov     r12, r4                 // (x) number
    bne     r12, zero, MoveLeft     // If not on first column, do not need to
                                    // adjust the row
AdjustFirstCol:
    subi    r11, r11, ROW_FIX       // What is seen in the first column is
    movi    r12, SIZE_X             // stored in the VRAM at end of the previous
    jmpi    CalcAddr                // row
MoveLeft:
    subi    r12, r12, COL_FIX       // If not in first column, just shift left
                                    // since each pixel in display is represented
                                    // by pixel in column before in VRAM
CalcAddr:
    movia   r2, VRAM_BASE           // Calculate address
    muli    r11, r11, NUM_COLS_VRAM // addr = VRAM_BASE + NUM_ROWS_VRAM * row + col
    add     r2, r2, r11
    add     r2, r2, r12             // Return the address in the VRAM corresponding
    ret                             // to pixel in display

// clear_display
//
// Description:      This function resets the LCD display so that every pixel
//                   in the display is white. White is used as the background
//                   color for the display so that other pixels displayed will
//                   be more easily readable regardless of the color of the pixel.
//                   This function is used everytime the system is reset or powered
```

```
//                   on to replace any garbage in the display/VRAM with a clear
//                   screen.
//
// Operation:        This function clears the display by writing a white pixel
//                   to every byte of the VRAM. Through the row transfer and serial
//                   output cycles, these white pixels will be displayed on the LCD
//                   for a clear screen. This is done by starting with the first
//                   row in the VRAM and writing a white pixel to every column
//                   of that row. This same process is repeated for every row
//                   of the VRAM. Since nothing is stored in the regions of the
//                   VRAM that is not displayed, all bytes of the VRAM can be cleared.
//
// Arguments:        None
// Return Value:     None
//
// Local Variables:  row [r5] - row of VRAM clearing pixels in
//                   col [r4] - column of VRAM clearing pixels in
// Shared Variables: None
// Global Variables: None
//
// Input:            None
// Output:           Display is completely clear, with all pixels being white
//
// Error Handling:   None
//
// Algorithms:       Iterate through every pixel of the VRAM by going
//                   through all the columns in a row for every row.
// Data Structures:  None
//
// Registers Changed:r4, r5, r6, r10, r11, r12, r2
// Stack Depth:      1 word
//
// Author:           Maitreyi Ashok
// Last Modified:    05/28/17   Maitreyi Ashok  Initial revision
//                   05/31/17   Maitreyi Ashok  Fixed implementation to go through
//                                              all pixels of display
//                   06/29/17   Maitreyi Ashok  Updated comments

.global clear_display
.align 4
.type clear_display, @function

clear_display:
InitVars:
    mov     r4, zero                // Start at top left corner of display
    mov     r5, zero
    movia   r6, PIXEL_WHITE         // To clear display, show white pixel at
                                    // all positions
    PUSH    ra                      // Store return address from this function
                                    // so it can be restored after calling other
                                    // functions
CheckRow:
    cmpgei  r10, r5, NUM_ROWS_VRAM  // Once go through all the rows in the VRAM
    bne     r10, zero, EndClear     // done clearing display
CheckCol:
    cmpgei  r10, r4, NUM_COLS_VRAM  // Once go through all columns of a row in
    bne     r10, zero, DoneRow      // VRAM need to move to next row
ClearPixel:
    call    plot_pixel              // Otherwise, valid row & column number to
                                    // clear a pixel at by plotting a white pixel
    addi    r4, r4, 1               // Move to next column and check if it
    jmpi    CheckCol                // is in valid range of VRAM
DoneRow:
    mov     r5, zero                // When finished a row, move to next row
```

```
    addi    r5, r5, 1               // and first column within that row
    jmpi    CheckRow                // Check if next row is within valid range of
                                    // VRAM
EndClear:
    POP_VAL ra                      // Restore the return address into caller
    ret                             // function


// plot_pixel
//
// Description:      This function plots a pixel of a specified color at the
//                   given row and column of the display. The row and column
//                   are specified as an index in the display. Row 0 and column 0
//                   specify the top left corner of the display. Increasing the
//                   row number moves down in the display, and increasing the
//                   column number moves right in the display. The pixel color
//                   is given by an 8 bit value that represents the RGB color
//                   code for the color. The 8 bits of color includes 3 bits of red,
//                   3 bits of green, and 2 bits of blue data. These bits are
//                   the most significant bits of the 8 bits representing each
//                   of the primary colors. The non-controllable bits are always
//                   1s.
//
// Operation:        This function plots a pixel of the given color in the given
//                   position on the display. This is done by using the get_addr
//                   function to find the address in the VRAM of the given row
//                   and column number of the display. Then, a direct write
//                   memory is done to write the pixel byte to that location in
//                   the VRAM. This will then be output from the SAM through
//                   the serial output cycle, and displayed on the LCD.
//
// Arguments:        x [r4] - Column number of the pixel to plot
//                   y [r5] - Row number of pixel to plot
//                   color [r6] - Pixel color to plot
// Return Value:     None
//
// Local Variables:  vram_addr [r2] - address in VRAM to write to
// Shared Variables: None
// Global Variables: None
//
// Input:            None
// Output:           A pixel is plotted on display at specified position with
//                   the specified color
//
// Error Handling:   None
//
// Algorithms:       None
// Data Structures:  None
//
// Registers Changed:r2, r11, r12
// Stack Depth:      1 words
//
// Author:           Maitreyi Ashok
// Last Modified:    05/28/17   Maitreyi Ashok  Initial revision
//                   06/01/17   Maitreyi Ashok  Changed address calculation to
//                                              use get_addr
//                   06/29/17   Maitreyi Ashok  Updated comments

.global plot_pixel
.align 4
.type plot_pixel, @function

plot_pixel:
GetAddr:
```

```
    PUSH    ra                      // Save return address before calling other
                                    // functions
    call    get_addr                // Get address corresponding to (x, y) passed
                                    // in for position of pixel to plot
    stb     r6, 0(r2)               // Store the pixel color passed in at that
                                    // address
    POP_VAL ra                      // Restore return address into caller function
    ret
```

```
/*****************************************************************************/
/*                                                                           */
/*                                  DISP.H                                   */
/*                            Display Definitions                            */
/*                               Include File                                */
/*                        Digital Oscilloscope Project                       */
/*                                  EE/CS 52                                 */
/*                                                                           */
/*****************************************************************************/

/*
 * This file contains the constants for the plotting pixels and clearing the
 * the display. These are used to iterate through all the pixels of the VRAM
 * as well as fix a hardware issue with the VRAM and display controller through
 * software.
 *
 * Revision History:
 *  5/28/17  Maitreyi Ashok  Initial Revision
 *  6/01/17  Maitreyi Ashok  Added constants for software fix of display
 *  6/29/17  Maitreyi Ashok  Updated comments
*/

#ifndef DISP_H_
#define DISP_H_


#define NUM_ROWS_VRAM   512;        // Number of rows in VRAM
#define NUM_COLS_VRAM   512;        // Number of columns in VRAM
#define COL_FIX         1;          // Shift all columns to left by this
                                    // amount when displaying
#define ROW_FIX         1;          // Shift row for first column up by this
                                    // amount when displaying
#endif /* DISP_H_ */
```

```
/************************************************************************/
/*                                                                      */
/*                              TRACUTIL                                */
/*                       Trace Utility Functions                        */
/*                     Digital Oscilloscope Project                     */
/*                              EE/CS 52                                */
/*                                                                      */
/************************************************************************/

/*
   This file contains the utility functions for handling traces (capturing
   and displaying data) for the Digital Oscilloscope project.  The functions
   included are:
      clear_saved_areas  - clear all the save areas
      do_trace           - start a trace
      init_trace         - initialize the trace routines
      plot_trace         - plot a trace (sampled data)
      restore_menu_trace - restore the saved area under the menus
      restore_trace      - restore the saved area of a trace
      set_display_scale  - set the type of displayed scale (and display it)
      set_mode           - set the triggering mode
      set_save_area      - determine an area of a trace to save
      set_trace_size     - set the number of samples in a trace
      trace_done         - inform this module that a trace has been completed
      trace_rdy          - determine if system is ready to start another trace
      trace_rearm        - re-enable tracing (in one-shot triggering mode)

   The local functions included are:
      none

   The locally global variable definitions included are:
      cur_scale    - current scale type
      sample_size  - the size of the sample for the trace
      sampling     - currently doing a sample
      saved_area   - saved trace under a specified area
      saved_axis_x - saved trace under the x lines (axes or grid)
      saved_axis_y - saved trace under the y lines (axes or grid)
      saved_menu   - saved trace under the menu
      saved_pos_x  - starting position (x coorindate) of area to save
      saved_pos_y  - starting position (y coorindate) of area to save
      saved_end_x  - ending position (x coorindate) of area to save
      saved_end_y  - ending position (y coorindate) of area to save
      trace_status - whether or not ready to start another trace


   Revision History
      3/8/94   Glen George       Initial revision.
      3/13/94  Glen George       Updated comments.
      3/13/94  Glen George       Fixed inversion of signal in plot_trace.
      3/13/94  Glen George       Added sampling flag and changed the functions
                                 init_trace, do_trace and trace_done to update
                                 the flag.  Also the function trace_rdy now
                                 uses it.  The function set_mode was updated
                                 to always say a trace is ready for normal
                                 triggering.
      3/13/94  Glen George       Fixed bug in trace restoring due to operator
                                 misuse (&& instead of &) in the functions
                                 set_axes, restore_menu_trace, and
                                 restore_trace.
      3/13/94  Glen George       Fixed bug in trace restoring due to the clear
                                 function (clear_saved_areas) not clearing all
                                 of the menu area.
      3/13/94  Glen George       Fixed comparison bug when saving traces in
                       plot_trace.
```

```
      3/13/94  Glen George       Changed name of set_axes to set_display_scale
                                 and the name of axes_state to cur_scale to
                                 more accurately reflect the function/variable
                                 use (especially if add scale display types).
      3/17/97  Glen George       Updated comments.
      3/17/97  Glen George       Changed set_display_scale to use plot_hline
                                 and plot_vline functions to output axes.
      5/3/06   Glen George       Updated formatting.
      5/9/06   Glen George       Updated do_trace function to match the new
                                 definition of start_sample().
      5/9/06   Glen George       Removed normal_trg variable, its use is now
                                 handled by the get_trigger_mode() accessor.
      5/9/06   Glen George       Added tick marks to the axes display.
      5/9/06   Glen George       Added ability to display a grid.
      5/27/08  Glen George       Added is_sampling() function to be able to
                                 tell if the system is currently taking a
                   sample.
      5/27/08  Glen George       Changed set_mode() to always turn off the
                                 sampling flag so samples with the old mode
                                 setting are ignored.
      6/3/08   Glen George       Fixed problems with non-power of 2 display
                                 sizes not working.
      6/17/17  Maitreyi AShok    Change color of horizontal grid/axis lines
*/



/* library include files */
  /* none */

/* local include files */
#include  "scopedef.h"
#include  "lcdout.h"
#include  "menu.h"
#include  "menuact.h"
#include  "tracutil.h"




/* locally global variables */

static int  trace_status;   /* ready to start another trace */

static int  sampling;           /* currently sampling data */

static int  sample_size;    /* number of data points in a sample */

static enum scale_type  cur_scale;  /* current display scale type */

/* traces (sampled data) saved under the axes */
static unsigned char  saved_axis_x[2 * Y_TICK_CNT + 1][PLOT_SIZE_X/8];  /* saved trace under x
lines */
static unsigned char  saved_axis_y[2 * X_TICK_CNT + 1][PLOT_SIZE_Y/8];  /* saved trace under y
lines */

/* traces (sampled data) saved under the menu */
static unsigned char  saved_menu[MENU_SIZE_Y][(MENU_SIZE_X + 7)/8];

/* traces (sampled data) saved under any area */
static unsigned char  saved_area[SAVE_SIZE_Y][SAVE_SIZE_X/8]; /* saved trace under any area */
static int        saved_pos_x;    /* starting x position of saved area */
static int        saved_pos_y;    /* starting y position of saved area */
static int        saved_end_x;    /* ending x position of saved area */
```

```
static int        saved_end_y;    /* ending y position of saved area */




/*
   init_trace

   Description:      This function initializes all of the locally global
                     variables used by these routines.  The saved areas are
             set to non-existant with cleared saved data.  Normal
             normal triggering is set, the system is ready for a
             trace, the scale is turned off and the sample size is set
             to the screen size.

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: trace_status - set to TRUE.
             sampling     - set to FALSE.
             cur_scale    - set to SCALE_NONE (no displayed scale).
             sample_size  - set to screen size (SIZE_X).
             saved_axis_x - cleared.
             saved_axis_y - cleared.
             saved_menu   - cleared.
             saved_area   - cleared.
             saved_pos_x  - set to off-screen.
             saved_pos_y  - set to off-screen.
             saved_end_x  - set to off-screen.
             saved_end_y  - set to off-screen.

   Author:           Glen George
   Last Modified:    May 9, 2006

*/

void  init_trace()
{
    /* variables */
      /* none */



    /* initialize system status variables */

    /* ready for a trace */
    trace_status = TRUE;

    /* not currently sampling data */
    sampling = FALSE;

    /* turn off the displayed scale */
    cur_scale = SCALE_NONE;

    /* sample size is the screen size */
    sample_size = SIZE_X;
```

```
    /* clear save areas */
    clear_saved_areas();

    /* also clear the general saved area location variables (off-screen) */
    saved_pos_x = SIZE_X + 1;
    saved_pos_y = SIZE_Y + 1;
    saved_end_x = SIZE_X + 1;
    saved_end_y = SIZE_Y + 1;


    /* done initializing, return */
    return;

}




/*
   set_mode

   Description:      This function sets the locally global triggering mode
                     based on the passed value (one of the possible enumerated
             values).  The triggering mode is used to determine when
             the system is ready for another trace.  The sampling flag
                     is also reset so a new sample will be started (if that is
                     appropriate).

   Arguments:        trigger_mode (enum trigger_type) - the mode with which to
                            set the triggering.
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: sampling     - set to FALSE to turn off sampling
                  trace_status - set to TRUE if not one-shot triggering.

   Author:           Glen George
   Last Modified:    May 27, 2008

*/

void  set_mode(enum trigger_type trigger_mode)
{
    /* variables */
      /* none */



    /* if not one-shot triggering - ready for trace too */
    trace_status = (trigger_mode != ONESHOT_TRIGGER);


    /* turn off the sampling flag so will start a new sample */
    sampling = FALSE;
```

```
    /* all done, return */
    return;

}




/*
   is_sampling

   Description:      This function determines whether the system is currently
                     taking a sample or not.  This is just the value of the
             sampling flag.

   Arguments:        None.
   Return Value:     (int) - the current sampling status (TRUE if currently
             trying to take a sample, FALSE otherwise).

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: sampling - determines if taking a sample or not.

   Author:           Glen George
   Last Modified:    May 27, 2008

*/

int  is_sampling()
{
    /* variables */
      /* none */



    /* currently sampling if sampling flag is set */
    return  sampling;

}




/*
   trace_rdy

   Description:      This function determines whether the system is ready to
                     start another trace.  This is determined by whether or
             not the system is still sampling (sampling flag) and if
             it is ready for another trace (trace_status flag).

   Arguments:        None.
   Return Value:     (int) - the current trace status (TRUE if ready to do
             another trace, FALSE otherwise).

   Input:            None.
   Output:           None.
```

```
   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: sampling     - determines if ready for another trace.
             trace_status - determines if ready for another trace.

   Author:           Glen George
   Last Modified:    Mar. 13, 1994

*/

int  trace_rdy()
{
    /* variables */
      /* none */



    /* ready for another trace if not sampling and trace is ready */
    return  (!sampling && trace_status);

}




/*
   trace_done

   Description:      This function is called to indicate a trace has been
                     completed.  If in normal triggering mode this means the
             system is ready for another trace.

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: trace_status - may be set to TRUE.
             sampling     - set to FALSE.

   Author:           Glen George
   Last Modified:    May 9, 2006

*/

void  trace_done()
{
    /* variables */
      /* none */



    /* done with a trace - if retriggering, ready for another one */
    if (get_trigger_mode() != ONESHOT_TRIGGER)
```

```
        /* in a retriggering mode - set trace_status to TRUE (ready) */
    trace_status = TRUE;

    /* no longer sampling data */
    sampling = FALSE;


    /* done so return */
    return;

}




/*
   trace_rearm

   Description:      This function is called to rearm the trace.  It sets the
                     trace status to ready (TRUE).  It is used to rearm the
             trigger in one-shot mode.

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: trace_status - set to TRUE.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

void  trace_rearm()
{
    /* variables */
      /* none */



    /* rearm the trace - set status to ready (TRUE) */
    trace_status = TRUE;


    /* all done - return */
    return;

}




/*
   set_trace_size

   Description:      This function sets the locally global sample size to the
                     passed value.  This is used to scale the data when
```

```
            plotting a trace.

   Arguments:        size (int) - the trace sample size.
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: sample_size - set to the passed value.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

void  set_trace_size(int size)
{
    /* variables */
      /* none */



    /* set the locally global sample size */
    sample_size = size;


    /* all done, return */
    return;

}




/*
   set_display_scale

   Description:      This function sets the displayed scale type to the passed
             argument.  If the scale is turned on, it draws it.  If it
             is turned off (SCALE_NONE), it restores the saved trace
             under the scale.  Scales can be axes with tick marks
                     (SCALE_AXES) or a grid (SCALE_GRID).

   Arguments:        scale (scale_type) - new scale type.
   Return Value:     None.

   Input:            None.
   Output:           Either a scale is output or the trace under the old scale
             is restored.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: cur_scale    - set to the passed value.
                     saved_axis_x - used to restore trace data under x-axis.
                     saved_axis_y - used to restore trace data under y-axis.
```

```
   Author:           Maitreyi Ashok
   Last Modified:    June 17, 2017

*/

void  set_display_scale(enum scale_type scale)
{
    /* variables */
    int  p;             /* x or y coordinate */

    int  i;     /* loop indices */
    int  j;



    /* whenever change scale type, need to clear out previous scale */
    /* unnecessary if going to SCALE_GRID or from SCALE_NONE or not changing the scale */
    if ((scale != SCALE_GRID) && (cur_scale != SCALE_NONE) && (scale != cur_scale))  {

        /* need to restore the trace under the lines (tick, grid, or axis) */

    /* go through all points on horizontal lines */
    for (j = -Y_TICK_CNT; j <= Y_TICK_CNT; j++)  {

        /* get y position of the line */
        p = X_AXIS_POS + j * Y_TICK_SIZE;
        /* make sure it is in range */
        if (p >= PLOT_SIZE_Y)
            p = PLOT_SIZE_Y - 1;
        if (p < 0)
            p = 0;

        /* look at entire horizontal line */
        for (i = 0; i < PLOT_SIZE_X; i++)  {
            /* check if this point is on or off (need to look at bits) */
        if ((saved_axis_x[j + Y_TICK_CNT][i / 8] & (0x80 >> (i % 8))) == 0)
            /* saved pixel is off */
            plot_pixel(i, p, PIXEL_WHITE);
        else
            /* saved pixel is on */
            plot_pixel(i, p, PIXEL_BLACK);
        }
    }

    /* go through all points on vertical lines */
    for (j = -X_TICK_CNT; j <= X_TICK_CNT; j++)  {

        /* get x position of the line */
        p = Y_AXIS_POS + j * X_TICK_SIZE;
        /* make sure it is in range */
        if (p >= PLOT_SIZE_X)
            p = PLOT_SIZE_X - 1;
        if (p < 0)
            p = 0;

        /* look at entire vertical line */
        for (i = 0; i < PLOT_SIZE_Y; i++)  {
            /* check if this point is on or off (need to look at bits) */
        if ((saved_axis_y[j + X_TICK_CNT][i / 8] & (0x80 >> (i % 8))) == 0)
            /* saved pixel is off */
            plot_pixel(p, i, PIXEL_WHITE);
        else
            /* saved pixel is on */
            plot_pixel(p, i, PIXEL_BLACK);
```

```
    }
}
}


/* now handle the scale type appropriately */
switch (scale)  {

    case SCALE_AXES:    /* axes for the scale */
    case SCALE_GRID:    /* grid for the scale */

            /* draw x lines (grid or tick marks) */
        for (i = -Y_TICK_CNT; i <= Y_TICK_CNT; i++)  {

        /* get y position of the line */
        p = X_AXIS_POS + i * Y_TICK_SIZE;
        /* make sure it is in range */
        if (p >= PLOT_SIZE_Y)
            p = PLOT_SIZE_Y - 1;
        if (p < 0)
            p = 0;

        /* should we draw a grid, an axis, or a tick mark */
        if (scale == SCALE_GRID)
            /* drawing a grid line */
                plot_hline(X_GRID_START, p, (X_GRID_END - X_GRID_START), PIXEL_BLUE);
        else if (i == 0)
            /* drawing the x axis */
                plot_hline(X_AXIS_START, p, (X_AXIS_END - X_AXIS_START), PIXEL_BLUE);
        else
            /* must be drawing a tick mark */
                plot_hline((Y_AXIS_POS - (TICK_LEN / 2)), p, TICK_LEN, PIXEL_BLUE);
        }

            /* draw y lines (grid or tick marks) */
        for (i = -X_TICK_CNT; i <= X_TICK_CNT; i++)  {

        /* get x position of the line */
        p = Y_AXIS_POS + i * X_TICK_SIZE;
        /* make sure it is in range */
        if (p >= PLOT_SIZE_X)
            p = PLOT_SIZE_X - 1;
            if (p < 0)
            p = 0;

        /* should we draw a grid, an axis, or a tick mark */
        if (scale == SCALE_GRID)
            /* drawing a grid line */
                plot_vline(p, Y_GRID_START, (Y_GRID_END - Y_GRID_START));
        else if (i == 0)
            /* drawing the y axis */
                plot_vline(p, Y_AXIS_START, (Y_AXIS_END - Y_AXIS_START));
        else
            /* must be drawing a tick mark */
                plot_vline(p, (X_AXIS_POS - (TICK_LEN / 2)), TICK_LEN);
        }

        /* done with the axes */
        break;

    case SCALE_NONE:    /* there is no scale */
        /* already restored plot so nothing to do */
        break;
```

```
    }


    /* now remember the new (now current) scale type */
    cur_scale = scale;


    /* scale is taken care of, return */
    return;

}




/*
   clear_saved_areas

   Description:      This function clears all the saved areas (for saving the
                     trace under the axes, menus, and general areas).

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: saved_axis_x - cleared.
             saved_axis_y - cleared.
             saved_menu   - cleared.
             saved_area   - cleared.

   Author:           Glen George
   Last Modified:    May 9, 2006

*/

void  clear_saved_areas()
{
    /* variables */
    int  i;     /* loop indices */
    int  j;



    /* clear x-axis and y-axis save areas */
    for (j = 0; j <= (2 * Y_TICK_CNT); j++)
        for (i = 0; i < (SIZE_X / 8); i++)
            saved_axis_x[j][i] = 0;
    for (j = 0; j <= (2 * X_TICK_CNT); j++)
        for (i = 0; i < (SIZE_Y / 8); i++)
            saved_axis_y[j][i] = 0;

    /* clear the menu save ares */
    for (i = 0; i < MENU_SIZE_Y; i++)
        for (j = 0; j < ((MENU_SIZE_X + 7) / 8); j++)
        saved_menu[i][j] = 0;

    /* clear general save area */
```

```
    for (i = 0; i < SAVE_SIZE_Y; i++)
        for (j = 0; j < (SAVE_SIZE_X / 8); j++)
        saved_area[i][j] = 0;


    /* done clearing the saved areas - return */
    return;

}




/*
   restore_menu_trace

   Description:      This function restores the trace under the menu when the
                     menus are turned off.  (The trace was previously saved.)

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           The trace under the menu is restored to the LCD screen.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: saved_menu - used to restore trace data under the menu.

   Author:           Glen George
   Last Modified:    Mar. 13, 1994

*/

void  restore_menu_trace()
{
    /* variables */
    int  bit_position;  /* position of bit to restore (in saved data) */
    int  bit_offset;    /* offset (in bytes) of bit within saved row */

    int  x;     /* loop indices */
    int  y;



    /* loop, restoring the trace under the menu */
    for (y = MENU_UL_Y; y < (MENU_UL_Y + MENU_SIZE_Y); y++)  {

        /* starting a row - initialize bit position */
    bit_position = 0x80;    /* start at high-order bit in the byte */
    bit_offset = 0;     /* first byte of the row */

        for (x = MENU_UL_X; x < (MENU_UL_X + MENU_SIZE_X); x++)  {

        /* check if this point is on or off (need to look at bits) */
        if ((saved_menu[y - MENU_UL_Y][bit_offset] & bit_position) == 0)
            /* saved pixel is off */
        plot_pixel(x, y, PIXEL_WHITE);
        else
            /* saved pixel is on */
        plot_pixel(x, y, PIXEL_BLACK);
```

```
        /* move to the next bit position */
        bit_position >>= 1;
        /* check if moving to next byte */
        if (bit_position == 0)  {
            /* now on high bit of next byte */
        bit_position = 0x80;
        bit_offset++;
        }
        }
    }


    /* restored menu area - return */
    return;

}




/*
   set_save_area

   Description:      This function sets the position and size of the area to
                     be saved when traces are drawn.  It also clears any data
             currently saved.

   Arguments:        pos_x (int)  - x position of upper left corner of the
                    saved area.
             pos_y (int)  - y position of upper left corner of the
                        saved area.
             size_x (int) - horizontal size of the saved area.
             size_y (int) - vertical size of the saved area.
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: saved_area  - cleared.
             saved_pos_x - set to passed value.
             saved_pos_y - set to passed value.
             saved_end_x - computed from passed values.
             saved_end_y - computed from passed values.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

void  set_save_area(int pos_x, int pos_y, int size_x, int size_y)
{
    /* variables */
    int  x;     /* loop indices */
    int  y;



    /* just setup all the locally global variables from the passed values */
```

```
    saved_pos_x = pos_x;
    saved_pos_y = pos_y;
    saved_end_x = pos_x + size_x;
    saved_end_y = pos_y + size_y;


    /* clear the save area */
    for (y = 0; y < SAVE_SIZE_Y; y++)  {
        for (x = 0; x < (SAVE_SIZE_X / 8); x++)  {
        saved_area[y][x] = 0;
        }
    }


    /* setup the saved area - return */
    return;

}




/*
   restore_trace

   Description:      This function restores the trace under the set saved
                     area.  (The area was previously set and the trace was
             previously saved.)

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           The trace under the saved ares is restored to the LCD.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: saved_area  - used to restore trace data.
             saved_pos_x - gives starting x position of saved area.
             saved_pos_y - gives starting y position of saved area.
             saved_end_x - gives ending x position of saved area.
             saved_end_y - gives ending y position of saved area.

   Author:           Glen George
   Last Modified:    Mar. 13, 1994

*/

void  restore_trace()
{
    /* variables */
    int  bit_position;  /* position of bit to restore (in saved data) */
    int  bit_offset;    /* offset (in bytes) of bit within saved row */

    int  x;     /* loop indices */
    int  y;



    /* loop, restoring the saved trace */
    for (y = saved_pos_y; y < saved_end_y; y++)  {
```

```
        /* starting a row - initialize bit position */
    bit_position = 0x80;    /* start at high-order bit in the byte */
    bit_offset = 0;     /* first byte of the row */

        for (x = saved_pos_x; x < saved_end_x; x++)  {

        /* check if this point is on or off (need to look at bits) */
        if ((saved_area[y - saved_pos_y][bit_offset] & bit_position) == 0)
            /* saved pixel is off */
        plot_pixel(x, y, PIXEL_WHITE);
        else
            /* saved pixel is on */
        plot_pixel(x, y, PIXEL_BLACK);

        /* move to the next bit position */
        bit_position >>= 1;
        /* check if moving to next byte */
        if (bit_position == 0)  {
            /* now on high bit of next byte */
        bit_position = 0x80;
        bit_offset++;
        }
        }
    }


    /* restored the saved area - return */
    return;

}




/*
   do_trace

   Description:      This function starts a trace.  It starts the hardware
                     sampling data (via a function call) and sets the trace
             ready flag (trace_status) to FALSE and the sampling flag
             (sampling) to TRUE.

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: trace_status - set to FALSE (not ready for another trace).
             sampling     - set to TRUE (doing a sample now).

   Author:           Glen George
   Last Modified:    Mar. 13, 1994

*/

void  do_trace()
{
```

```
    /* variables */
      /* none */



    /* start up the trace */
    /* indicate whether using automatic triggering or not */
    start_sample(get_trigger_mode() == AUTO_TRIGGER);

    /* now not ready for another trace (currently doing one) */
    trace_status = FALSE;

    /* and are currently sampling data */
    sampling = TRUE;


    /* trace is going, return */
    return;

}




/*
   plot_trace

   Description:      This function plots the passed trace.  The trace is
                     assumed to contain sample_size points of sampled data.
             Any points falling within any of the save areas are also
             saved by this routine.  The data is also scaled to be
             within the range of the entire screen.

   Arguments:        sample (unsigned char far *) - sample to plot.
   Return Value:     None.

   Input:            None.
   Output:           The sample is plotted on the screen.

   Error Handling:   None.

   Algorithms:       If there are more sample points than screen width the
             sample is plotted with multiple points per horizontal
             position.
   Data Structures:  None.

   Global Variables: cur_scale    - determines type of scale to plot.
             sample_size  - determines size of passed sample.
             saved_axis_x - stores trace under x-axis.
             saved_axis_y - stores trace under y-axis.
             saved_menu   - stores trace under the menu.
             saved_area   - stores trace under the saved area.
             saved_pos_x  - determines location of saved area.
             saved_pos_y  - determines location of saved area.
             saved_end_x  - determines location of saved area.
             saved_end_y  - determines location of saved area.

   Author:           Glen George
   Last Modified:    May 9, 2006

*/

void  plot_trace(unsigned char *sample)
{
```

```
    /* variables */
    int  x = 0;             /* current x position to plot */
    int  x_pos = (PLOT_SIZE_X / 2); /* "fine" x position for multiple point plotting */

    int  y;             /* y position of point to plot */

    int  p;                             /* an x or y coordinate */

    int  i;             /* loop indices */
    int  j;



    /* first, clear the display to get rid of old plots */
    clear_display();

    /* clear the saved areas too */
    clear_saved_areas();

    /* re-display the menu (if it was on) */
    refresh_menu();


    /* plot the sample */
    for (i = 0; i < sample_size; i++)  {

        /* determine y position of point (note: screen coordinates invert) */
    y = (PLOT_SIZE_Y - 1) - ((sample[i] * (PLOT_SIZE_Y - 1)) / 255);

        /* plot this point */
    plot_pixel(x, y, PIXEL_BLACK);


    /* check if the point is in a save area */

    /* check if in the menu area */
    if ((x >= MENU_UL_X) && (x < (MENU_UL_X + MENU_SIZE_X)) &&
        (y >= MENU_UL_Y) && (y < (MENU_UL_Y + MENU_SIZE_Y)))
        /* point is in the menu area - save it */
        saved_menu[y - MENU_UL_Y][(x - MENU_UL_X)/8] |= (0x80 >> ((x - MENU_UL_X) % 8));

    /* check if in the saved area */
    if ((x >= saved_pos_x) && (x <= saved_end_x) && (y >= saved_pos_y) && (y <= saved_end_y))
        /* point is in the save area - save it */
        saved_area[y - saved_pos_y][(x - saved_pos_x)/8] |= (0x80 >> ((x - saved_pos_x) % 8));

    /* check if on a grid line */
    /* go through all the horizontal lines */
    for (j = -Y_TICK_CNT; j <= Y_TICK_CNT; j++)  {

        /* get y position of the line */
        p = X_AXIS_POS + j * Y_TICK_SIZE;
        /* make sure it is in range */
        if (p >= PLOT_SIZE_Y)
            p = PLOT_SIZE_Y - 1;
        if (p < 0)
            p = 0;

        /* if the point is on this line, save it */
        if (y == p)
        saved_axis_x[j + Y_TICK_CNT][x / 8] |= (0x80 >> (x % 8));
    }

    /* go through all the vertical lines */
```

```
    for (j = -X_TICK_CNT; j <= X_TICK_CNT; j++)  {

        /* get x position of the line */
        p = Y_AXIS_POS + j * X_TICK_SIZE;
        /* make sure it is in range */
        if (p >= PLOT_SIZE_X)
            p = PLOT_SIZE_X - 1;
        if (p < 0)
            p = 0;

        /* if the point is on this line, save it */
        if (x == p)
        saved_axis_y[j + X_TICK_CNT][y / 8] |= (0x80 >> (y % 8));
    }


    /* update x position */
    x_pos += PLOT_SIZE_X;
    /* check if at next horizontal position */
    if (x_pos >= sample_size)  {
        /* at next position - update positions */
        x++;
        x_pos -= sample_size;
    }
    }


    /* finally, output the scale if need be */
    set_display_scale(cur_scale);


    /* done with plot, return */
    return;

}
```

```
/*****************************************************************************/
/*                                                                           */
/*                                TRACUTIL.H                                 */
/*                          Trace Utility Functions                          */
/*                                Include File                               */
/*                        Digital Oscilloscope Project                       */
/*                                 EE/CS 52                                  */
/*                                                                           */
/*****************************************************************************/

/*
   This file contains the constants and function prototypes for the trace
   utility functions (defined in tracutil.c) for the Digital Oscilloscope
   project.


   Revision History:
      3/8/94   Glen George       Initial revision.
      3/13/94  Glen George       Updated comments.
      3/13/94  Glen George       Changed name of set_axes function to
                                 set_display_scale.
      5/9/06   Glen George       Added the constants for grids and tick marks.
      5/27/08  Glen George       Added is_sampling() function to be able to
                                 tell if the system is currently taking a
                                 sample.
      6/3/08   Glen George       Removed Y_SCALE_FACTOR - no longer used to
                                 fix problems with non-power of 2 display
                                 sizes.
*/



#ifndef  __TRACUTIL_H__
    #define  __TRACUTIL_H__


/* library include files */
  /* none */

/* local include files */
#include  "interfac.h"
#include  "menuact.h"




/* constants */

/* plot size */
#define  PLOT_SIZE_X    SIZE_X      /* plot takes entire screen width */
#define  PLOT_SIZE_Y    SIZE_Y      /* plot takes entire screen height */

/* axes position and size */
#define  X_AXIS_START   0                  /* starting x position of x-axis */
#define  X_AXIS_END     (PLOT_SIZE_X - 1)  /* ending x position of x-axis */
#define  X_AXIS_POS (PLOT_SIZE_Y / 2)      /* y position of x-axis */
#define  Y_AXIS_START   0                  /* starting y position of y-axis */
#define  Y_AXIS_END     (PLOT_SIZE_Y - 1)  /* ending y position of y-axis */
#define  Y_AXIS_POS (PLOT_SIZE_X / 2)      /* x position of y-axis */

/* tick mark and grid constants */
#define  TICK_LEN       5                  /* length of axis tick mark */
/* tick mark counts are for a single quadrant, thus total number of tick */
/* marks or grids is twice this number */
```

```
#define  X_TICK_CNT     5                  /* always 5 tick marks on x axis */
#define  X_TICK_SIZE    (PLOT_SIZE_X / (2 * X_TICK_CNT))   /* distance between tick marks */
#define  Y_TICK_SIZE    X_TICK_SIZE        /* same size as x */
#define  Y_TICK_CNT     (PLOT_SIZE_Y / (2 * Y_TICK_SIZE))  /* number of y tick marks */
#define  X_GRID_START   0                  /* starting x position of x grid */
#define  X_GRID_END     (PLOT_SIZE_X - 1)  /* ending x position of x grid */
#define  Y_GRID_START   0                  /* starting y position of y-axis */
#define  Y_GRID_END     (PLOT_SIZE_Y - 1)  /* ending y position of y-axis */

/* maximum size of the save area (in pixels) */
#define  SAVE_SIZE_X    120 /* maximum width */
#define  SAVE_SIZE_Y    16  /* maximum height */




/* structures, unions, and typedefs */
    /* none */




/* function declarations */

/* initialize the trace utility routines */
void  init_trace(void);

/* trace status functions */
void  set_mode(enum trigger_type);  /* set the triggering mode */
int   is_sampling(void);            /* currently trying to take a sample */
int   trace_rdy(void);              /* determine if ready to start a trace */
void  trace_done(void);             /* signal a trace has been completed */
void  trace_rearm(void);            /* re-enable tracing */

/* trace save area functions */
void  clear_saved_areas(void);            /* clears all saved areas */
void  restore_menu_trace(void);           /* restore the trace under menus */
void  set_save_area(int, int, int, int);  /* set an area of a trace to save */
void  restore_trace(void);                /* restore saved area of a trace */

/* set the scale type */
void  set_display_scale(enum scale_type);

/* setup and plot a trace */
void  set_trace_size(int);              /* set the number of samples in a trace */
void  do_trace(void);                   /* start a trace */
void  plot_trace(unsigned char *);      /* plot a trace (sampled data) */


#endif
```

```
/*****************************************************************************/
/*                                                                           */
/*                                  LCDOUT                                   */
/*                           LCD Output Functions                            */
/*                        Digital Oscilloscope Project                       */
/*                                  EE/CS 52                                 */
/*                                                                           */
/*****************************************************************************/

/*
   This file contains the functions for doing output to the LCD screen for the
   Digital Oscilloscope project.  The functions included are:
      clear_region - clear a region of the display
      plot_char    - output a character
      plot_hline   - draw a horizontal line
      plot_string  - output a string
      plot_vline   - draw a vertical line

   The local functions included are:
      none

   The locally global variable definitions included are:
      none


   Revision History
      3/8/94   Glen George       Initial revision.
      3/13/94  Glen George       Updated comments.
      3/13/94  Glen George       Simplified code in plot_string function.
      3/17/97  Glen George       Updated comments.
      3/17/97  Glen George       Change plot_char() and plot_string() to use
                                 enum char_style instead of an int value.
      5/27/98  Glen George       Change plot_char() to explicitly declare the
                                 size of the external array to avoid linker
                                 errors.
*/



/* library include files */
  /* none */

/* local include files */
#include  "interfac.h"
#include  "scopedef.h"
#include  "lcdout.h"




/*
   clear_region

   Description:     This function clears the passed region of the display.
                    The region is described by its upper left corner pixel
                    coordinate and the size (in pixels) in each dimension.

   Arguments:       x_ul (int)   - x coordinate of upper left corner of the
                    region to be cleared.
                    y_ul (int)   - y coordinate of upper left corner of the
                    region to be cleared.
                    x_size (int) - horizontal size of the region.
                    y_size (int) - vertical size of the region.
   Return Value:     None.
```

```
   Input:           None.
   Output:          A portion of the screen is cleared (set to PIXEL_WHITE).

   Error Handling:  No error checking is done on the coordinates.

   Algorithms:      None.
   Data Structures: None.

   Global Variables:None.

   Author:          Glen George
   Last Modified:   Mar. 8, 1994

*/

void  clear_region(int x_ul, int y_ul, int x_size, int y_size)
{
    /* variables */
    int  x;     /* x coordinate to clear */
    int  y;     /* y coordinate to clear */



    /* loop, clearing the display */
    for (x = x_ul; x < (x_ul + x_size); x++)  {
        for (y = y_ul; y < (y_ul + y_size); y++)  {

        /* clear this pixel */
        plot_pixel(x, y, PIXEL_WHITE);
        }
    }


    /* done clearing the display region - return */
    return;

}




/*
   plot_hline

   Description:      This function draws a horizontal line from the passed
                     position for the passed length.  The line is always drawn
                     with the color PIXEL_BLACK.  The position (0,0) is the
             upper left corner of the screen.

   Arguments:        start_x (int) - starting x coordinate of the line.
             start_y (int) - starting y coordinate of the line.
             length (int)  - length of the line (positive for a line
                         to the "right" and negative for a line to
                     the "left").
   Return Value:     None.

   Input:            None.
   Output:           A horizontal line is drawn at the specified position.

   Error Handling:   No error checking is done on the coordinates.

   Algorithms:       None.
   Data Structures:  None.
```

```
   Global Variables: None.

   Author:           Glen George
   Last Modified:    Mar. 7, 1994

*/

void  plot_hline(int start_x, int start_y, int length, char color)
{
    /* variables */
    int  x;     /* x position while plotting */

    int  init_x;    /* starting x position to plot */
    int  end_x;     /* ending x position to plot */



    /* check if a line to the "right" or "left" */
    if (length > 0)  {

        /* line to the "right" - start at start_x, end at start_x + length */
    init_x = start_x;
    end_x = start_x + length;
    }
    else  {

        /* line to the "left" - start at start_x + length, end at start_x */
    init_x = start_x + length;
    end_x = start_x;
    }


    /* loop, outputting points for the line (always draw to the "right") */
    for (x = init_x; x < end_x; x++)
        /* plot a point of the line */
    plot_pixel(x, start_y, color);


    /* done plotting the line - return */
    return;

}




/*
   plot_vline

   Description:      This function draws a vertical line from the passed
                     position for the passed length.  The line is always drawn
                     with the color PIXEL_BLACK.  The position (0,0) is the
             upper left corner of the screen.

   Arguments:        start_x (int) - starting x coordinate of the line.
             start_y (int) - starting y coordinate of the line.
             length (int)  - length of the line (positive for a line
                         going "down" and negative for a line
                     going "up").
   Return Value:     None.

   Input:            None.
   Output:           A vertical line is drawn at the specified position.
```

```
   Error Handling:   No error checking is done on the coordinates.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: None.

   Author:           Glen George
   Last Modified:    Mar. 7, 1994

*/

void  plot_vline(int start_x, int start_y, int length)
{
    /* variables */
    int  y;     /* y position while plotting */

    int  init_y;    /* starting y position to plot */
    int  end_y;     /* ending y position to plot */



    /* check if an "up" or "down" line */
    if (length > 0)  {

        /* line going "down" - start at start_y, end at start_y + length */
    init_y = start_y;
    end_y = start_y + length;
    }
    else  {

        /* line going "up" - start at start_y + length, end at start_y */
    init_y = start_y + length;
    end_y = start_y;
    }


    /* loop, outputting points for the line (always draw "down") */
    for (y = init_y; y < end_y; y++)
        /* plot a point of the line */
    plot_pixel(start_x, y, PIXEL_BLACK);


    /* done plotting the line - return */
    return;

}




/*
   plot_char

   Description:      This function outputs the passed character to the LCD
                     screen at passed location.  The passed location is given
                     as a character position with (0,0) being the upper left
             corner of the screen.  The character can be drawn in
             "normal video" (black on white) or "reverse video" (white
             on black).

   Arguments:        pos_x (int)             - x coordinate (in character
                              cells) of the character.
```

```
            pos_y (int)             - y coordinate (in character
                                cells) of the character.
            c (char)                - the character to plot.
            style (enum char_style) - style with which to plot the
                                  character (NORMAL or REVERSE).
   Return Value:     None.

   Input:            None.
   Output:           A character is output to the LCD screen.

   Error Handling:   No error checking is done on the coordinates or the
            character (to ensure there is a bit pattern for it).

   Algorithms:       None.
   Data Structures:  The character bit patterns are stored in an external
            array.

   Global Variables: None.

   Author:           Glen George
   Last Modified:    May 27, 2008

*/

void  plot_char(int pos_x, int pos_y, char c, enum char_style style)
{
    /* variables */

    /* pointer to array of character bit patterns */
    extern const unsigned char   char_patterns[(VERT_SIZE - 1) * 128];

    int  bits;          /* a character bit pattern */

    int  col;       /* column loop index */
    int  row;           /* character row loop index */

    int  x;     /* x pixel position for the character */
    int  y;     /* y pixel position for the character */



    /* setup the pixel positions for the character */
    x = pos_x * HORIZ_SIZE;
    y = pos_y * VERT_SIZE;


    /* loop outputting the bits to the screen */
    for (row = 0; row < VERT_SIZE; row++)  {

        /* get the character bits for this row from the character table */
    if (row == (VERT_SIZE - 1))
        /* last row - blank it */
        bits = 0;
    else
        /* in middle of character, get the row from the bit patterns */
            bits = char_patterns[(c * (VERT_SIZE - 1)) + row];

    /* take care of "normal/reverse video" */
    if (style == REVERSE)
        /* invert the bits for "reverse video" */
        bits = ~bits;

        /* get the bits "in position" (high bit is output first */
    bits <<= (8 - HORIZ_SIZE);
```

```
    /* now output the row of the character, pixel by pixel */
    for (col = 0; col < HORIZ_SIZE; col++)  {

            /* output this pixel in the appropriate color */
        if ((bits & 0x80) == 0)
            /* blank pixel - output in PIXEL_WHITE */
        plot_pixel(x + col, y, PIXEL_WHITE);
        else
            /* black pixel - output in PIXEL_BLACK */
        plot_pixel(x + col, y, PIXEL_BLACK);

        /* shift the next bit into position */
        bits <<= 1;
        }


    /* next row - update the y position */
    y++;
    }


    /* all done, return */
    return;

}




/*
   plot_string

   Description:      This function outputs the passed string to the LCD screen
                     at passed location.  The passed location is given as a
                     character position with (0,0) being the upper left corner
             of the screen.  There is no line wrapping, so the entire
             string must fit on the passed line (pos_y).  The string
             can be drawn in "normal video" (black on white) or
             "reverse video" (white on black).

   Arguments:        pos_x (int)             - x coordinate (in character
                                 cells) of the start of the
                           string.
             pos_y (int)             - y coordinate (in character
                                 cells) of the start of the
                           string.
             s (const char *)        - the string to output.
             style (enum char style) - style with which to plot
                                       characters of the string.
   Return Value:     None.

   Input:            None.
   Output:           A string is output to the LCD screen.

   Error Handling:   No checking is done to insure the string is fully on the
             screen (the x and y coordinates and length of the string
             are not checked).

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: None.
```

```
   Author:           Glen George
   Last Modified:    Mar. 17, 1997

*/

void  plot_string(int pos_x, int pos_y, const char *s, enum char_style style)
{
    /* variables */
      /* none */



    /* loop, outputting characters from string s */
    while (*s != '\0')

        /* output this character and move to the next character and screen position */
    plot_char(pos_x++, pos_y, *s++, style);


    /* all done, return */
    return;

}
```

```
/*****************************************************************************/
/*                                                                           */
/*                                  LCDOUT.H                                 */
/*                            LCD Output Functions                           */
/*                                Include File                               */
/*                          Digital Oscilloscope Project                     */
/*                                  EE/CS  52                                */
/*                                                                           */
/*****************************************************************************/

/*
   This file contains the constants and function prototypes for the LCD output
   functions used in the Digital Oscilloscope project and defined in lcdout.c.


   Revision History:
      3/8/94   Glen George       Initial revision.
      3/13/94  Glen George       Updated comments.
      3/17/97  Glen George       Added enumerated type char_style and updated
                                    function prototypes.
*/




#ifndef  __LCDOUT_H__
    #define  __LCDOUT_H__


/* library include files */
  /* none */

/* local include files */
  /* none */




/* constants */

/* character output styles */

/* size of a character (includes 1 pixel space to the left and below character) */
#define  VERT_SIZE    8         /* vertical size (in pixels -> 7+1) */
#define  HORIZ_SIZE   6         /* horizontal size (in pixels -> 5+1) */




/* structures, unions, and typedefs */

/* character output styles */
enum  char_style  {  NORMAL,    /* "normal video" */
                     REVERSE    /* "reverse video" */
             };




/* function declarations */

void  clear_region(int, int, int, int);       /* clear part of the display */

void  plot_hline(int, int, int, char);        /* draw a horizontal line */
```

```
void  plot_vline(int, int, int);          /* draw a vertical line */

void  plot_char(int, int, char, enum char_style); /* output a character */
void  plot_string(int, int, const char *, enum char_style);  /* output a string */


#endif
```

```
/*****************************************************************************/
/*                                                                           */
/*                                 MENUACT                                   */
/*                            Menu Action Functions                          */
/*                         Digital Oscilloscope Project                      */
/*                                 EE/CS 52                                  */
/*                                                                           */
/*****************************************************************************/

/*
   This file contains the functions for carrying out menu actions for the
   Digital Oscilloscope project.  These functions are invoked when the <Left>
   or <Right> key is pressed for a menu item.  Also included are the functions
   for displaying the current menu option selection.  The functions included
   are:
      display_mode      - display trigger mode
      display_scale     - display the scale type
      display_sweep     - display the sweep rate
      display_trg_delay - display the tigger delay
      display_trg_level - display the trigger level
      display_trg_slope - display the trigger slope
      get_trigger_mode  - get the current trigger mode
      mode_down         - go to the "next" trigger mode
      mode_up           - go to the "previous" trigger mode
      no_display        - nothing to display for option setting
      no_menu_action    - no action to perform for <Left> or <Right> key
      scale_down        - go to the "next" scale type
      scale_up          - go to the "previous" scale type
      set_scale         - set the scale type
      set_sweep         - set the sweep rate
      set_trg_delay     - set the tigger delay
      set_trg_level     - set the trigger level
      set_trg_slope     - set the trigger slope
      set_trigger_mode  - set the trigger mode
      sweep_down        - decrease the sweep rate
      sweep_up          - increase the sweep rate
      trg_delay_down    - decrease the trigger delay
      trg_delay_up      - increase the trigger delay
      trg_level_down    - decrease the trigger level
      trg_level_up      - increase the trigger level
      trg_slope_toggle  - toggle the trigger slope between "+" and "-"

   The local functions included are:
      adjust_trg_delay  - adjust the trigger delay for a new sweep rate
      cvt_num_field     - converts a numeric field value to a string

   The locally global variable definitions included are:
      delay         - current trigger delay
      level         - current trigger level
      scale         - current display scale type
      slope         - current trigger slope
      sweep         - current sweep rate
      sweep_rates   - table of information on possible sweep rates
      trigger_mode  - current triggering mode


   Revision History
      3/8/94   Glen George       Initial revision.
      3/13/94  Glen George       Updated comments.
      3/13/94  Glen George       Changed all arrays of constant strings to be
                     static so compiler generates correct code.
      3/13/94  Glen George       Changed scale to type enum scale_type and
                     output the selection as "None" or "Axes".
                  This will allow for easier future expansion.
```

```
      3/13/94  Glen George       Changed name of set_axes function (in
                    tracutil.c) to set_display_scale.
      3/10/95  Glen George       Changed calculation of displayed trigger
                    level to use constants MIN_TRG_LEVEL_SET and
                  MAX_TRG_LEVEL_SET to get the trigger level
                  range.
      3/17/97  Glen George       Updated comments.
      5/3/06   Glen George       Changed sweep definitions to include new
                    sweep rates of 100 ns, 200 ns, 500 ns, and
                    1 us and updated functions to handle these
                  new rates.
      5/9/06   Glen George       Added new a triggering mode (automatic
                                 triggering) and a new scale (grid) and
                                 updated functions to implement these options.
      5/9/06   Glen George       Added functions for setting the triggering
                                 mode and scale by going up and down the list
                                 of possibilities instead of just toggling
                                 between one of two possibilities (since there
                  are more than two now).
      5/9/06   Glen George       Added accessor function (get_trigger_mode)
                                 to be able to get the current trigger mode.
*/



/* library include files */
  /* none */

/* local include files */
#include  "interfac.h"
#include  "scopedef.h"
#include  "lcdout.h"
#include  "menuact.h"
#include  "tracutil.h"




/* local function declarations */
static void  adjust_trg_delay(int, int);       /* adjust the trigger delay for new sweep */
static void  cvt_num_field(long int, char *);   /* convert a number to a string */




/* locally global variables

/* trace parameters */
static enum trigger_type   trigger_mode; /* current triggering mode */
static enum scale_type     scale;    /* current scale type */
static int         sweep;        /* sweep rate index */
static int         level;    /* current trigger level */
static enum slope_type     slope;    /* current trigger slope */
static long int            delay;    /* current trigger delay */

/* sweep rate information */
static const struct sweep_info  sweep_rates[] =
    { { 10000000L, " 100 ns" },
      {  5000000L, " 200 ns" },
      {  2000000L, " 500 ns" },
      {  1000000L, " 1 \004s  " },
      {   500000L, " 2 \004s  " },
      {   200000L, " 5 \004s  " },
      {   100000L, " 10 \004s " },
```

```
      {    50000L, " 20 \004s " },
      {    20000L, " 50 \004s " },
      {    10000L, " 100 \004s" },
      {     5000L, " 200 \004s" },
      {     2000L, " 500 \004s" },
      {     1000L, " 1 ms  "    },
      {      500L, " 2 ms  "    },
      {      200L, " 5 ms  "    },
      {      100L, " 10 ms "    },
      {       50L, " 20 ms "    } };




/*
   no_menu_action

   Description:      This function handles a menu action when there is nothing
                     to be done.  It just returns.

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: None.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

void  no_menu_action()
{
    /* variables */
      /* none */



    /* nothing to do - return */
    return;

}




/*
   no_display

   Description:      This function handles displaying a menu option's setting
                     when there is nothing to display.  It just returns,
             ignoring all arguments.

   Arguments:        x_pos (int) - x position (in character cells) at which to
                   display the menu option (not used).
             y_pos (int) - y position (in character cells) at which to
                   display the menu option (not used).
```

```
            style (int) - style with which to display the menu option
                          (not used).
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: None.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

void  no_display(int x_pos, int y_pos, int style)
{
    /* variables */
      /* none */



    /* nothing to do - return */
    return;

}




/*
   set_trigger_mode

   Description:      This function sets the triggering mode to the passed
                     value.

   Arguments:        m (enum trigger_type) - mode to which to set the
                         triggering mode.
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: trigger_mode - initialized to the passed value.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

void  set_trigger_mode(enum trigger_type m)
{
    /* variables */
      /* none */
```

```
    /* set the trigger mode */
    trigger_mode = m;

    /* set the new mode */
    set_mode(trigger_mode);


    /* all done setting the trigger mode - return */
    return;

}




/*
   get_trigger_mode

   Description:      This function returns the current triggering mode.

   Arguments:        None.
   Return Value:     (enum trigger_type) - current triggering mode.

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: trigger_mode - value is returned (not changed).

   Author:           Glen George
   Last Modified:    May 9, 2006

*/

enum trigger_type  get_trigger_mode()
{
    /* variables */
      /* none */



    /* return the current trigger mode */
    return  trigger_mode;

}




/*
   mode_down

   Description:      This function handles moving down the list of trigger
                     modes.  It changes to the "next" triggering mode and
                     sets that as the current mode.

   Arguments:        None.
   Return Value:     None.
```

```
   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: trigger_mode - changed to "next" trigger mode.

   Author:           Glen George
   Last Modified:    May 9, 2006

*/

void  mode_down()
{
    /* variables */
      /* none */



    /* move to the "next" triggering mode */
    if (trigger_mode == NORMAL_TRIGGER)
        trigger_mode = AUTO_TRIGGER;
    else if (trigger_mode == AUTO_TRIGGER)
        trigger_mode = ONESHOT_TRIGGER;
    else
        trigger_mode = NORMAL_TRIGGER;

    /* set the new mode */
    set_mode(trigger_mode);


    /* all done with the trigger mode - return */
    return;

}




/*
   mode_up

   Description:      This function handles moving up the list of trigger
                     modes.  It changes to the "previous" triggering mode and
                     sets that as the current mode.

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: trigger_mode - changed to "previous" trigger mode.

   Author:           Glen George
```

```
   Last Modified:    May 9, 2006

*/

void  mode_up()
{
    /* variables */
      /* none */



    /* move to the "previous" triggering mode */
    if (trigger_mode == NORMAL_TRIGGER)
        trigger_mode = ONESHOT_TRIGGER;
    else if (trigger_mode == AUTO_TRIGGER)
        trigger_mode = NORMAL_TRIGGER;
    else
        trigger_mode = AUTO_TRIGGER;

    /* set the new mode */
    set_mode(trigger_mode);


    /* all done with the trigger mode - return */
    return;

}




/*
   display_mode

   Description:      This function displays the current triggering mode at the
                     passed position, in the passed style.

   Arguments:        x_pos (int) - x position (in character cells) at which to
                   display the trigger mode.
             y_pos (int) - y position (in character cells) at which to
                   display the trigger mode.
             style (int) - style with which to display the trigger
                      mode.
   Return Value:     None.

   Input:            None.
   Output:           The trigger mode is displayed at the passed position on
             the screen.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: trigger_mode - determines which string is displayed.

   Author:           Glen George
   Last Modified:    May 9, 2006

*/

void  display_mode(int x_pos, int y_pos, int style)
{
    /* variables */
```

```
    /* the mode strings (must match enumerated type) */
    const static char * const  modes[] =  {  " Normal   ",
                                             " Automatic",
                                             " One-Shot "  };



    /* display the trigger mode */
    plot_string(x_pos, y_pos, modes[trigger_mode], style);


    /* all done displaying the trigger mode - return */
    return;

}




/*
   set_scale

   Description:      This function sets the scale type to the passed value.

   Arguments:        s (enum scale_type) - scale type to which to initialize
                       the scale status.
   Return Value:     None.

   Input:            None.
   Output:           The new trace display is updated with the new scale.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: scale - initialized to the passed value.

   Author:           Glen George
   Last Modified:    Mar. 13, 1994

*/

void  set_scale(enum scale_type s)
{
    /* variables */
      /* none */



    /* set the scale type */
    scale = s;

    /* output the scale appropriately */
    set_display_scale(scale);


    /* all done setting the scale type - return */
    return;

}
```

```
/*
   scale_down

   Description:      This function handles moving down the list of scale
                     types.  It changes to the "next" type of scale and sets
             this as the current scale type.

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           The new scale is output to the trace display.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: scale - changed to the "next" scale type.

   Author:           Glen George
   Last Modified:    May 9, 2006

*/

void  scale_down()
{
    /* variables */
      /* none */



    /* change to the "next" scale type */
    if (scale == SCALE_NONE)
        scale = SCALE_AXES;
    else if (scale == SCALE_AXES)
        scale = SCALE_GRID;
    else
        scale = SCALE_NONE;

    /* set the scale type */
    set_display_scale(scale);


    /* all done with toggling the scale type - return */
    return;

}




/*
   scale_up

   Description:      This function handles moving up the list of scale types.
                     It changes to the "previous" type of scale and sets this
             as the current scale type.

   Arguments:        None.
   Return Value:     None.
```

```
   Input:            None.
   Output:           The new scale is output to the trace display.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: scale - changed to the "previous" scale type.

   Author:           Glen George
   Last Modified:    May 9, 2006

*/

void  scale_up()
{
    /* variables */
      /* none */



    /* change to the "previous" scale type */
    if (scale == SCALE_NONE)
        scale = SCALE_GRID;
    else if (scale == SCALE_AXES)
        scale = SCALE_NONE;
    else
        scale = SCALE_AXES;

    /* set the scale type */
    set_display_scale(scale);


    /* all done with toggling the scale type - return */
    return;

}




/*
   display_scale

   Description:      This function displays the current scale type at the
                     passed position, in the passed style.

   Arguments:        x_pos (int) - x position (in character cells) at which to
                   display the scale type.
             y_pos (int) - y position (in character cells) at which to
                   display the scale type.
             style (int) - style with which to display the scale type.
   Return Value:     None.

   Input:            None.
   Output:           The scale type is displayed at the passed position on the
             display.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.
```

```
   Global Variables: scale - determines which string is displayed.

   Author:           Glen George
   Last Modified:    Mar. 13, 1994

*/

void  display_scale(int x_pos, int y_pos, int style)
{
    /* variables */

    /* the scale type strings (must match enumerated type) */
    const static char * const  scale_stat[] =  {  " None",
                                                  " Axes",
                                                  " Grid"  };



    /* display the scale status */
    plot_string(x_pos, y_pos, scale_stat[scale], style);


    /* all done displaying the scale status - return */
    return;

}




/*
   set_sweep

   Description:      This function sets the sweep rate to the passed value.
                     The passed value gives the sweep rate to choose from the
             list of sweep rates (it gives the list index).

   Arguments:        s (int) - index into the list of sweep rates to which to
                   set the current sweep rate.
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   The passed index is not checked for validity.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: sweep - initialized to the passed value.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

void  set_sweep(int s)
{
    /* variables */
    int  sample_size;       /* sample size for this sweep rate */



    /* set the new sweep rate */
```

```
    sweep = s;

    /* set the sweep rate for the hardware */
    sample_size = set_sample_rate(sweep_rates[sweep].sample_rate);
    /* also set the sample size for the trace capture */
    set_trace_size(sample_size);


    /* all done initializing the sweep rate - return */
    return;

}




/*
   sweep_down

   Description:      This function handles decreasing the current sweep rate.
             The new sweep rate (and sample size) is sent to the
             hardware (and trace routines).  If an attempt is made to
             lower the sweep rate below the minimum value it is not
             changed.  This routine also updates the sweep delay based
             on the new sweep rate (to keep the delay time constant).

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: sweep - decremented if not already 0.
             delay - increased to keep delay time constant.

   Known Bugs:       The updated delay time is not displayed.  Since the time
             is typically only rounded to the new sample rate, this is
             not a major problem.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

void  sweep_down()
{
    /* variables */
    int  sample_size;       /* sample size for the new sweep rate */



    /* decrease the sweep rate, if not already the minimum */
    if (sweep > 0)  {
        /* not at minimum, adjust delay for new sweep */
    adjust_trg_delay(sweep, (sweep - 1));
    /* now set new sweep rate */
        sweep--;
    }
```

```
    /* set the sweep rate for the hardware */
    sample_size = set_sample_rate(sweep_rates[sweep].sample_rate);
    /* also set the sample size for the trace capture */
    set_trace_size(sample_size);


    /* all done with lowering the sweep rate - return */
    return;

}




/*
   sweep_up

   Description:      This function handles increasing the current sweep rate.
             The new sweep rate (and sample size) is sent to the
             hardware (and trace routines).  If an attempt is made to
             raise the sweep rate above the maximum value it is not
             changed.  This routine also updates the sweep delay based
             on the new sweep rate (to keep the delay time constant).

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: sweep - incremented if not already the maximum value.
             delay - decreased to keep delay time constant.

   Known Bugs:       The updated delay time is not displayed.  Since the time
             is typically only rounded to the new sample rate, this is
             not a major problem.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

void  sweep_up()
{
    /* variables */
    int  sample_size;       /* sample size for the new sweep rate */



    /* increase the sweep rate, if not already the maximum */
    if (sweep < (NO_SWEEP_RATES - 1))  {
        /* not at maximum, adjust delay for new sweep */
    adjust_trg_delay(sweep, (sweep + 1));
    /* now set new sweep rate */
        sweep++;
    }

    /* set the sweep rate for the hardware */
    sample_size = set_sample_rate(sweep_rates[sweep].sample_rate);
```

```
    /* also set the sample size for the trace capture */
    set_trace_size(sample_size);


    /* all done with raising the sweep rate - return */
    return;

}




/*
   display_sweep

   Description:      This function displays the current sweep rate at the
                     passed position, in the passed style.

   Arguments:        x_pos (int) - x position (in character cells) at which to
                    display the sweep rate.
             y_pos (int) - y position (in character cells) at which to
                    display the sweep rate.
             style (int) - style with which to display the sweep rate.
   Return Value:     None.

   Input:            None.
   Output:           The sweep rate is displayed at the passed position on the
             display.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: sweep - determines which string is displayed.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

void  display_sweep(int x_pos, int y_pos, int style)
{
    /* variables */
      /* none */



    /* display the sweep rate */
    plot_string(x_pos, y_pos, sweep_rates[sweep].s, style);


    /* all done displaying the sweep rate - return */
    return;

}




/*
   set_trg_level

   Description:      This function sets the trigger level to the passed value.
```

```
   Arguments:        l (int) - value to which to set the trigger level.
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   The passed value is not checked for validity.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: level - initialized to the passed value.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

void  set_trg_level(int l)
{
    /* variables */
      /* none */



    /* set the trigger level */
    level = l;

    /* set the trigger level in hardware too */
    set_trigger(level, slope);


    /* all done initializing the trigger level - return */
    return;

}




/*
   trg_level_down

   Description:      This function handles decreasing the current trigger
             level.  The new trigger level is sent to the hardware.
             If an attempt is made to lower the trigger level below
             the minimum value it is not changed.

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: level - decremented if not already at the minimum value.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994
```

```
*/

void  trg_level_down()
{
    /* variables */
      /* none */



    /* decrease the trigger level, if not already the minimum */
    if (level > MIN_TRG_LEVEL_SET)
        level--;

    /* set the trigger level for the hardware */
    set_trigger(level, slope);


    /* all done with lowering the trigger level - return */
    return;

}




/*
   trg_level_up

   Description:      This function handles increasing the current trigger
             level.  The new trigger level is sent to the hardware.
             If an attempt is made to raise the trigger level above
             the maximum value it is not changed.

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: level - incremented if not already the maximum value.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

void  trg_level_up()
{
    /* variables */
      /* none */



    /* increase the trigger level, if not already the maximum */
    if (level < MAX_TRG_LEVEL_SET)
        level++;

    /* tell the hardware the new trigger level */
```

```
    set_trigger(level, slope);


    /* all done raising the trigger level - return */
    return;

}




/*
   display_trg_level

   Description:      This function displays the current trigger level at the
                     passed position, in the passed style.

   Arguments:        x_pos (int) - x position (in character cells) at which to
                   display the trigger level.
             y_pos (int) - y position (in character cells) at which to
                   display the trigger level.
             style (int) - style with which to display the trigger
                      level.
   Return Value:     None.

   Input:            None.
   Output:           The trigger level is displayed at the passed position on
             the display.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: level - determines the value displayed.

   Author:           Glen George
   Last Modified:    Mar. 10, 1995

*/

void  display_trg_level(int x_pos, int y_pos, int style)
{
    /* variables */
    char      level_str[] = "        "; /* string containing the trigger level */
    long int  l;            /* trigger level in mV */



    /* compute the trigger level in millivolts */
    l = ((long int) MAX_LEVEL - MIN_LEVEL) * level / (MAX_TRG_LEVEL_SET - MIN_TRG_LEVEL_SET) +
    MIN_LEVEL;

    /* convert the level to the string (leave first character blank) */
    cvt_num_field(l, &level_str[1]);

    /* add in the units */
    level_str[7] = 'V';

    /* now finally display the trigger level */
    plot_string(x_pos, y_pos, level_str, style);
    /* all done displaying the trigger level - return */
    return;
```

```
}




/*
   set_trg_slope

   Description:      This function sets the trigger slope to the passed value.

   Arguments:        s (enum slope_type) - trigger slope type to which to set
                       the locally global slope.
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: slope - set to the passed value.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

void  set_trg_slope(enum slope_type s)
{
    /* variables */
      /* none */



    /* set the slope type */
    slope = s;

    /* also tell the hardware what the slope is */
    set_trigger(level, slope);


    /* all done setting the trigger slope - return */
    return;

}




/*
   trg_slope_toggle

   Description:      This function handles toggling (and setting) the current
             trigger slope.

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   None.
```

```
   Algorithms:       None.
   Data Structures:  None.

   Global Variables: slope - toggled.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

void  trg_slope_toggle()
{
    /* variables */
      /* none */



    /* toggle the trigger slope */
    if (slope == SLOPE_POSITIVE)
        slope = SLOPE_NEGATIVE;
    else
        slope = SLOPE_POSITIVE;

    /* set the new trigger slope */
    set_trigger(level, slope);


    /* all done with the trigger slope - return */
    return;

}




/*
   display_trg_slope

   Description:      This function displays the current trigger slope at the
                     passed position, in the passed style.

   Arguments:        x_pos (int) - x position (in character cells) at which to
                   display the trigger slope.
             y_pos (int) - y position (in character cells) at which to
                   display the trigger slope.
             style (int) - style with which to display the trigger
                      slope.
   Return Value:     None.

   Input:            None.
   Output:           The trigger slope is displayed at the passed position on
             the screen.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: slope - determines which string is displayed.

   Author:           Glen George
   Last Modified:    Mar. 13, 1994
```

```
*/

void  display_trg_slope(int x_pos, int y_pos, int style)
{
    /* variables */

    /* the trigger slope strings (must match enumerated type) */
    const static char * const  slopes[] =  {  " +", " -"  };



    /* display the trigger slope */
    plot_string(x_pos, y_pos, slopes[slope], style);


    /* all done displaying the trigger slope - return */
    return;

}




/*
   set_trg_delay

   Description:      This function sets the trigger delay to the passed value.

   Arguments:        d (long int) - value to which to set the trigger delay.
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   The passed value is not checked for validity.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: delay - initialized to the passed value.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

void  set_trg_delay(long int d)
{
    /* variables */
      /* none */



    /* set the trigger delay */
    delay = d;

    /* set the trigger delay in hardware too */
    set_delay(delay);


    /* all done initializing the trigger delay - return */
    return;

}
```

```
/*
   trg_delay_down

   Description:      This function handles decreasing the current trigger
             delay.  The new trigger delay is sent to the hardware.
             If an attempt is made to lower the trigger delay below
             the minimum value it is not changed.

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: delay - decremented if not already at the minimum value.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

void  trg_delay_down()
{
    /* variables */
      /* none */



    /* decrease the trigger delay, if not already the minimum */
    if (delay > MIN_DELAY)
        delay--;

    /* set the trigger delay for the hardware */
    set_delay(delay);


    /* all done with lowering the trigger delay - return */
    return;

}




/*
   trg_delay_up

   Description:      This function handles increasing the current trigger
             delay.  The new trigger delay is sent to the hardware.
             If an attempt is made to raise the trigger delay above
             the maximum value it is not changed.

   Arguments:        None.
   Return Value:     None.
```

```
   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: delay - incremented if not already the maximum value.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

void  trg_delay_up()
{
    /* variables */
      /* none */



    /* increase the trigger delay, if not already the maximum */
    if (delay < MAX_DELAY)
        delay++;

    /* tell the hardware the new trigger delay */
    set_delay(delay);


    /* all done raising the trigger delay - return */
    return;

}




/*
   adjust_trg_delay

   Description:      This function adjusts the trigger delay for a new sweep
             rate.  The factor to adjust the delay by is determined
             by looking up the sample rates in the sweep_rates array.
             If the delay goes out of range, due to the adjustment it
             is reset to the maximum or minimum valid value.

   Arguments:        old_sweep (int) - old sweep rate (index into sweep_rates
                       array).
             new_sweep (int) - new sweep rate (index into sweep_rates
                       array).
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       The delay is multiplied by 10 times the ratio of the
             sweep sample rates then divided by 10.  This is done to
             avoid floating point arithmetic and integer truncation
             problems.
   Data Structures:  None.
```

```
   Global Variables: delay - adjusted based on passed sweep rates.

   Known Bugs:       The updated delay time is not displayed.  Since the time
             is typically only rounded to the new sample rate, this is
             not a major problem.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

static void  adjust_trg_delay(int old_sweep, int new_sweep)
{
    /* variables */
      /* none */



    /* multiply by 10 times the ratio of sweep rates */
    delay *= (10 * sweep_rates[new_sweep].sample_rate) / sweep_rates[old_sweep].sample_rate;
    /* now divide the factor of 10 back out */
    delay /= 10;

    /* make sure delay is not out of range */
    if (delay > MAX_DELAY)
        /* delay is too large - set to maximum */
        delay = MAX_DELAY;
    if (delay < MIN_DELAY)
        /* delay is too small - set to minimum */
    delay = MIN_DELAY;


    /* tell the hardware the new trigger delay */
    set_delay(delay);


    /* all done adjusting the trigger delay - return */
    return;

}




/*
   display_trg_delay

   Description:      This function displays the current trigger delay at the
                     passed position, in the passed style.

   Arguments:        x_pos (int) - x position (in character cells) at which to
                   display the trigger delay.
             y_pos (int) - y position (in character cells) at which to
                   display the trigger delay.
             style (int) - style with which to display the trigger
                      delay.
   Return Value:     None.

   Input:            None.
   Output:           The trigger delay is displayed at the passed position on
             the display.

   Error Handling:   None.
```

```
   Algorithms:       None.
   Data Structures:  None.

   Global Variables: delay - determines the value displayed.

   Author:           Glen George
   Last Modified:    May 3, 2006

*/

void  display_trg_delay(int x_pos, int y_pos, int style)
{
    /* variables */
    char      delay_str[] = "         "; /* string containing the trigger delay */
    long int  units_adj;         /* adjustment to get to microseconds */

    long int  d;                         /* delay in appropriate units */


    /* compute the delay in the appropriate units */
    /* have to watch out for overflow, so be careful */
    if (sweep_rates[sweep].sample_rate > 1000000L)  {
        /* have a fast sweep rate, could overflow */
        /* first compute in units of 100 ns */
        d = delay * (10000000L / sweep_rates[sweep].sample_rate);
    /* now convert to nanoseconds */
    d *= 100L;
    /* need to divide by 1000 to get to microseconds */
    units_adj = 1000;
    }
    else  {
        /* slow sweep rate, don't have to worry about overflow */
        d = delay * (1000000L / sweep_rates[sweep].sample_rate);
    /* already in microseconds, so adjustment is 1 */
    units_adj = 1;
    }

    /* convert it to the string (leave first character blank) */
    cvt_num_field(d, &delay_str[1]);

    /* add in the units */
    if (((d / units_adj) < 1000) && ((d / units_adj) > -1000) && (units_adj == 1000)) {
        /* delay is in microseconds */
    delay_str[7] = '\004';
    delay_str[8] = 's';
    }
    else if (((d / units_adj) < 1000000) && ((d / units_adj) > -1000000)) {
        /* delay is in milliseconds */
    delay_str[7] = 'm';
    delay_str[8] = 's';
    }
    else if (((d / units_adj) < 1000000000) && ((d / units_adj) > -1000000000))  {
        /* delay is in seconds */
    delay_str[7] = 's';
    delay_str[8] = ' ';
    }
    else  {
        /* delay is in kiloseconds */
    delay_str[7] = 'k';
    delay_str[8] = 's';
    }


    /* now actually display the trigger delay */
```

```
    plot_string(x_pos, y_pos, delay_str, style);


    /* all done displaying the trigger delay - return */
    return;

}




/*
   cvt_num_field

   Description:      This function converts the passed number (numeric field
                     value) to a string and returns that in the passed string
             reference.  The number may be signed, and a sign (+ or -)
             is always generated.  The number is assumed to have three
             digits to the right of the decimal point.  Only the four
             most significant digits of the number are displayed and
             the decimal point is shifted appropriately.  (Four digits
             are always generated by the function).

   Arguments:        n (long int) - numeric field value to convert.
             s (char *)   - pointer to string in which to return the
                        converted field value.
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       The algorithm used assumes four (4) digits are being
             converted.
   Data Structures:  None.

   Global Variables: None.

   Known Bugs:       If the passed long int is the largest negative long int,
             the function will display garbage.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

static void  cvt_num_field(long int n, char *s)
{
    /* variables */
    int  dp = 3;        /* digits to right of decimal point */
    int  d;         /* digit weight (power of 10) */

    int  i = 0;         /* string index */



    /* first get the sign (and make n positive for conversion) */
    if (n < 0)  {
        /* n is negative, set sign and convert to positive */
    s[i++] = '-';
    n = -n;
    }
    else  {
```

```
        /* n is positive, set sign only */
    s[i++] = '+';
    }


    /* make sure there are no more than 4 significant digits */
    while (n > 9999)  {
        /* have more than 4 digits - get rid of one */
    n /= 10;
    /* adjust the decimal point */
    dp--;
    }

    /* if decimal point is non-positive, make positive */
    /* (assume will take care of adjustment with output units in this case) */
    while (dp <= 0)
       dp += 3;


    /* adjust dp to be digits to the right of the decimal point */
    /* (assuming 4 digits) */
    dp = 4 - dp;


    /* finally, loop getting and converting digits */
    for (d = 1000; d > 0; d /= 10)  {

        /* check if need decimal the decimal point now */
    if (dp-- == 0)
        /* time for decimal point */
        s[i++] = '.';

    /* get and convert this digit */
    s[i++] = (n / d) + '0';
    /* remove this digit from n */
    n %= d;
    }


    /* all done converting the number, return */
    return;

}
```

```
/*****************************************************************************/
/*                                                                           */
/*                                 MENUACT.H                                 */
/*                            Menu Action Functions                          */
/*                                 Include File                              */
/*                          Digital Oscilloscope Project                     */
/*                                   EE/CS 52                                */
/*                                                                           */
/*****************************************************************************/

/*
   This file contains the constants and function prototypes for the functions
   which carry out menu actions and display and initialize menu settings for
   the Digital Oscilloscope project (the functions are defined in menuact.c).


   Revision History:
      3/8/94   Glen George       Initial revision.
      3/13/94  Glen George       Updated comments.
      3/13/94  Glen George       Changed definition of enum scale_type (was
                       enum scale_status).
      3/10/95  Glen George       Changed MAX_TRG_LEVEL_SET (maximum trigger
                       level) to 127 to match specification.
      3/17/97  Glen George       Updated comments.
      5/3/06   Glen George       Updated comments.
      5/9/06   Glen George       Added a new mode (AUTO_TRIGGER) and a new
                                 scale (SCALE_GRID).
      5/9/06   Glen George       Added menu functions for mode and scale to
                                 move up and down a list instead of just
                     toggling the selection.
      5/9/06   Glen George       Added declaration for the accessor to the
                                 current trigger mode (get_trigger_mode).
*/



#ifndef  __MENUACT_H__
    #define  __MENUACT_H__


/* library include files */
  /* none */

/* local include files */
#include  "interfac.h"
#include  "lcdout.h"




/* constants */

/* min and max trigger level settings */
#define  MIN_TRG_LEVEL_SET   0
#define  MAX_TRG_LEVEL_SET   127

/* number of different sweep rates */
#define  NO_SWEEP_RATES     (sizeof(sweep_rates) / sizeof(struct sweep_info))




/* structures, unions, and typedefs */
```

```
/* types of triggering modes */
enum trigger_type  {  NORMAL_TRIGGER,       /* normal triggering */
             AUTO_TRIGGER,     /* automatic triggering */
             ONESHOT_TRIGGER       /* one-shot triggering */
          };

/* types of displayed scales */
enum scale_type    {  SCALE_NONE,       /* no scale is displayed */
             SCALE_AXES,       /* scale is a set of axes */
             SCALE_GRID        /* scale is a grid */
          };

/* types of trigger slopes */
enum slope_type    {  SLOPE_POSITIVE,       /* positive trigger slope */
             SLOPE_NEGATIVE        /* negative trigger slope */
          };

/* sweep rate information */
struct sweep_info  {  long int     sample_rate;    /* sample rate */
             const char  *s;          /* sweep rate string */
          };




/* function declarations */

/* menu option actions */
void  no_menu_action(void);    /* no action to perform */
void  mode_down(void);         /* change to the "next" trigger mode */
void  mode_up(void);           /* change to the "previous" trigger mode */
void  scale_down(void);        /* change to the "next" scale type */
void  scale_up(void);          /* change to the "previous" scale type */
void  sweep_down(void);        /* decrease the sweep rate */
void  sweep_up(void);          /* increase the sweep rate */
void  trg_level_down(void);    /* decrease the trigger level */
void  trg_level_up(void);      /* increase the trigger level */
void  trg_slope_toggle(void);  /* toggle the trigger slope */
void  trg_delay_down(void);    /* decrease the trigger delay */
void  trg_delay_up(void);      /* increase the trigger delay */

/* option accessor routines */
enum trigger_type  get_trigger_mode(void);  /* get the current trigger mode */

/* option initialization routines */
void  set_trigger_mode(enum trigger_type);  /* set the trigger mode */
void  set_scale(enum scale_type);           /* set the scale type */
void  set_sweep(int);                   /* set the sweep rate */
void  set_trg_level(int);               /* set the trigger level */
void  set_trg_slope(enum slope_type);       /* set the trigger slope */
void  set_trg_delay(long int);          /* set the tigger delay */

/* option display routines */
void  no_display(int, int, int);     /* no option setting to display */
void  display_mode(int, int, int);       /* display trigger mode */
void  display_scale(int, int, int);      /* display the scale type */
void  display_sweep(int, int, int);      /* display the sweep rate */
void  display_trg_level(int, int, int);  /* display the trigger level */
void  display_trg_slope(int, int, int);  /* display the trigger slope */
void  display_trg_delay(int, int, int);  /* display the tigger delay */


#endif
```

```
/*****************************************************************************/
/*                                                                           */
/*                                   MENU                                    */
/*                              Menu Functions                               */
/*                        Digital Oscilloscope Project                       */
/*                                  EE/CS 52                                 */
/*                                                                           */
/*****************************************************************************/

/*
   This file contains the functions for processing menu entries for the
   Digital Oscilloscope project.  These functions take care of maintaining the
   menus and handling menu updates for the system.  The functions included
   are:
      clear_menu       - remove the menu from the display
      display_menu     - display the menu
      init_menu        - initialize menus
      menu_entry_left  - take care of <Left> key for a menu entry
      menu_entry_right - take care of <Right> key for a menu entry
      next_entry       - next menu entry
      previous_entry   - previous menu entry
      refresh_menu     - re-display the menu if currently being displayed
      reset_menu       - reset the current selection to the top of the menu

   The local functions included are:
      display_entry    - display a menu entry (including option setting)

   The locally global variable definitions included are:
      menu             - the menu
      menu_display     - whether or not the menu is currently displayed
      menu_entry       - the currently selected menu entry


   Revision History
      3/8/94   Glen George       Initial revision.
      3/9/94   Glen George       Changed position of const keyword in array
                 declarations involving pointers.
      3/13/94  Glen George       Updated comments.
      3/13/94  Glen George       Added display_entry function to output a menu
                 entry and option setting to the LCD (affects
                 many functions).
      3/13/94  Glen George       Changed calls to set_status due to changing
                      enum scale_status definition.
      3/13/94  Glen George       No longer clear the menu area before
                 restoring the trace in clear_menu() (not
                 needed).
      3/17/97  Glen George       Updated comments.
      3/17/97  Glen George       Fixed minor bug in reset_menu().
      3/17/97  Glen George       When initializing the menu in init_menu(),
                 set the delay to MIN_DELAY instead of 0 and
                 trigger to a middle value instead of
                 MIN_TRG_LEVEL_SET.
      5/3/06   Glen George       Changed to a more appropriate constant in
                                 display_entry().
      5/3/06   Glen George       Updated comments.
      5/9/06   Glen George       Changed menus to handle a list for mode and
                               scale (move up and down list), instead of
                      toggling values.
*/



/* library include files */
  /* none */
```

```
/* local include files */
#include  "scopedef.h"
#include  "lcdout.h"
#include  "menu.h"
#include  "menuact.h"
#include  "tracutil.h"




/* local function declarations */
static void  display_entry(int, int);      /* display a menu entry and its setting */




/* locally global variables */
static int  menu_display;           /* TRUE if menu is currently displayed */

const static struct menu_item  menu[] =     /* the menu */
    { { "Mode",    0, 4, display_mode      },
      { "Scale",   0, 5, display_scale     },
      { "Sweep",   0, 5, display_sweep     },
      { "Trigger", 0, 7, no_display        },
      { "Level",   2, 7, display_trg_level },
      { "Slope",   2, 7, display_trg_slope },
      { "Delay",   2, 7, display_trg_delay },
    };

static int  menu_entry;         /* currently selected menu entry */




/*
   init_menu

   Description:      This function initializes the menu routines.  It sets
                     the current menu entry to the first entry, indicates the
             display is off, and initializes the options (and
             hardware) to normal trigger mode, scale displayed, the
             fastest sweep rate, a middle trigger level, positive
             trigger slope, and minimum delay.  Finally, it displays
             the menu.

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           The menu is displayed.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: menu_display - reset to FALSE.
             menu_entry   - reset to first entry (0).

   Author:           Glen George
   Last Modified:    Mar. 17, 1997

*/
```

```
void  init_menu(void)
{
    /* variables */
      /* none */



    /* set the menu parameters */
    menu_entry = 0;     /* first menu entry */
    menu_display = FALSE;   /* menu is not currently displayed (but it will be shortly) */


    /* set the scope (option) parameters */
    set_trigger_mode(NORMAL_TRIGGER);   /* normal triggering */
    set_scale(SCALE_AXES);      /* scale is axes */
    set_sweep(0);           /* first sweep rate */
    set_trg_level((MIN_TRG_LEVEL_SET + MAX_TRG_LEVEL_SET) / 2); /* middle trigger level */
    set_trg_slope(SLOPE_POSITIVE);  /* positive slope */
    set_trg_delay(MIN_DELAY);       /* minimum delay */


    /* now display the menu */
    display_menu();


    /* done initializing, return */
    return;

}




/*
   clear_menu

   Description:      This function removes the menu from the display.  The
                     trace under the menu is restored.  The flag menu_display,
             is cleared, indicating the menu is no longer being
             displayed.  Note: if the menu is not currently being
             displayed this function does nothing.

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           The menu if displayed, is removed and the trace under it
             is rewritten.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: menu_display - checked and set to FALSE.

   Author:           Glen George
   Last Modified:    Mar. 13, 1994

*/

void  clear_menu(void)
{
```

```
    /* variables */
      /* none */



    /* check if the menu is currently being displayed */
    if (menu_display)  {

        /* menu is being displayed - turn it off and restore the trace in that area */
    restore_menu_trace();
    }


    /* no longer displaying the menu */
    menu_display = FALSE;


    /* all done, return */
    return;

}




/*
   display_menu

   Description:      This function displays the menu.  The trace under the
                     menu is overwritten (but it was saved).  The flag
             menu_display, is also set, indicating the menu is
             currently being displayed.  Note: if the menu is already
             being displayed this function does not redisplay it.

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           The menu is displayed.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: menu_display - set to TRUE.
             menu_entry   - used to highlight currently selected entry.

   Author:           Glen George
   Last Modified:    Mar. 13, 1994

*/

void  display_menu(void)
{
    /* variables */
    int  i;     /* loop index */



    /* check if the menu is currently being displayed */
    if (!menu_display)  {

        /* menu is not being displayed - turn it on */
```

```
    /* display it entry by entry */
    for (i = 0; i < NO_MENU_ENTRIES; i++)  {

        /* display this entry - check if it should be highlighted */
        if (i == menu_entry)
            /* currently selected entry - highlight it */
            display_entry(i, TRUE);
        else
            /* not the currently selected entry - "normal video" */
            display_entry(i, FALSE);
        }
    }


    /* now are displaying the menu */
    menu_display = TRUE;


    /* all done, return */
    return;

}




/*
   refresh_menu

   Description:      This function displays the menu if it is currently being
             displayed.  The trace under the menu is overwritten (but
             it was already saved).

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           The menu is displayed.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: menu_display - determines if menu should be displayed.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

void  refresh_menu(void)
{
    /* variables */
      /* none */



    /* check if the menu is currently being displayed */
    if (menu_display)  {

        /* menu is currently being displayed - need to refresh it */
    /* do this by turning off the display, then forcing it back on */
    menu_display = FALSE;
```

```
    display_menu();
    }


    /* refreshed the menu if it was displayed, now return */
    return;

}




/*
   reset_menu

   Description:      This function resets the current menu selection to the
                     first menu entry.  If the menu is currently being
             displayed the display is updated.

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           The menu display is updated if it is being displayed.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: menu_display - checked to see if menu is displayed.
             menu_entry   - reset to 0 (first entry).

   Author:           Glen George
   Last Modified:    Mar. 17, 1997

*/

void  reset_menu(void)
{
    /* variables */
      /* none */



    /* check if the menu is currently being displayed */
    if (menu_display)  {

        /* menu is being displayed */
    /* remove highlight from currently selected entry */
    display_entry(menu_entry, FALSE);
    }


    /* reset the currently selected entry */
    menu_entry = 0;


    /* finally, highlight the first entry if the menu is being displayed */
    if (menu_display)
    display_entry(menu_entry, TRUE);
```

```
    /* all done, return */
    return;

}




/*
   next_entry

   Description:      This function changes the current menu selection to the
                     next menu entry.  If the current selection is the last
             entry in the menu, it is not changed.  If the menu is
             currently being displayed, the display is updated.

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           The menu display is updated if it is being displayed and
             the entry selected changes.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: menu_display - checked to see if menu is displayed.
             menu_entry   - updated to a new entry (if not at end).

   Author:           Glen George
   Last Modified:    Mar. 13, 1994

*/

void  next_entry(void)
{
    /* variables */
      /* none */



    /* only update if not at end of the menu */
    if (menu_entry < (NO_MENU_ENTRIES - 1))  {

        /* not at the end of the menu */

    /* turn off current entry if displaying */
    if (menu_display)
            /* displaying menu - turn off currently selected entry */
        display_entry(menu_entry, FALSE);

    /* update the menu entry to the next one */
    menu_entry++;

    /* now highlight this entry if displaying the menu */
    if (menu_display)
            /* displaying menu - highlight newly selected entry */
        display_entry(menu_entry, TRUE);
    }


    /* all done, return */
```

```
    return;

}




/*
   previous_entry

   Description:      This function changes the current menu selection to the
                     previous menu entry.  If the current selection is the
             first entry in the menu, it is not changed.  If the menu
             is currently being displayed, the display is updated.

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           The menu display is updated if it is being displayed and
             the currently selected entry changes.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: menu_display - checked to see if menu is displayed.
             menu_entry   - updated to a new entry (if not at start).

   Author:           Glen George
   Last Modified:    Mar. 13, 1994

*/

void  previous_entry(void)
{
    /* variables */
      /* none */



    /* only update if not at the start of the menu */
    if (menu_entry > 0)  {

        /* not at the start of the menu */

    /* turn off current entry if displaying */
    if (menu_display)
            /* displaying menu - turn off currently selected entry */
        display_entry(menu_entry, FALSE);

    /* update the menu entry to the previous one */
    menu_entry--;

    /* now highlight this entry if displaying the menu */
    if (menu_display)
            /* displaying menu - highlight newly selected entry */
        display_entry(menu_entry, TRUE);

    }


    /* all done, return */
```

```
    return;

}




/*
   menu_entry_left

   Description:      This function handles the <Left> key for the current menu
                     selection.  It does this by doing a table lookup on the
             current menu selection.

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           The menu display is updated if it is being displayed and
             the <Left> key causes a change to the display.

   Error Handling:   None.

   Algorithms:       Table lookup is used to determine what to do for the
             input key.
   Data Structures:  An array holds the table of key processing routines.

   Global Variables: menu_entry - used to select the processing function.

   Author:           Glen George
   Last Modified:    May 9, 2006

*/

void  menu_entry_left(void)
{
    /* variables */

    /* key processing functions */
    static void  (* const process[])(void) =
       /*  Mode            Scale             Sweep           Trigger      */
        {  mode_down,      scale_down,       sweep_down,     trace_rearm,
           trg_level_down, trg_slope_toggle, trg_delay_down               };
       /*  Level           Slope             Delay                        */



    /* invoke the appropriate <Left> key function */
    process[menu_entry]();

    /* if displaying menu entries, display the new value */
    /* note: since it is being changed - know this option is selected */
    if (menu_display)  {
        menu[menu_entry].display((MENU_X + menu[menu_entry].opt_off),
                         (MENU_Y + menu_entry), OPTION_SELECTED);
    }


    /* all done, return */
    return;

}
```

```
/*
   menu_entry_right

   Description:      This function handles the <Right> key for the current
                     menu selection.  It does this by doing a table lookup on
             the current menu selection.

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           The menu display is updated if it is being displayed and
             the <Right> key causes a change to the display.

   Error Handling:   None.

   Algorithms:       Table lookup is used to determine what to do for the
             input key.
   Data Structures:  An array holds the table of key processing routines.

   Global Variables: menu       - used to display the new menu value.
             menu_entry - used to select the processing function.

   Author:           Glen George
   Last Modified:    May 9, 2006

*/

void  menu_entry_right(void)
{
    /* variables */

    /* key processing functions */
    static void  (* const process[])(void) =
       /*  Mode          Scale             Sweep           Trigger      */
        {  mode_up    ,  scale_up,     sweep_up,       trace_rearm,
           trg_level_up, trg_slope_toggle, trg_delay_up                 };
       /*  Level         Slope             Delay                        */



    /* invoke the appropriate <Right> key function */
    process[menu_entry]();

    /* if displaying menu entries, display the new value */
    /* note: since it is being changed - know this option is selected */
    if (menu_display)  {
        menu[menu_entry].display((MENU_X + menu[menu_entry].opt_off),
                         (MENU_Y + menu_entry), OPTION_SELECTED);
    }


    /* all done, return */
    return;

}




/*
   display_entry
```

```
   Description:      This function displays the passed menu entry and its
             current option setting.  If the second argument is TRUE
             it displays them with color SELECTED and OPTION_SELECTED
             respectively.  If the second argument is FALSE it
             displays the menu entry with color NORMAL and the option
             setting with color OPTION_NORMAL.

   Arguments:        entry (int)    - menu entry to be displayed.
             selected (int) - whether or not the menu entry is
                          currently selected (determines the color
                       with which the entry is output).
   Return Value:     None.

   Input:            None.
   Output:           The menu entry is output to the LCD.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: menu - used to display the menu entry.

   Author:           Glen George
   Last Modified:    Aug. 13, 2004

*/

static void  display_entry(int entry, int selected)
{
    /* variables */
      /* none */



    /* output the menu entry with the appropriate color */
    plot_string((MENU_X + menu[entry].h_off), (MENU_Y + entry), menu[entry].s,
            (selected ? SELECTED : NORMAL));
    /* also output the menu option with the appropriate color */
    menu[entry].display((MENU_X + menu[entry].opt_off), (MENU_Y + entry),
                (selected ? OPTION_SELECTED : OPTION_NORMAL));


    /* all done outputting this menu entry - return */
    return;

}
```

```
/*****************************************************************************/
/*                                                                           */
/*                                  MENU.H                                   */
/*                              Menu Functions                               */
/*                               Include File                                */
/*                        Digital Oscilloscope Project                       */
/*                                 EE/CS 52                                  */
/*                                                                           */
/*****************************************************************************/

/*
   This file contains the constants and function prototypes for the functions
   which deal with menus (defined in menu.c) for the Digital Oscilloscope
   project.


   Revision History:
      3/8/94   Glen George       Initial revision.
      3/13/94  Glen George       Updated comments.
      3/13/94  Glen George       Added definitions for SELECTED,
                     OPTION_NORMAL, and OPTION_SELECTED.
*/



#ifndef  __MENU_H__
    #define  __MENU_H__


/* library include files */
  /* none */

/* local include files */
#include  "interfac.h"
#include  "scopedef.h"
#include  "lcdout.h"




/* constants */

/* menu size */
#define  MENU_WIDTH   16        /* menu width (in characters) */
#define  MENU_HEIGHT   7        /* menu height (in characters) */
#define  MENU_SIZE_X  (MENU_WIDTH * HORIZ_SIZE)  /* menu width (in pixels) */
#define  MENU_SIZE_Y  (MENU_HEIGHT * VERT_SIZE)  /* menu height (in pixels) */

/* menu position */
#define  MENU_X    (LCD_WIDTH - MENU_WIDTH - 1)  /* x position (in characters) */
#define  MENU_Y    0                    /* y position (in characters) */
#define  MENU_UL_X (MENU_X * HORIZ_SIZE)         /* x position (in pixels) */
#define  MENU_UL_Y (MENU_Y * VERT_SIZE)          /* y position (in pixels) */

/* menu colors */
#define  SELECTED         REVERSE   /* color for a selected menu entry */
#define  OPTION_SELECTED  NORMAL        /* color for a selected menu entry option */
#define  OPTION_NORMAL    NORMAL        /* color for an unselected menu entry option */

/* number of menu entries */
#define  NO_MENU_ENTRIES  (sizeof(menu) / sizeof(struct menu_item))
```

```
/* structures, unions, and typedefs */

/* data for an item in a menu */
struct menu_item  {  const char  *s;        /* string for menu entry */
             int          h_off;    /* horizontal offset of entry */
             int          opt_off;  /* horizontal offset of option setting */
             void       (*display)(int, int, int);  /* option display function */
          };




/* function declarations */

/* menu initialization function */
void  init_menu(void);

/* menu display functions */
void  clear_menu(void);        /* clear the menu display */
void  display_menu(void);      /* display the menu */
void  refresh_menu(void);      /* refresh the menu */

/* menu update functions */
void  reset_menu(void);        /* reset the menu to first entry */
void  next_entry(void);        /* go to the next menu entry */
void  previous_entry(void);    /* go to the previous menu entry */

/* menu entry functions */
void  menu_entry_left(void);       /* do the <Left> key for the menu entry */
void  menu_entry_right(void);      /* do the <Right> key for the menu entry */


#endif
```

```
/*****************************************************************************/
/*                                                                           */
/*                                  KEYPROC                                  */
/*                           Key Processing Functions                        */
/*                         Digital Oscilloscope Project                      */
/*                                  EE/CS 52                                 */
/*                                                                           */
/*****************************************************************************/

/*
   This file contains the key processing functions for the Digital
   Oscilloscope project.  These functions are called by the main loop of the
   system.  The functions included are:
      menu_down  - process the <Down> key while in a menu
      menu_key   - process the <Menu> key
      menu_left  - process the <Left> key while in a menu
      menu_right - process the <Right> key while in a menu
      menu_up    - process the <Up> key while in a menu
      no_action  - nothing to do

   The local functions included are:
      none

   The locally global variable definitions included are:
      none


   Revision History
      3/8/94   Glen George       Initial revision.
      3/13/94  Glen George       Updated comments.
*/



/* library include files */
  /* none */

/* local include files */
#include  "scopedef.h"
#include  "keyproc.h"
#include  "menu.h"




/*
   no_action

   Description:      This function handles a key when there is nothing to be
                     done.  It just returns.

   Arguments:        cur_state (enum status) - the current system state.
   Return Value:     (enum status) - the new system state (same as current
             state).

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: None.
```

```
   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

enum status  no_action(enum status cur_state)
{
    /* variables */
      /* none */



    /* return the current state */
    return  cur_state;

}




/*
   menu_key

   Description:      This function handles the <Menu> key.  If the passed
                     state is MENU_ON, the menu is turned off.  If the passed
             state is MENU_OFF, the menu is turned on.  The returned
             state is the "opposite" of the passed state.

   Arguments:        cur_state (enum status) - the current system state.
   Return Value:     (enum status) - the new system state ("opposite" of the
             as current state).

   Input:            None.
   Output:           The menu is either turned on or off.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: None.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

enum status  menu_key(enum status cur_state)
{
    /* variables */
      /* none */



    /* check if need to turn the menu on or off */
    if (cur_state == MENU_ON)
        /* currently the menu is on, turn it off */
    clear_menu();
    else
        /* currently the menu is off, turn it on */
    display_menu();
```

```
    /* all done, return the "opposite" of the current state */
    if (cur_state == MENU_ON)
        /* state was MENU_ON, change it to MENU_OFF */
        return  MENU_OFF;
    else
        /* state was MENU_OFF, change it to MENU_ON */
        return  MENU_ON;

}




/*
   menu_up

   Description:      This function handles the <Up> key when in a menu.  It
                     goes to the previous menu entry and leaves the system
             state unchanged.

   Arguments:        cur_state (enum status) - the current system state.
   Return Value:     (enum status) - the new system state (same as current
             state).

   Input:            None.
   Output:           The menu display is updated.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: None.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

enum status  menu_up(enum status cur_state)
{
    /* variables */
      /* none */



    /* go to the previous menu entry */
    previous_entry();


    /* return the current state */
    return  cur_state;

}




/*
   menu_down

   Description:      This function handles the <Down> key when in a menu.  It
                     goes to the next menu entry and leaves the system state
             unchanged.
```

```
   Arguments:        cur_state (enum status) - the current system state.
   Return Value:     (enum status) - the new system state (same as current
             state).

   Input:            None.
   Output:           The menu display is updated.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: None.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

enum status  menu_down(enum status cur_state)
{
    /* variables */
      /* none */



    /* go to the next menu entry */
    next_entry();


    /* return the current state */
    return  cur_state;

}




/*
   menu_left

   Description:      This function handles the <Left> key when in a menu.  It
                     invokes the left function for the current menu entry and
             leaves the system state unchanged.

   Arguments:        cur_state (enum status) - the current system state.
   Return Value:     (enum status) - the new system state (same as current
             state).

   Input:            None.
   Output:           The menu display may be updated.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: None.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/
```

```
enum status  menu_left(enum status cur_state)
{
    /* variables */
      /* none */



    /* invoke the <Left> key function for the current menu entry */
    menu_entry_left();


    /* return the current state */
    return  cur_state;

}




/*
   menu_right

   Description:      This function handles the <Right> key when in a menu.  It
                     invokes the right function for the current menu entry and
             leaves the system state unchanged.

   Arguments:        cur_state (enum status) - the current system state.
   Return Value:     (enum status) - the new system state (same as current
             state).

   Input:            None.
   Output:           The menu display may be updated.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: None.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

enum status  menu_right(enum status cur_state)
{
    /* variables */
      /* none */



    /* invoke the <Right> key function for the current menu entry */
    menu_entry_right();


    /* return the current state */
    return  cur_state;

}
```

```
/*****************************************************************************/
/*                                                                           */
/*                                KEYPROC.H                                  */
/*                          Key Processing Functions                         */
/*                                Include File                               */
/*                        Digital Oscilloscope Project                       */
/*                                 EE/CS 52                                  */
/*                                                                           */
/*****************************************************************************/

/*
   This file contains the constants and function prototypes for the key
   processing functions (defined in keyproc.c) for the Digital Oscilloscope
   project.


   Revision History:
      3/8/94   Glen George       Initial revision.
      3/13/94  Glen George       Updated comments.
*/



#ifndef  __KEYPROC_H__
    #define  __KEYPROC_H__


/* library include files */
  /* none */

/* local include files */
#include  "scopedef.h"




/* constants */
    /* none */




/* structures, unions, and typedefs */
    /* none */




/* function declarations */

enum status  no_action(enum status);      /* nothing to do */

enum status  menu_key(enum status);   /* process the <Menu> key */

enum status  menu_up(enum status);    /* <Up> key in a menu */
enum status  menu_down(enum status);      /* <Down> key in a menu */
enum status  menu_left(enum status);      /* <Left> key in a menu */
enum status  menu_right(enum status);     /* <Right> key in a menu */


#endif
```

```
/*************************************************************************/
/*                                                                       */
/*                                CHAR57                                 */
/*                          5x7 Dot Matrix Codes                         */
/*                      Digital Oscilloscope Project                     */
/*                                EE/CS 52                               */
/*                                                                       */
/*************************************************************************/

/*
   This file contains a table of dot matrix patterns for vertically scanned
   5x7 characters.  The table entries are in ASCII order with 7 bytes per
   character.  The table starts with 32 special characters (mostly blank
   characters) then space, the start of the printable ASCII character set.
   The table is called char_patterns.  In each byte (horizontal row) the
   leftmost pixel is given by bit 4 and the rightmost by bit 0.


   Revision History
      5/27/08  Glen George       Initial revision (from 3/10/95 version of
                                 char57.asm).
*/




/* library include files */
  /* none */

/* local include files */
  /* none */



/* the character pattern table */
const unsigned char  char_patterns[] = {

    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,   /* UNUSED (0x00)           */
    0x04, 0x0E, 0x15, 0x04, 0x04, 0x04, 0x04,   /* up arrow (0x01)         */
    0x04, 0x04, 0x04, 0x04, 0x15, 0x0E, 0x04,   /* down arrow (0x02)       */
    0x00, 0x04, 0x08, 0x1F, 0x08, 0x04, 0x00,   /* left arrow (0x03)       */
    0x00, 0x11, 0x11, 0x11, 0x1B, 0x14, 0x10,   /* greek u (mu) (0x04)     */
    0x00, 0x04, 0x02, 0x1F, 0x02, 0x04, 0x00,   /* right arrow (0x05)      */
    0x00, 0x11, 0x0A, 0x04, 0x0A, 0x11, 0x00,   /* multiply symbol (0x06)  */
    0x00, 0x04, 0x00, 0x1F, 0x00, 0x04, 0x00,   /* divide symbol (0x07)    */
    0x04, 0x04, 0x1F, 0x04, 0x04, 0x00, 0x1F,   /* plus/minus symbol (0x08) */
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,   /* UNUSED (0x09)           */
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,   /* UNUSED (0x0A)           */
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,   /* UNUSED (0x0B)           */
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,   /* UNUSED (0x0C)           */
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,   /* UNUSED (0x0D)           */
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,   /* UNUSED (0x0E)           */
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,   /* UNUSED (0x0F)           */
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,   /* UNUSED (0x10)           */
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,   /* UNUSED (0x11)           */
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,   /* UNUSED (0x12)           */
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,   /* UNUSED (0x13)           */
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,   /* UNUSED (0x14)           */
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,   /* UNUSED (0x15)           */
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,   /* UNUSED (0x16)           */
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,   /* UNUSED (0x17)           */
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,   /* UNUSED (0x18)           */
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,   /* UNUSED (0x19)           */
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,   /* UNUSED (0x1A)           */
```

```
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,   /* UNUSED (0x1B)            */
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,   /* UNUSED (0x1C)            */
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,   /* UNUSED (0x1D)            */
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,   /* UNUSED (0x1E)            */
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,   /* UNUSED (0x1F)            */
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,   /* space (0x20)             */
    0x04, 0x04, 0x04, 0x04, 0x04, 0x00, 0x04,   /* !                        */
    0x0A, 0x0A, 0x0A, 0x00, 0x00, 0x00, 0x00,   /* "                        */
    0x0A, 0x0A, 0x1F, 0x0A, 0x1F, 0x0A, 0x0A,   /* #                        */
    0x04, 0x0F, 0x14, 0x0E, 0x05, 0x1E, 0x04,   /* $                        */
    0x18, 0x19, 0x02, 0x04, 0x08, 0x13, 0x03,   /* %                        */
    0x08, 0x14, 0x14, 0x08, 0x15, 0x12, 0x0D,   /* &                        */
    0x0C, 0x0C, 0x08, 0x10, 0x00, 0x00, 0x00,   /* '                        */
    0x02, 0x04, 0x08, 0x08, 0x08, 0x04, 0x02,   /* (                        */
    0x08, 0x04, 0x02, 0x02, 0x02, 0x04, 0x08,   /* )                        */
    0x04, 0x15, 0x0E, 0x1F, 0x0E, 0x15, 0x04,   /* *                        */
    0x00, 0x04, 0x04, 0x1F, 0x04, 0x04, 0x00,   /* +                        */
    0x00, 0x00, 0x00, 0x0C, 0x0C, 0x08, 0x10,   /* ,                        */
    0x00, 0x00, 0x00, 0x1F, 0x00, 0x00, 0x00,   /* -                        */
    0x00, 0x00, 0x00, 0x00, 0x00, 0x0C, 0x0C,   /* .                        */
    0x00, 0x01, 0x02, 0x04, 0x08, 0x10, 0x00,   /* /                        */
    0x0E, 0x11, 0x13, 0x15, 0x19, 0x11, 0x0E,   /* 0                        */
    0x04, 0x0C, 0x04, 0x04, 0x04, 0x04, 0x0E,   /* 1                        */
    0x0E, 0x11, 0x01, 0x0E, 0x10, 0x10, 0x1F,   /* 2                        */
    0x0E, 0x11, 0x01, 0x06, 0x01, 0x11, 0x0E,   /* 3                        */
    0x02, 0x06, 0x0A, 0x12, 0x1F, 0x02, 0x02,   /* 4                        */
    0x1F, 0x10, 0x1E, 0x01, 0x01, 0x11, 0x0E,   /* 5                        */
    0x06, 0x08, 0x10, 0x1E, 0x11, 0x11, 0x0E,   /* 6                        */
    0x1F, 0x01, 0x02, 0x04, 0x08, 0x10, 0x10,   /* 7                        */
    0x0E, 0x11, 0x11, 0x0E, 0x11, 0x11, 0x0E,   /* 8                        */
    0x0E, 0x11, 0x11, 0x0F, 0x01, 0x02, 0x0C,   /* 9                        */
    0x00, 0x0C, 0x0C, 0x00, 0x0C, 0x0C, 0x00,   /* :                        */
    0x0C, 0x0C, 0x00, 0x0C, 0x0C, 0x08, 0x10,   /* ;                        */
    0x02, 0x04, 0x08, 0x10, 0x08, 0x04, 0x02,   /* <                        */
    0x00, 0x00, 0x1F, 0x00, 0x1F, 0x00, 0x00,   /* =                        */
    0x08, 0x04, 0x02, 0x01, 0x02, 0x04, 0x08,   /* >                        */
    0x0E, 0x11, 0x01, 0x02, 0x04, 0x00, 0x04,   /* ?                        */
    0x0E, 0x11, 0x01, 0x0D, 0x15, 0x15, 0x0E,   /* @                        */
    0x04, 0x0A, 0x11, 0x11, 0x1F, 0x11, 0x11,   /* A                        */
    0x1E, 0x09, 0x09, 0x0E, 0x09, 0x09, 0x1E,   /* B                        */
    0x0E, 0x11, 0x10, 0x10, 0x10, 0x11, 0x0E,   /* C                        */
    0x1E, 0x09, 0x09, 0x09, 0x09, 0x09, 0x1E,   /* D                        */
    0x1F, 0x10, 0x10, 0x1C, 0x10, 0x10, 0x1F,   /* E                        */
    0x1F, 0x10, 0x10, 0x1C, 0x10, 0x10, 0x10,   /* F                        */
    0x0F, 0x10, 0x10, 0x13, 0x11, 0x11, 0x0F,   /* G                        */
    0x11, 0x11, 0x11, 0x1F, 0x11, 0x11, 0x11,   /* H                        */
    0x0E, 0x04, 0x04, 0x04, 0x04, 0x04, 0x0E,   /* I                        */
    0x01, 0x01, 0x01, 0x01, 0x01, 0x11, 0x0E,   /* J                        */
    0x11, 0x12, 0x14, 0x18, 0x14, 0x12, 0x11,   /* K                        */
    0x10, 0x10, 0x10, 0x10, 0x10, 0x10, 0x1F,   /* L                        */
    0x11, 0x1B, 0x15, 0x15, 0x11, 0x11, 0x11,   /* M                        */
    0x11, 0x19, 0x15, 0x13, 0x11, 0x11, 0x11,   /* N                        */
    0x0E, 0x11, 0x11, 0x11, 0x11, 0x11, 0x0E,   /* O                        */
    0x1E, 0x11, 0x11, 0x1E, 0x10, 0x10, 0x10,   /* P                        */
    0x0E, 0x11, 0x11, 0x11, 0x15, 0x12, 0x0D,   /* Q                        */
    0x1E, 0x11, 0x11, 0x1E, 0x14, 0x12, 0x11,   /* R                        */
    0x0E, 0x11, 0x10, 0x0E, 0x01, 0x11, 0x0E,   /* S                        */
    0x1F, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04,   /* T                        */
    0x11, 0x11, 0x11, 0x11, 0x11, 0x11, 0x0E,   /* U                        */
    0x11, 0x11, 0x11, 0x0A, 0x0A, 0x04, 0x04,   /* V                        */
    0x11, 0x11, 0x11, 0x11, 0x15, 0x1B, 0x11,   /* W                        */
    0x11, 0x11, 0x0A, 0x04, 0x0A, 0x11, 0x11,   /* X                        */
    0x11, 0x11, 0x0A, 0x04, 0x04, 0x04, 0x04,   /* Y                        */
    0x1F, 0x01, 0x02, 0x04, 0x08, 0x10, 0x1F,   /* Z                        */
```

```
    0x0E, 0x08, 0x08, 0x08, 0x08, 0x08, 0x0E,   /* [                    */
    0x00, 0x10, 0x08, 0x04, 0x02, 0x01, 0x00,   /* \                    */
    0x0E, 0x02, 0x02, 0x02, 0x02, 0x02, 0x0E,   /* ]                    */
    0x04, 0x0A, 0x11, 0x00, 0x00, 0x00, 0x00,   /* ^                    */
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x1F,   /* _                    */
    0x06, 0x06, 0x04, 0x02, 0x00, 0x00, 0x00,   /* `                    */
    0x00, 0x00, 0x0E, 0x01, 0x0F, 0x11, 0x0F,   /* a                    */
    0x10, 0x10, 0x16, 0x19, 0x11, 0x19, 0x16,   /* b                    */
    0x00, 0x00, 0x0E, 0x11, 0x10, 0x11, 0x0E,   /* c                    */
    0x01, 0x01, 0x0D, 0x13, 0x11, 0x13, 0x0D,   /* d                    */
    0x00, 0x00, 0x0E, 0x11, 0x1F, 0x10, 0x0E,   /* e                    */
    0x02, 0x05, 0x04, 0x0E, 0x04, 0x04, 0x04,   /* f                    */
    0x0D, 0x13, 0x13, 0x0D, 0x01, 0x11, 0x0E,   /* g                    */
    0x10, 0x10, 0x16, 0x19, 0x11, 0x11, 0x11,   /* h                    */
    0x04, 0x00, 0x0C, 0x04, 0x04, 0x04, 0x0E,   /* i                    */
    0x01, 0x00, 0x01, 0x01, 0x01, 0x11, 0x0E,   /* j                    */
    0x10, 0x10, 0x12, 0x14, 0x18, 0x14, 0x12,   /* k                    */
    0x0C, 0x04, 0x04, 0x04, 0x04, 0x04, 0x0E,   /* l                    */
    0x00, 0x00, 0x1A, 0x15, 0x15, 0x15, 0x15,   /* m                    */
    0x00, 0x00, 0x16, 0x19, 0x11, 0x11, 0x11,   /* n                    */
    0x00, 0x00, 0x0E, 0x11, 0x11, 0x11, 0x0E,   /* o                    */
    0x16, 0x19, 0x11, 0x19, 0x16, 0x10, 0x10,   /* p                    */
    0x0D, 0x13, 0x11, 0x13, 0x0D, 0x01, 0x01,   /* q                    */
    0x00, 0x00, 0x16, 0x19, 0x10, 0x10, 0x10,   /* r                    */
    0x00, 0x00, 0x0F, 0x10, 0x0E, 0x01, 0x1E,   /* s                    */
    0x04, 0x04, 0x1F, 0x04, 0x04, 0x05, 0x02,   /* t                    */
    0x00, 0x00, 0x11, 0x11, 0x11, 0x13, 0x0D,   /* u                    */
    0x00, 0x00, 0x11, 0x11, 0x11, 0x0A, 0x04,   /* v                    */
    0x00, 0x00, 0x11, 0x11, 0x15, 0x15, 0x0A,   /* w                    */
    0x00, 0x00, 0x11, 0x0A, 0x04, 0x0A, 0x11,   /* x                    */
    0x11, 0x11, 0x11, 0x0F, 0x01, 0x11, 0x0E,   /* y                    */
    0x00, 0x00, 0x1F, 0x02, 0x04, 0x08, 0x1F,   /* z                    */
    0x02, 0x04, 0x04, 0x08, 0x04, 0x04, 0x02,   /* {                    */
    0x04, 0x04, 0x04, 0x00, 0x04, 0x04, 0x04,   /* |                    */
    0x08, 0x04, 0x04, 0x02, 0x04, 0x04, 0x08,   /* }                    */
    0x08, 0x15, 0x02, 0x00, 0x00, 0x00, 0x00,   /* ~                    */
    0x0A, 0x15, 0x0A, 0x15, 0x0A, 0x15, 0x0A    /* DEL (0x7F)           */

};
```

2.4 Analog Input

The main input of the oscilloscope is the analog signal itself. This is sampled in hardware, but the data saved in the FIFO as well as the parameters set for the analog controller must be set in software, interacting with other elements of the system.

The software can be divided into two parts, the setter and the sampler functions. The setter functions set the parameters the analog controller uses to determine when a trigger event should be generated. The arguments to these functions are from user input given via the menu. The function set_sample_rate sets the sampling frequency as a time between consecutive samples, which is used in hardware to generate a sampling clock. The function set_trigger sets both the trigger level and trigger slope of the trigger controller to the values determined by the user. Similarly, set_delay sets the trigger delay in terms of the number of sample times to wait before saving data to the FIFO. The start_sample function starts the sampling cycle by setting the type of trigger (only manual or manual and auto) as well as enabling triggers in the controller.

When the sampling based on these parameters is over, the FIFO will become full after storing 480 samples of an analog signal (the pixel in each column on the display is one sample). This will cause an interrupt in the CPU. The init_fifo_full initializes this PIO interrupt to be handled by the handle_fifo_full event handler. This handler will use the first in-first out nature of the data structure to read one byte (one sample) of data at a time, and store it in an array in the SRAM. During this process, the trigger is also disabled so that the FIFO will not be overwritten with new samples. After saving all the data from the FIFO, the Done flag is set so that it is known that a sampling cycle is complete. Then, when the main loop repeatedly calls the sampling_done function, the pointer to the buffer of data can be returned when it is available (and Done flag reset to allow for new samples to be taken).

These interactions are shown in the following diagram:

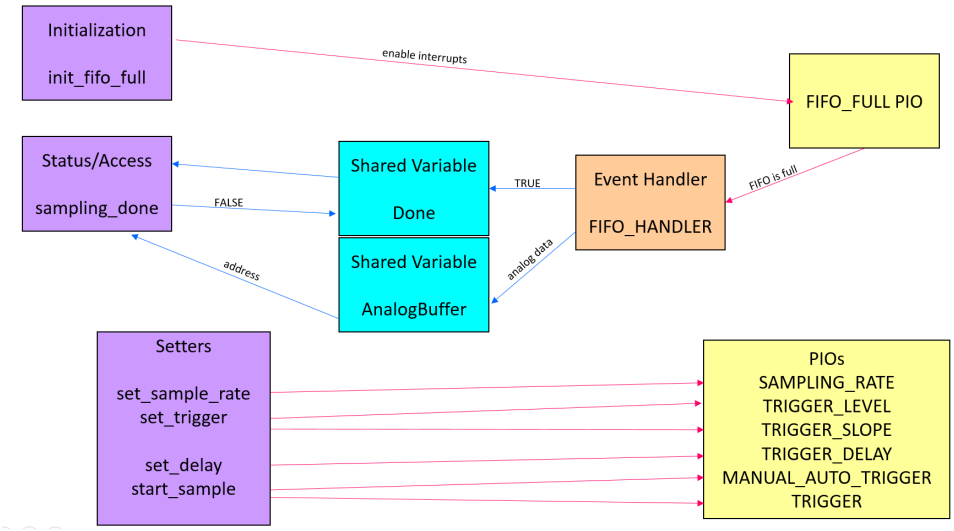


Figure 110: Interaction between functions, variables, and hardware in the analog component of software

Since along with the hardware, none of the analog software was tested, stub functions were used instead to interact with the display. For all the setter functions, none of the passed in values were set. The sample_done function was configured to return a sample of a ringing square wave. The sample rate and size of this signal were chosen experimentally for best trace on the display. In the following pages of code, the stub functions and test code are included after the actual implementations of the functions.

Files included
analog.S/analog.h - Functions to set parameters for sampling and retrieve sampled data
stubfncs.c - Stub functions for the functions in analog.S used for testing
testcode.c/testcode.h - Sample used for testing

```
////////////////////////////////////////////////////////////////////////////////
//                                                                            //
//                                   ANALOG                                   //
//                               Analog Sampling                              //
//                         Digital Oscilloscope Project                       //
//                                  EE/CS 52                                  //
//                                                                            //
////////////////////////////////////////////////////////////////////////////////


// This file contains the routines to retrieve analog data converted to 8 bit
// digital data through the analog to digital converter. This is done by reading
// the FIFO in the analog hardware, and returning a buffer of data to the main
// loop when the sample is done. The sampling parameters can also be set using
// routines in this file. These parameters are sampling rate, trigger level, delay,
// and slope. None of the functions in these file have been tested.
//
// Table of contents
// init_fifo_full - Initializes event handler for when FIFO is full
// handle_fifo_full - Copies data from analog input into buffer when FIFO
//      is full
// set_sample_rate - Set sampling rate to given value
// set_trigger - Sets trigger level and slope to given values
// set_delay - Sets trigger delay to given value
// start_sample - Start sampling data whenever there is a trigger event (or auto
//      trigger times out)
// sample_done - Checks if sample is collected and returns a pointer to the
//      sampled data
//
// Revision History:
//    06/03/17  Maitreyi Ashok      Initial revision
//    06/29/17  Maitreyi Ashok      Fixed set_sampling_rate to return a constant
//    06/30/17  Maitreyi Ashok      Updated comments



// contains analog PIO definitions
#include "system.h"
// contains constants for analog sampling parameters
#include "analog.h"
// contains definitions for working with stack
#include "macros.m"
// contains general definitions
#include "scopedef.h"

.section .text

// init_fifo_full
//
// Description:      This function initializes interrupts from the FIFO_FULL
//                   PIO as well as install the handler for it. This interrupt
//                   occurs when NUM_SAMPLES bytes have been written to the FIFO,
//                   as that many samples have been taken of the analog signal
//                   by the analog to digital converter. When the FIFO is full,
//                   the data must be saved before it is overwritten so it is
//                   important to pause the system so this can be done.
//
// Operation:        Installs the handler using the IRQ constants of the
//                   FIFO_FULL PIO in the alt_ic_isr_register function which
//                   enables hardware interrupts so that the software can use an
//                   interrupt service routine. Then, the interrupts for the used
//                   bits of this PIO are enabled and the bits used in the edge
//                   capture register for this PIO are cleared so that any pending
//                   interrupts will be turned off.
```

```
//
// Arguments:        None
// Return Value:     None
//
// Local Variables:  FIFO_PIO [r9] - contains address of the FIFO_FULL
//                          PIO register
// Shared Variables: None
// Global Variables: None
//
// Input:            None
// Output:           None
//
// Error Handling:   The function does not return until the installing of the
//                   handler using alt_ic_isr_register succeeds
//
// Algorithms:       None
// Data Structures:  None
//
// Registers Changed:r4, r5, r6, r7, r2, r8, r9
// Stack Depth:      2 words
//
// Author:           Maitreyi Ashok
// Last Modified:    06/03/17   Maitreyi Ashok      Initial revision
//                   06/30/17   Maitreyi Ashok      Updated comments

.global init_fifo_full
.align 4
.type init_fifo_full, @function

init_fifo_full:
SetUpFIFOHandler:
    movui   r4, FIFO_FULL_IRQ_INTERRUPT_CONTROLLER_ID
    movui   r5, FIFO_FULL_IRQ               // move the FIFO full interrupt id and
                                            // IRQ into registers as arguments to
                                            // the installer function
    movia   r6, handle_fifo_full            // function pointer to event handler is
                                            // another argument
    mov     r7, zero                        // pass a null pointer as isr_context
                                            // argument since it is unused
    PUSH    ra                              // store return address on stack before
                                            // calling another function
    PUSH    zero                            // store a null pointer for the flags
                                            // argument since it is also unnecessary
CallFifoHandlerSetup:
    call    alt_ic_isr_register             // install event handler for the FIFO
                                            // full set up
                                            // If installing failed, try again
    bne     r2, zero, Call_fifo_handler_setup
ReenableInterrupts:
    POP                                     // Remove argument from stack
    POP_VAL ra                              // Restore the return address
    movia   r9, FIFO_FULL_BASE              // Get FIFO full PIO register address
    ldwio   r8, 8(r9)                       // and the value in the interrupt mask
                                            // register
    ori     r8, r8, ENABLE_FIFO_INT         // Enable interrupts and store the
    stwio   r8, 8(r9)                       // enabled value to the register

    ldwio   r8, 12(r9)                      // Get value in edgecapture register
    ori     r8, r8, ENABLE_FIFO_INT         // Clear any pending interrupts and
    stwio   r8, 12(r9)                      // store cleared value into edge
                                            // capture register
    ret

// handle_fifo_full
```

```
//
// Description:      This function handles interrupts due to the FIFO filling
//                   up. Hardware interrupts due to this event will be
//                   registered in the FIFO_FULL PIO, causing this handler
//                   function to be called. TThis function reads every byte of
//                   data from the FIFO and moves it to a buffer stored in SRAM.
//                   This is done by sending a software created clock to the FIFO
//                   to read a byte of data on rising edges of the clock. When
//                   this is done, the Done flag is asserted to indicate that
//                   a full sample of analog data has been completed.
//
// Operation:        This function first disables interrupts during the handling
//                   of the event to avoid any extra events happening. Then, it
//                   disables the trigger so that no new trigger events will occur
//                   and overwrite data that has not been saved anywhere. After this,
//                   the Data_Read signal is made to go high from the PIO. When
//                   this signal goes high, the FIFO outputs the next byte of data
//                   (in first in - first out order). This data is stored in
//                   the analog buffer and buffer pointer incremented to the next
//                   memory address to store data at. Then, the data read signal
//                   is pulsed low again so that the next rising edge can have
//                   data be output from the FIFO. If the entire FIFO has not
//                   been read, the process is repeated. If the FIFO has been read
//                   completely, the Done flag is asserted since a sampling cycle
//                   has finished. Afterwards the FIFO_FULL interrupts are enabled
//                   again to allow for future hardware interrupts to be registered.
//
// Arguments:        None
// Return Value:     None
//
// Local Variables:  bufferPtr [r10] - Points to next address to write analog data
//                          to
//                   FIFO_Data [r13] - PIO address to read FIFO data from
//                   Data_Read_Addr [r16] - Address of PIO for data read clock
// Shared Variables: AnalogBuffer - Stores data read from FIFO of bytes of analog
//                          data captured
//
// Global Variables: None
//
// Input:            FIFO storing analog signal data is full. Data is read from
//                   FIFO
// Output:           None
//
// Error Handling:   None
//
// Algorithms:       None
// Data Structures:  None
//
// Registers Changed:r8, r9, r10, r13, r14, r15, r16, r17, r18
// Stack Depth:      0 words
//
// Author:           Maitreyi Ashok
// Last Modified:    06/03/17   Maitreyi Ashok      Initial revision
//                   06/30/17   Maitreyi Ashok      Updated comments

.global handle_fifo_full
.align 4
.type handle_fifo_full, @function

handle_fifo_full:
GetVarAddr:
    movia   r9, FIFO_FULL_BASE      // Store address of FIFO full PIO
    stw     zero, 8(r9)             // Disable further interrupts while handler
                                    // executes
```

```
    movia   r13, FIFO_DATA_BASE     // Store addresses of FIFO data and Data
    movia   r16, DATA_READ_BASE     // Read PIOs
    movi    r17, TRUE               // Store TRUE and FALSE constants in registers
    movi    r18, FALSE              // to be used in register-only instructions

    movia   r10, AnalogBuffer       // Store address of Analog Buffer as a
                                    // buffer pointer
DisableTrigger:
    movia   r15, TRIGGER_ENABLE_BASE    // Get address of trigger enable PIO
    stb     r18, 0(r15)                 // and disable new trigger events
CopyBuffer:
    stb     r17, 0(r16)             // Make data read signal go high
    ldbio   r14, 0(r13)             // Load a byte of data from the FIFO
    stb     r14, 0(r10)             // and store the data in the SRAM buffer
    addi    r10, r10, 1             // Increment the buffer pointer
    stb     r18, 0(r16)             // Send data read signal low
    cmpgei  r14, r10, BUFF_SIZE     // If have not reached end of buffer
    beq     r14, zero, CopyBuffer   // then repeat process for next byte
MarkAsDone:
    movia   r10, Done               // If have reached end of buffer
    stb     r17, 0(r10)             // assert that sampling is done
    movi    r17, ENABLE_FIFO_INT
    stw     r17, 8(r9)              // Re-enable interrupts for the FIFO_FULL
    ret                             // PIO

// set_sample_rate
//
// Description:      This function sets the sampling rate for the analog signal
//                   sampling to the given value. The value is given as samples
//                   per second, and this is converted into a sampling time in
//                   100s of nanoseconds. This is the scale used since the minimum
//                   sampling time is 100 nanoseconds, and other sampling times
//                   supported by the oscilloscope are multiples of 100 ns. This
//                   sampling time is stored in the output PIO for sampling
//                   rate and used in the hardware logic to make a sampling
//                   clock. The number of samples to be taken is returned, and
//                   is a fixed value (same as the size of the FIFO)
//
// Operation:        This function first converts the sampling rate passed in
//                   to a sampling time. This is done by getting the reciprocal
//                   of the sampling frequency for a time period. This is then
//                   multiplied by a constant to convert from seconds to 100s of
//                   nanoseconds. This is done in 2 steps due to the size of the
//                   multiplier being larger than a maximum 32 bit value.
//                   This value is stored at the address for the PIO in the CPU
//                   so that it can be output to be used to create a sampling
//                   clock to use for any counters in the sampling hardware.
//                   In addition, the function returns a set value as the number
//                   of samples that will be captured, NUM_SAMPLES. This does not
//                   depend on the sampling frequency, and instead the time for
//                   a complete sample of the signal will vary based on the
//                   sampling frequency.
//
// Arguments:        sampling_rate [r4] - Number of samples per second to be taken
// Return Value:     num_samples [r2] - Number of samples that will be taken at
//                          the set frequency
//
// Local Variables:  sample_time [r4] - Time for each sample in 100s of nanoseconds
// Shared Variables: None
//
// Global Variables: None
//
// Input:            None
// Output:           Sampling time (25 bits) set in hardware PIO configured for
```

```
//                          output
//
// Error Handling:   None
//
// Algorithms:       None
// Data Structures:  None
//
// Registers Changed:r2, r4, r9
// Stack Depth:      0 words
//
// Author:           Maitreyi Ashok
// Last Modified:    06/03/17   Maitreyi Ashok      Initial revision
//                   06/29/17   Maitreyi Ashok      Fixed return value to constant
//                   06/30/17   Maitreyi Ashok      Updated comments

.global set_sample_rate
.align 4
.type set_sample_rate, @function

set_sample_rate:
    movia   r9, SAMPLING_RATE_MULTIPLIER_1  // Retrieve the sampling rate
                                    // multiplier to convert from seconds/sample
                                    // to 100s of nanoseconds/sample
    div     r4, r9, r4              // Convert number of samples/second to
    muli    r4, r4, SAMPLING_RATE_MULTIPLIER_2
                                    // 100s of nanoseconds/sample by
                                    // 1/sample_rate * SAMPLING_RATE_MULITPLIER (1*2)
    movia   r9, SAMPLING_RATE_BASE  // Store this value in the PIO for the
    stw     r4, 0(r9)               // sampling rate
    movi    r2, NUM_SAMPLES         // Always take a fixed number of samples at
                                    // any frequency
    ret

// set_trigger
//
// Description:      This function sets the trigger level and trigger slope to
//                   the given values. This is done by writing to the data register
//                   of the PIO for each value. The trigger level is a value between
//                   0 and 127 (all possible 7 bit unsigned values) that represents
//                   a voltage between MIN_LEVEL and MAX_LEVEL (defined in interfac.h).
//                   The trigger slope is 1 for a negative trigger slope
//                   (meaning a trigger event will happen if the trigger level
//                   is passed when the signal is sloping downwards). Conversely,
//                   the trigger slope is 0 for a positive trigger slope (trigger
//                   event occurs if the trigger level is passed with the signal
//                   increasing).
//
// Operation:        This function sets the trigger level and slope. This is done
//                   by taking each of the arguments to the function, as well as
//                   the base address of each PIO. Then, the argument is written
//                   to the data register (offset 0) of the PIO, which is set
//                   for output from the CPU. These values are then used in the
//                   analog controller to generate trigger events manually.
//
// Arguments:        trig_level [r4] - Trigger level to set for sampling (between
//                         0 and 127)
//                   trig_slope [r5] - Trigger slope to set for sampling (1 for
//                         negative slope, 0 for positive slope)
// Return Value:     None
//
// Local Variables:  None
// Shared Variables: None
//
// Global Variables: None
```

```
//
// Input:            None
// Output:           Trigger level (7 bits) and trigger slope (1 bit) set in hardware
//                         PIOs configured for output
// Error Handling:   None
//
// Algorithms:       None
// Data Structures:  None
//
// Registers Changed:r9
// Stack Depth:      0 words
//
// Author:           Maitreyi Ashok
// Last Modified:    06/03/17   Maitreyi Ashok      Initial revision
//                   06/30/17   Maitreyi Ashok      Updated comments

.global set_trigger
.align 4
.type set_trigger, @function

set_trigger:
    movia   r9, TRIGGER_LEVEL_BASE  // Store the given trigger level in the
    sth     r4, 0(r9)               // appropriate output PIO
    movia   r9, TRIGGER_SLOPE_BASE  // Store the given trigger slope in the
    sth     r5, 0(r9)               // appropriate output PIO
    ret

// set_delay
//
// Description:      This function sets the trigger delay to the given value.
//                   This is done by writing to the data register of the PIO
//                   TRIGGER_DELAY. The trigger delay is a value between
//                   0 and 50000 that represents the number of samples to wait
//                   between a trigger event and saving the signal data to the FIFO.
//                   Once either a manual trigger event occurs due to the trigger
//                   level being passed with the correct slope or from an auto
//                   trigger timeout,a counter using a clock at the sampling
//                   frequency will count up to this trigger delay before latching
//                   the write request signal to the FIFO.
//
// Operation:        This function sets the trigger delay. This is done
//                   by taking the argument to the function, as well as
//                   the base address of the PIO. Then, the argument is written
//                   to the data register (offset 0) of the PIO, which is set
//                   for output from the CPU. These values are then used in the
//                   analog controller to generate a write request signal.
//
// Arguments:        trig_delay [r4] - Trigger delay in units of samples
// Return Value:     None
//
// Local Variables:  None
// Shared Variables: None
//
// Global Variables: None
//
// Input:            None
// Output:           Trigger delay (16 bits) set in hardware PIO set up for output
//
// Error Handling:   None
//
// Algorithms:       None
// Data Structures:  None
//
// Registers Changed:r9
```

```
// Stack Depth:      0 words
//
// Author:           Maitreyi Ashok
// Last Modified:    06/03/17   Maitreyi Ashok      Initial revision
//                   06/30/17   Maitreyi Ashok      Updated comments

.global set_delay
.align 4
.type set_delay, @function

set_delay:
    movia   r9, TRIGGER_DELAY_BASE  // Store the given trigger delay in the
    stw     r4, 0(r9)               // appropriate output PIO
    ret

// start_sample
//
// Description:      This function immediately starts sampling data using the analog
//                   controller. This is done by either manual triggering or auto
//                   trigger timeout, depending on the setting chosen by the caller
//                   function. If the caller passes in TRUE, then auto trigger
//                   timeout can be used to start saving data to the FIFO. If the
//                   caller passes in FALSE, then only manual trigger events allow
//                   for data to start being saved to the FIFO. In addition, to
//                   start waiting for a trigger, the trigger enable signal is
//                   made active.
//
// Operation:        This function starts sampling in the hardware by both setting
//                   how a trigger event can be generated and enabling trigger
//                   events in general. The address of the manual/auto trigger
//                   option is saved, and the choice of the caller function is
//                   written to the data register. Since the argument is TRUE if
//                   auto trigger can be used, but the PIO uses TRUE if only manual
//                   trigger can be used, the value of the argument is inverted
//                   before being saved. Inverting is equivalent to XORing with
//                   TRUE, since if the argument is TRUE, the result will be FALSE.
//                   If the argument is FALSE, then the result will be TRUE since
//                   there will be an odd amount of TRUEs being XORed.
//
// Arguments:        auto_trigger [r4] - Whether auto trigger timeout can be used
//                   to start sampling (TRUE) or a trigger event must occur (FALSE)
// Return Value:     None
//
// Local Variables:  None
// Shared Variables: None
//
// Global Variables: None
//
// Input:            None
// Output:           Manual or auto trigger setting is set in a hardware PIO. Trigger
//                   events are also enabled in an output hardware PIO. The analog
//                   controller will start waiting for a trigger event due to
//                   manual trigger or auto trigger timeout (if enabled)
//
// Error Handling:   None
//
// Algorithms:       None
// Data Structures:  None
//
// Registers Changed:r4, r9, r15, r18
// Stack Depth:      0 words
//
// Author:           Maitreyi Ashok
// Last Modified:    06/03/17   Maitreyi Ashok      Initial revision
```

```
//                  06/30/17   Maitreyi Ashok      Updated comments

.global start_sample
.align 4
.type start_sample, @function

start_sample:
AutoTriggerPIO:
    movia   r9, MANUAL_AUTO_TRIGGER_BASE // Store the manual vs. auto trigger
    xori    r4, r4, TRUE                 // Invert manual vs auto trigger value
    sth     r4, 0(r9)                    // option in the appropriate output PIO
    movi    r18, TRUE
    movia   r15, TRIGGER_ENABLE_BASE     // Enable triggering so that a new sample
    stb     r18, 0(r15)                  // can be collected and stored in the FIFO

    ret

// sample_done
//
// Description:      This function returns the sampled data once the sampling of
//                   the analog signal is finished. If the sampling is complete,
//                   then the pointer to the sampled data is returned. If the
//                   sampling is not complete, then a NULL pointer is returned.
//                   If the sampling is complete (Done flag set at end of FIFO
//                   full event handler), then the pointer can be set to the return
//                   value. In addition, the Done flag is reset so that the
//                   function returns a non-NULL pointer once for each call to
//                   the start_sample function.
//
// Operation:        This function first retrieves the value of the Done flag from
//                   memory. If the value of the flag is not TRUE, or is FALSE
//                   then the sampling is not complete and a NULL pointer is
//                   returned. If the value of the flag is TRUE, sampling is complete
//                   so the done flag is reset so this function will only return
//                   a pointer to the buffer once for each time the sampling is
//                   started. In addition, the return value is set to the address
//                   of the AnalogBuffer stored in the data section. The size of
//                   the AnalogBuffer is the number of bytes returned by the
//                   sample_rate setting function.
//
// Arguments:        None
// Return Value:     bufferAddr [r2] - Pointer to buffer storing FIFO data, NULL
//                          if sampling is not complete
//
// Local Variables:  Done_Addr [r9] - address of Done flag stored in memory
// Shared Variables: Done - flag storing whether sampling is complete (TRUE) or
//                          not (FALSE)
//                   AnalogBuffer - buffer of data stored from the FIFO of
//                          analog data
//
// Global Variables: None
//
// Input:            None
// Output:           None
//
// Error Handling:   None
//
// Algorithms:       None
// Data Structures:  None
//
// Registers Changed:r2, r9, r10, r11
// Stack Depth:      0 words
//
// Author:           Maitreyi Ashok
```

```
// Last Modified:    06/03/17   Maitreyi Ashok      Initial revision
//                   06/30/17   Maitreyi Ashok      Updated comments

.global sample_done
.align 4
.type sample_done, @function

sample_done:
    movia   r9, Done                // Find the value of the Done flag
    ldb     r11, 0(r9)
    cmpeqi  r10, r11, TRUE          // If the Done flag is not TRUE, then
    beq     r10, zero, NotDone      // a sample is not done completely
DoneSample:
    movi    r10, FALSE              // If the sample is done completely, then
    stb     r10, 0(r9)              // the done flag is reset so a sample is only
                                    // read once from the buffer
    movia   r2, AnalogBuffer        // Return the address of the AnalogBuffer
                                    // so the data can be read and displayed
    jmpi    EndDoneSample
NotDone:
    mov     r2, zero                // No sample complete, so a NULL pointer is
                                    // returned
EndDoneSample:
    ret


.section .data
.align 4
Done:           .byte FALSE             // Stores whether a sample has been
                .skip 1                 // completed with the data stored in the
                                        // AnalogBuffer so it cannot be overwritten
                                        // in the FIFO
AnalogBuffer:   .byte 0                 // Stores a copy of data from the FIFO
                .skip NUM_SAMPLES       // collected from the analog to digital
                                        // converter after a trigger event has
                                        // occurred or auto trigger timeout
```

```
/*************************************************************************/
/*                                                                       */
/*                                 ANALOG.H                              */
/*                            Analog Definitions                         */
/*                              Include File                             */
/*                       Digital Oscilloscope Project                    */
/*                                 EE/CS 52                              */
/*                                                                       */
/*************************************************************************/

/*
   This file contains the constants for interfacing with the analog hardware. It
   contains a mask for the FIFO PIO in the CPU, as well as constants for the
   various parts of analog sampling.

   Revision History:
      06/03/17  Maitreyi Ashok      Initial revision
      06/29/17  Maitreyi Ashok      Added NUM_SAMPLES constant
      06/30/17  Maitreyi Ashok      Updated comments
*/

#ifndef  __ANALOG_H__
    #define  __ANALOG_H__


#define     ENABLE_FIFO_INT             0x0001
#define     NUM_SAMPLES                 480
#define     BUFF_SIZE                   NUM_SAMPLES
#define     SAMPLING_RATE_MULTIPLIER_1  0x2710
#define     SAMPLING_RATE_MULTIPLIER_2  0x03E8

#endif
```

```
/*************************************************************************/
/*                                                                       */
/*                                STUBFNCS                               */
/*                        Oscilloscope Stub Functions                    */
/*                        Digital Oscilloscope Project                   */
/*                                 EE/CS 52                              */
/*                                                                       */
/*************************************************************************/

/*
   This file contains stub functions for the hardware interfacing code for the
   Digital Oscilloscope project.  The file is meant to allow linking of the
   main code without necessarily having all of the low-level functions or
   hardware working.  The functions included are:
      key_available   - check if a key is available
      getkey          - get a key
      clear_display   - clear the display
      plot_pixel      - plot a pixel
      set_sample_rate - set the sample rate
      set_trigger     - set the trigger level and slope
      set_delay       - set the trigger delay
      start_sample    - start sampling
      sample_done     - sampling status

   The local functions included are:
      none

   The locally global variable definitions included are:
      none


   Revision History
      3/8/94   Glen George       Initial revision.
      3/13/94  Glen George       Updated comments.
      3/13/94  Glen George       Changed set_sample_rate to return SIZE_X.
      5/9/06   Glen George       Updated start_sample stub to match the new
                                 specification.
      6/16/17  Maitreyi Ashok    Updated sample_done to return test code generated
                                 sample
*/



/* library include files */
  /* none */

/* local include files */
#include  "interfac.h"
#include  "scopedef.h"

static int trg_level;
static int old_trg_level;

void init_analog()
{
    trg_level = 0;
}
/* sampling parameter functions */

int  set_sample_rate(long int rate)
{
    return  SIZE_X;
}
```

```
void  set_trigger(int level, int slope)
{
    return;
}


void  set_delay(long int delay)
{
    return;
}



/* sampling functions */

void  start_sample(int auto_trigger)
{
    return;
}

unsigned char *sample_done()
{
    // Allocate a buffer to store the sample and get sample data
    unsigned char *sample = malloc(SIZE_X*sizeof(unsigned char));
    get_test_sample(SIZE_X, SIZE_X*5, sample);
    return sample;
}
```

```
/*****************************************************************************/
/*                                                                           */
/*                                TESTCODE                                   */
/*                             Test Functions                                */
/*                       Digital Oscilloscope Project                        */
/*                                EE/CS 52                                   */
/*                                                                           */
/*****************************************************************************/

/*
   This file contains test functions for the Digital Oscilloscope project.
   These functions are just used to test various features of the system
   without necessarily needing all the hardware.  The functions included are:
      get_test_sample - get a test sample

   The local functions included are:
      none

   The locally global variable definitions included are:
      none


   Revision History
      3/13/94  Glen George       Initial revision.
      3/17/97  Glen George       Updated comments.
      5/3/06   Glen George       Added inclusion of scopedef.h to pick up
                                 definitions for portability.
      5/3/06   Glen George       Updated function to handle new, faster sweep
                                 rates.
*/



/* library include files */
  /* none */

/* local include files */
#include  "interfac.h"
#include  "scopedef.h"
#include  "testcode.h"




/* local definitions */
#define  NO_TEST_PTS    (sizeof(waveform) / sizeof(unsigned char))
                                              /* size of test waveform */




/*
   get_test_sample

   Description:      This function simulates getting a sample of the passed
             size at the passed rate.  The sample is returned via the
             passed pointer (3rd argument).  The sample rate is
             assumed to be less than 10000000.  The waveform returned
             is a ringing squarewave at approximately 521 Hz.

   Arguments:        sample_rate (long int)       - the number of samples per
                               second to simulate.
             sample_size (int)            - the number of samples to
                                 be returned.
```

```
            sample (unsigned char far *) - pointer to the location to
                                which to return the
                               simulated samples.
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       The sample point variable (sample_pt) is actually ten
             times the sample point number to allow for fractional
             sweep rate ratios without integer truncation problems.
   Data Structures:  None.

   Global Variables: None.

   Author:           Glen George
   Last Modified:    May 3, 2006

*/

void  get_test_sample(long int sample_rate, int sample_size, unsigned char *sample)
{
    /* variables */

    /* a half-cycle of the ringing square wave */
    const static unsigned char  waveform[] =
        {  128, 127, 126, 125, 124, 122, 120, 118, 116, 113, 111, 108, 105,
       102,  99,  96,  93,  90,  87,  83,  80,  77,  74,  70,  67,  64,
        61,  58,  55,  52,  49,  47,  44,  42,  40,  37,  36,  34,  32,
        31,  29,  28,  27,  26,  26,  25,  25,  25,  25,  25,  25,  26,
        26,  27,  28,  29,  30,  31,  33,  34,  36,  37,  39,  41,  42,
        44,  46,  48,  50,  52,  54,  56,  58,  60,  62,  64,  66,  68,
        69,  71,  73,  74,  76,  77,  79,  80,  81,  82,  83,  84,  85,
        86,  86,  87,  87,  88,  88,  88,  88,  88,  88,  88,  87,  87,
        86,  86,  85,  85,  84,  83,  82,  81,  80,  79,  78,  77,  76,
        75,  74,  72,  71,  70,  69,  68,  66,  65,  64,  63,  62,  61,
        60,  59,  58,  57,  56,  55,  54,  54,  53,  52,  52,  51,  51,
        50,  50,  50,  50,  50,  50,  50,  50,  50,  50,  50,  50,  51,
        51,  51,  52,  52,  53,  54,  54,  55,  55,  56,  57,  57,  58,
        59,  60,  60,  61,  62,  63,  63,  64,  65,  65,  66,  67,  67,
        68,  68,  69,  69,  70,  70,  71,  71,  71,  72,  72,  72,  72,
        73,  73,  73,  73,  73,  73,  73,  73,  72,  72,  72,  72,  72,
        71,  71,  71,  70,  70,  70,  69,  69,  68,  68,  68,  67,  67,
        66,  66,  65,  65,  64,  64,  64,  63,  63,  62,  62,  62,  61,
        61,  61,  60,  60,  60,  60,  59,  59,  59,  59,  59,  59,  59,
        59,  59,  59,  59,  59,  59,  59,  59,  59,  59,  59,  60,  60,
        60,  60,  60,  61,  61,  61,  61,  62,  62,  62,  62,  63,  63,
        63,  63,  64,  64,  64,  65,  65,  65,  65,  65,  66,  66,  66,
        66,  66,  66,  67,  67,  67,  67,  67,  67,  67,  67,  67,  67,
        67,  67,  67,  67,  67,  67,  67,  67,  67,  67,  67,  66,  66,
        66,  66,  66,  66,  66,  65,  65,  65,  65,  65,  65,  64,  64,
        64,  64,  64,  64,  64,  63,  63,  63,  63,  63,  63,  63,  63,
        62,  62,  62,  62,  62,  62,  62,  62,  62,  62,  62,  62,  62,
        62,  62,  62,  62,  62,  62,  62,  62,  62,  63,  63,  63,  63,
        63,  63,  63,  63,  63,  63,  63,  64,  64,  64,  64,  64,  64,
        64,  64,  64,  64,  64,  65,  65,  65,  65,  65,  65,  65,  65,
        65,  65,  65,  65,  65,  65,  65,  65,  65,  65,  65,  65,  65,
        65,  65,  65,  65,  65,  65,  65,  65,  65,  65,  65,  65,  65,
        65,  65,  64,  64,  64,  64,  64,  64,  64,  64,  64,  64,  64,
        64,  64,  64,  64,  64,  64,  64,  63,  63,  63,  63,  63,  63,
        63,  63,  63,  63,  63,  63,  63,  63,  63,  63,  63,  63,  63,
        63,  63,  63,  63,  63,  63,  64,  64,  64,  64,  64,  64,  64,
```

```
        64,  64,  64,  64,  64,  64,  64,  64,  64,  64,  64,  64
      };

    long int  sample_pt = 0;    /* current sample within the waveform */

    int       sample_sign = 1;  /* sample waveform sign */



    /* loop, getting sample points */
    while (sample_size-- > 0)  {

        /* get this sample point (value based on sign) */
    if (sample_sign > 0)
        *sample++ = waveform[sample_pt/20] + 127;
    else
        *sample++ = 128 - waveform[sample_pt/20];

    /* compute the time for the next sample point */
    sample_pt += (10000000L / sample_rate);
    /* keep sample_pt within the waveform (need to scale it) */
    while (sample_pt >= (20 * NO_TEST_PTS))  {
        /* keep sample_pt within the test waveform */
        sample_pt -= (20 * NO_TEST_PTS);
        /* each time through the waveform, invert it */
        sample_sign = -sample_sign;
        }
    }


    /* done getting the sample - return */
    return;

}
```

```
/*****************************************************************************/
/*                                                                           */
/*                                 TESTCODE.H                                */
/*                              Test Functions                               */
/*                               Include File                                */
/*                        Digital Oscilloscope Project                       */
/*                                 EE/CS 52                                  */
/*                                                                           */
/*****************************************************************************/

/*
   This file contains the constants and function prototypes for the Digital
   Oscilloscope project test functions defined in testcode.c.


   Revision History:
      3/13/94  Glen George       Initial revision.
*/



#ifndef  __TESTCODE_H__
    #define  __TESTCODE_H__



/* library include files */
  /* none */

/* local include files */
  /* none */




/* constants */
    /* none */




/* structures, unions, and typedefs */
    /* none */




/* function declarations */

/* get a simulated test sample */
void  get_test_sample(long int, int, unsigned char *);


#endif
```

2.5 Other General

The main component of the software that brings all the previous components together is the main loop. The main loop interfaces with these components, retrieving key events from the rotary encoder and processing them, sampling data from the analog component, and updating the display with the new trace and menu changes. The file is included below. In addition, general definitions for the oscilloscope are located in scopedef.h and constants for interfacing between the C code and assembly code and hardware are located in interfac.h. Finally, macros for pushing and popping values off the stack are located in macros.m. This is used in all the assembly files for both passing arguments and storing the return address of a function when using a nested function call.

```
/*****************************************************************************/
/*                                                                           */
/*                                  MAINLOOP                                 */
/*                              Main Program Loop                            */
/*                         Digital Oscilloscope Project                      */
/*                                  EE/CS 52                                 */
/*                                                                           */
/*****************************************************************************/

/*
   This file contains the main processing loop (background) for the Digital
   Oscilloscope project.  The only global function included is:
      main - background processing loop

   The local functions included are:
      key_lookup - get a key and look up its keycode

   The locally global variable definitions included are:
      none


   Revision History
      3/8/94   Glen George       Initial revision.
      3/9/94   Glen George       Changed initialized const arrays to static
                                 (in addition to const).
      3/9/94   Glen George       Moved the position of the const keyword in
                                 declarations of arrays of pointers.
      3/13/94  Glen George       Updated comments.
      3/13/94  Glen George       Removed display_menu call after plot_trace,
                                 the plot function takes care of the menu.
      3/17/97  Glen George       Updated comments.
      3/17/97  Glen George       Made key_lookup function static to make it
                 truly local.
      3/17/97  Glen George       Removed KEY_UNUSED and KEYCODE_UNUSED
                 references      (no longer used).
      5/27/08  Glen George       Changed code to only check for sample done if
                                 it is currently sampling.
      6/15/17  Maitreyi Ashok    Fixed compilation errors
      6/16/17  Maitreyi Ashok    Added initialization of rotary encoder and
                                 fifo full PIO interrupts
*/



/* library include files */
  /* none */

/* local include files */
#include  "interfac.h"
#include  "scopedef.h"
#include  "keyproc.h"
#include  "menu.h"
#include  "tracutil.h"



/* local function declarations */
static enum keycode  key_lookup(void);   /* translate key values into keycodes */
extern void Init_Rot(void);              /* initialize rotary encoder interrupts */
extern void init_fifo_full(void);        /* initialize FIFO full interrupt */
/*
   main

   Description:      This procedure is the main program loop for the Digital
```

```
                    Oscilloscope.  It loops getting keys from the keypad,
                    processing those keys as is appropriate.  It also handles
                    starting scope sample collection and updating the LCD
              screen.

   Arguments:       None.
   Return Value:    (int) - return code, always 0 (never returns).

   Input:           Keys from the keypad.
   Output:          Traces and menus to the display.

   Error Handling:  Invalid input is ignored.

   Algorithms:      The function is table-driven.  The processing routines
                    for each input are given in tables which are selected
                    based on the context (state) the program is operating in.
   Data Structures: Array (process_key) to associate keys with actions
             (functions to call).

   Global Variables: None.

   Author:          Glen George
   Last Modified:   May 27, 2008

*/

int  main()
{
    /* variables */
    enum keycode        key;            /* an input key */

    enum status         state = MENU_ON;    /* current program state */

    unsigned char *sample;          /* a captured trace */

    /* key processing functions (one for each system state type and key) */
    static enum status  (* const process_key[NUM_KEYCODES][NUM_STATES])(enum status) =
       /*   Current System State                       */
       /*  MENU_ON      MENU_OFF             Input Key  */
      { {  menu_key,    menu_key    },   /* <Menu>      */
        {  menu_up,     no_action   },   /* <Up>        */
        {  menu_down,   no_action   },   /* <Down>      */
        {  menu_left,   no_action   },   /* <Left>      */
        {  menu_right,  no_action   },   /* <Right>     */
        {  no_action,   no_action   } }; /* illegal key */



    /* first initialize everything */
    clear_display();        /* clear the display */

    init_trace();       /* initialize the trace routines */
    init_menu();        /* initialize the menu system */

    Init_Rot();         /* initialize rotary encoder PIO interrupt */
    init_fifo_full();   /* initialize FIFO full PIO interrupt */
    /* infinite loop processing input */
    while(TRUE)  {

        /* check if ready to do a trace */
    if (trace_rdy())
        /* ready for a trace - do it */
        do_trace();
    display_menu();
```

```
    /* check if have a trace to display */
    if (is_sampling() && ((sample = sample_done()) != NULL))  {

        /* have a trace - output it */
        plot_trace(sample);
        /* done processing this trace */
        trace_done();
    }


    /* now check for keypad input */
    if (key_available())  {

        /* have keypad input - get the key */
        key = key_lookup();

        /* execute processing routine for that key */
        state = process_key[key][state](state);
    }
    }


    /* done with main (never should get here), return 0 */
    return  0;

}




/*
   key_lookup

   Description:      This function gets a key from the keypad and translates
                     the raw keycode to an enumerated keycode for the main
                     loop.

   Arguments:        None.
   Return Value:     (enum keycode) - type of the key input on keypad.

   Input:            Keys from the keypad.
   Output:           None.

   Error Handling:   Invalid keys are returned as KEYCODE_ILLEGAL.

   Algorithms:       The function uses an array to lookup the key types.
   Data Structures:  Array of key types versus key codes.

   Global Variables: None.

   Author:           Glen George
   Last Modified:    Mar. 17, 1997

*/

static  enum keycode  key_lookup()
{
    /* variables */

    const static enum keycode  keycodes[] = /* array of keycodes */
        {                           /* order must match keys array exactly */
           KEYCODE_MENU,      /* <Menu>     */   /* also need an extra element */
       KEYCODE_UP,        /* <Up>       */   /* for unknown key codes */
```

```
      KEYCODE_DOWN,      /* <Down>     */
      KEYCODE_LEFT,      /* <Left>     */
      KEYCODE_RIGHT,     /* <Right>    */
      KEYCODE_ILLEGAL    /* other keys */
       };

   const static int  keys[] =   /* array of key values */
       {            /* order must match keycodes array exactly */
          KEY_MENU,    /* <Menu>     */
      KEY_UP,      /* <Up>       */
      KEY_DOWN,    /* <Down>     */
      KEY_LEFT,    /* <Left>     */
      KEY_RIGHT,   /* <Right>    */
       };

   int  key;       /* an input key */

   int  i;             /* general loop index */



   /* get a key */
   key = get_key();


   /* lookup key in keys array */
   for (i = 0; ((i < (sizeof(keys)/sizeof(int))) && (key != keys[i])); i++);


   /* return the appropriate key type */
   return  keycodes[i];

}
```

```
/*****************************************************************************/
/*                                                                           */
/*                               SCOPEDEF.H                                  */
/*                          General Definitions                              */
/*                              Include File                                 */
/*                      Digital Oscilloscope Project                         */
/*                                EE/CS 52                                   */
/*                                                                           */
/*****************************************************************************/

/*
   This file contains the general definitions for the Digital Oscilloscope
   project.  This includes constant and structure definitions along with the
   function declarations for the assembly language functions.


   Revision History:
      3/8/94   Glen George       Initial revision.
      3/13/94  Glen George       Updated comments.
      3/17/97  Glen George       Removed KEYCODE_UNUSED (no longer used).
      5/3/06   Glen George       Added conditional definitions for handling
                                 different architectures.
      5/9/06   Glen George       Updated declaration of start_sample() to
                                 match the new specification.
      5/27/08  Glen George       Added check for __nios__ definition to also
                                 indicate the compilation is for an Altera
                     NIOS CPU.
*/



#ifndef  __SCOPEDEF_H__
    #define  __SCOPEDEF_H__


/* library include files */
  /* none */

/* local include files */
#include  "interfac.h"
#include  "lcdout.h"




/* constants */

/* general constants */
#define  FALSE       0
#define  TRUE        !FALSE
#define  NULL        (void *) 0

/* display size (in characters) */
#define  LCD_WIDTH   (SIZE_X / HORIZ_SIZE)
#define  LCD_HEIGHT  (SIZE_Y / VERT_SIZE)




/* macros */

/* let __nios__ also mean a NIOS compilation */
#ifdef  __nios__
  #define  NIOS         /* use the standard NIOS defintion */
```

```
#endif

/* add the definitions necessary for the Altera NIOS chip */
#ifdef  NIOS
  #define  FLAT_MEMORY      /* use the flat memory model */
#endif


/* if a flat memory model don't need far pointers */
#ifdef  FLAT_MEMORY
  #define  far
#endif




/* structures, unions, and typedefs */

/* program states */
enum status  {  MENU_ON,    /* menu is displayed with the cursor in it */
        MENU_OFF,   /* menu is not displayed - no cursor */
        NUM_STATES  /* number of states */
         };

/* key codes */
enum keycode  {  KEYCODE_MENU,      /* <Menu>     */
             KEYCODE_UP,        /* <Up>       */
             KEYCODE_DOWN,      /* <Down>     */
             KEYCODE_LEFT,      /* <Left>     */
             KEYCODE_RIGHT,     /* <Right>    */
             KEYCODE_ILLEGAL,   /* other keys */
         NUM_KEYCODES       /* number of key codes */
              };




/* function declarations */

/* keypad functions */
unsigned char  key_available(void);     /* key is available */
char get_key(void);                     /* retrieve key event */

/* display functions  */
void  clear_display(void);                /* clear the display */
void  plot_pixel(unsigned int, unsigned int, int);    /* output a pixel */

/* sampling parameter functions */
int   set_sample_rate(long int);    /* set the sample rate */
void  set_trigger(int, int);        /* set trigger level and slope */
void  set_delay(long int);      /* set the trigger delay time */

/* sampling functions */
void          start_sample(int);    /* capture a sample */
unsigned char *sample_done(void);   /* sample captured status */


#endif
```

```
/*****************************************************************************/
/*                                                                           */
/*                                INTERFAC.H                                 */
/*                           Interface Definitions                           */
/*                               Include File                                */
/*                        Digital Oscilloscope Project                       */
/*                                 EE/CS 52                                  */
/*                                                                           */
/*****************************************************************************/

/*
   This file contains the constants for interfacing between the C code and
   the assembly code/hardware for the Digital Oscilloscope project.  This is
   a sample interface file to allow compilation of the .c files.


   Revision History:
      3/8/94   Glen George       Initial revision.
      3/13/94  Glen George       Updated comments.
      3/17/97  Glen George       Added constant MAX_SAMPLE_SIZE and removed
                                 KEY_UNUSED.
      4/17/17  Maitreyi Ashok    Changed keypad constants to match PIO inputs
*/



#ifndef  __INTERFAC_H__
    #define  __INTERFAC_H__


/* library include files */
  /* none */

/* local include files */
  /* none */




/* constants */

/* keypad constants */
#define  KEY_MENU       4   /* <Menu>      */
#define  KEY_UP         2   /* <Up>        */
#define  KEY_DOWN       1   /* <Down>      */
#define  KEY_LEFT       16  /* <Left>      */
#define  KEY_RIGHT      8   /* <Right>     */
#define  KEY_EXTRA      32  /* <Extra Key>*/
#define  KEY_ILLEGAL    6   /* illegal key */

/* display constants */
#define  SIZE_X         480 /* size in the x dimension */
#define  SIZE_Y         272 /* size in the y dimension */
#define  PIXEL_BLUE    0xf0 /* blue pixel RGB code */
#define  PIXEL_BLACK   0x00 /* black (dark blue) pixel RGB code */
#define  PIXEL_WHITE   0xff /* white pixel RGB code */
#define  PIXEL_PINK    0x44 /* pink pixel RGB code */
#define  PIXEL_GREEN   0x03 /* green pixel RGB code */

/* scope parameters */
#define  MIN_DELAY         0    /* minimum trigger delay */
#define  MAX_DELAY     50000    /* maximum trigger delay */
#define  MIN_LEVEL    -12000    /* minimum trigger level (in mV) */
#define  MAX_LEVEL     12000    /* maximum trigger level (in mV) */
```

```
/* sampling parameters */
#define  MAX_SAMPLE_SIZE   2400 /* maximum size of a sample (in samples) */


#endif
```

```
////////////////////////////////////////////////////////////////////////////////
//                                                                            //
//                                   MACROS                                   //
//                          Digital Oscilloscope Project                      //
//                                   EE/CS 52                                 //
//                                                                            //
////////////////////////////////////////////////////////////////////////////////

// This file contains the macros to allow for pushing and popping values from
// the stack
//
// Table of contents
// PUSH     - push a value from a register to stack
// POP_VAL  - pop a value from the stack into a register
// POP      - pop a value from the stack (does not need to be saved)
//
// Revision History:
//      5/15/17  Maitreyi Ashok  Initial Revision
//      6/30/17  Maitreyi Ashok  Updated comments

// PUSH
//
// Description: This macro pushes a value from the given register to
//              the stack.
// Operation:   The stack pointer is first moved to the new top of stack.
//              Since the stack grows upwards, this moves the stack pointer
//              up by 4 bytes (size of a register). The register is then stored
//              at the top of the stack.
// Arguments:   reg - value to push to stack
// Author:      Maitreyi Ashok
// Modified:    May 15, 2017

.macro PUSH reg
subi sp, sp, 4      // Move stack pointer up
stw \reg, 0(sp)     // Store value at new top of stack
.endm

// POP_VAL
//
// Description: This macro pops a value from the stack to a given register.
// Operation:   The value is first saved from the top of stack (where the stack
//              pointer points) to the given register. The stack pointer is then
//              moved to the new top of stack. Since the stack grows upwards,
//              this moves the stack pointer down by 4 bytes (size of a register).
// Arguments:   reg - location to pop value from stack into
// Author:      Maitreyi Ashok
// Modified:    May 15, 2017

.macro POP_VAL reg
ldw \reg, 0(sp)     // Store value from top of stack into register
addi sp, sp, 4      // Move stack pointer down
.endm

// POP
//
// Description: This macro removes a value from the stack.
// Operation:   The stack pointer is moved to the new top of stack. Since the
//              stack grows upwards and we wish to remove a value, this moves the
//              stack pointer down by 4 bytes (size of a register). Since there
//              is no need to save the value from the stack, it is not written to
//              a register
// Arguments:   None
// Author:      Maitreyi Ashok
// Modified:    May 15, 2017
```

```
.macro POP
addi sp, sp, 4      // Move stack pointer down
.endm
```