

 GLAST LAT SYSTEM SPECIFICATION	Document #	LAT-SS-00255-01	Date Effective
	Prepared by(s)	W. Neil Johnson	Supersedes None
	Subsystem/Office	Calorimeter Subsystem	
Document Title	Calorimeter Crystal Optical Test Bench ADC Control Design Description		

DRAFT D1

Gamma-ray Large Area Space Telescope (GLAST)
Large Area Telescope (LAT)
Calorimeter Crystal Optical Test Bench
ADC Control Design Description

CHANGE HISTORY LOG

Revision	Effective Date	Description of Changes	DCN #
1		Initial Release	

1 PURPOSE

This document describes the mechanical and electrical characteristics of the CsI Test Bench ADC Control Electronics for the Calorimeter subsystem of the GLAST Large Area Telescope (LAT). This document specifies the design of the Calorimeter (Cal) Crystal Optical Testing Station (COTS) ADC control electronics for the GLAST Large Area Telescope (LAT).

This board is housed in a single width NIM module and contains the electronics to control and readout up to four NIM ADCs, control one NIM dual high voltage power supply and transfer data to a PC.

2 DEFINITIONS

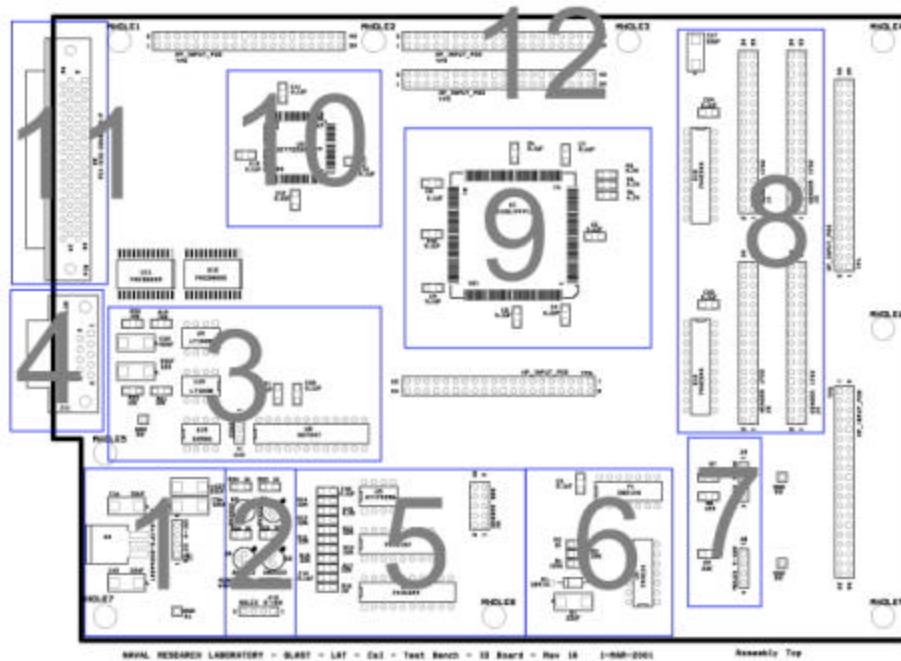
2.1 Acronyms

GLAST	Gamma-ray Large Area Space Telescope
LAT	Large Area Telescope
CAL	The Calorimeter subsystem of the LAT
TBR	To Be Resolved

2.2 Definitions

γ	Gamma Ray
μsec , μs	microsecond, 10^{-6} second
nm	nanometer
μm	micrometer
mm	millimeter
cm	centimeter
eV	Electron Volt
MeV	Million Electron Volts, 10^6 eV
ph	photons

3 APPLICABLE DOCUMENTS



4 Board Construction

5 Board Layout

5.1 Board Introduction

The IO board has twelve sections, explained below.

5.2 Section 1: Power

Power is drawn from a standard NIM box power supply. The IO board uses +6 (70mA), +12 (27mA), -12 (17mA) and ground. The +6V is passed through a National Semiconductor LM2940CS-5.0 1A Low Dropout Regulator to power the +5V power plane. The +12 and -12 are used to control the High Voltage control circuitry.

Voltage	Current
+6 V	70 mA
+12 V	27 mA
-12 V	17 mA

5.3 Section 2: HV Inhibit

The High Voltage (HV) Inhibit line is monitored and controlled by this circuitry. This line works in serial with the black box door safety relay. The HV Inhibit line is normally high, high voltage is on. When either the door relay or the IO board grounds the line, the HV is turned off.

5.4 Section 3: HV Value Selector

When the Canberra Model 3125 Dual HV Power Supply is in remote mode, the HV value can be controlled externally. The ratio is -1V program input to 1kV output, up to the supply max of 5k. For this application, the HV supply should have the limit set to 2kV max. The AD7247 is a dual 12bit DAC used to set the 0 to -5 voltages.

5.5 Section 4: HV Interface

The HV connector on the IO board is a DB9 pin connector. The HV supply has a Winchester (<http://litton-wed.com/rap/m-1.html>) connector, which uses a mating M9PLS plug. A H16 aluminum hood is used with the Winchester connector.

Pin DB9	Pin M9PLS	Function
1	A	Unit A (1) Inhibit
2	B	Unit A (1) Reference
3	C	Unit A (1) Program
4	D	Unit A (1) Ground
5	E	No connection
6	F	Unit B (2) Ground
7	H	Unit B (2) Program
8	J	Unit B (2) Reference
9	K	Unit B (2) Inhibit

5.6 Section 5: Xilinx PROM and control

The Xilinx EEPROM, ATMEL AT17C256, is a re-programmable serial memory. The EEPROM loads the Xilinx at power up.

5.7 Section 6: Power Reset and System Clock

This section contains the 10MHz clock and the power-up program circuitry for the Xilinx. The power-up circuitry holds the Xilinx program line low for a few milliseconds while the power supply stabilizes. Upon the program line going high, the Xilinx reads the EEPROM (see Section 5).

5.8 Section 7: LED Output and Test Switch

This section contains a header for the power and event LED. These and two other LEDs are mounted on the front panel.

LED Name	Header Number	Color	Function
Power	J9	Green	6 Volt NIM Bin Power
Trigger	J9	Yellow	Flashes for event readout
High Voltage A	J10	Red	Lighted for HV A enabled
High Voltage B	J10	Red	Lighted for HV B enabled

The Test Switch is used for debugging only. The normal configuration is for pins 2 and 3 to be shorted.

5.9 Section 8: Four ADC Inputs

The IO board has inputs for connecting up to 4 Canberra Model 8701 ADC inputs. A 34-pin conductor ribbon cable connects between each ADC module and a keyed header on the IO board. Since there was not enough space to mount the ADC connectors on the outside of the module, a slot at the rear is provided to enable the cables to pass through and make direct connection to the board.

ADC Number	Header Number
ADC 0	J1
ADC 1	J2
ADC 2	J3
ADC 3	J4

5.10 Section 9: FPGA/Xilinx

A Xilinx XC4010E FPGA is used to perform the control. The FPGA is programmed using the Verilog hardware description language. The Verilog code listing is attached.

5.11 Section 10: FIFO

An IDT 72285L10PF is used as the interface between the Xilinx and the PC. This is a high-speed synchronous memory used to buffer data flow.

5.12 Section 11: PC Interface

The PC interface is intended to be used with the National Instruments PCI-DIO-32HS board of the 653X series using the SH68-68 cable (<http://www.ni.com>). On the specifics for each product and protocols (Burst and Unstrobed), please see documentation from National Instrument.

The PCI-DIO-32HS has a 32-bit interface and 8 bits of control. The 32 IO lines are divided into two groups of 16. The first Group is for data collections and the second is for commanding.

Group 1 is configured for the Burst Protocol. The Burst Protocol is a synchronous (clocked) protocol. The IDT72285L10PF (<http://www.idt.com>) FIFO, in First Word Fall Through (FWFT) mode, was specifically chosen for this synchronous protocol. Since this is a 653X input operation, the default is for the 653X to generate the PCLK signal. The other two control lines used are REQ and ACK. The REQ is tied, with some logic, to the FIFO empty/output ready flag and the ACK is tied, with some logic, to the FIFO read enable.

Group 2 is configured for the Unstrobed protocol. Using the Unstrobed protocol frees the four control lines for group 2 to be used as extra data lines. The output lines, PCLK2 and ACK2, are used. When PCLK2 goes high, data is written into the IO board's register designated by the 2 most significant bits of the data word. ACK2 should be high during this operation. When ACK2 is low, the register selected by the IO board's MASK register, is placed on the data bus for the 653X to read.

Using this configuration separates the data collection from the commanding. Commanding cannot be hindered by the data collection, since the data is being collected in the background by DMA. Likewise data collection does not have to stop for commanding.

5.13 Section 12: Six Test Connectors

These are for debug only, see the schematic for pin functions.

Header Number	Function
TP1	ADC control lines
TP2	ADC data bus
TP3	FIFO control lines
TP4	FIFO data in bus
TP5	FIFO data out bus
TP6	Register bus

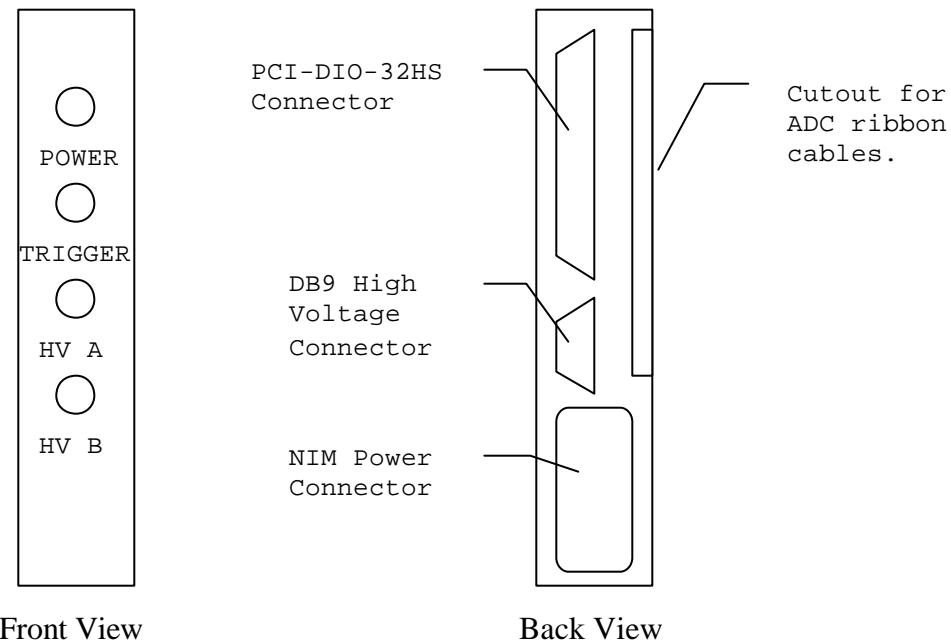
5.14 Board Modifications

There are two board modifications needed.

- 1) The first is a wire must be connected between TP3 pin 29 to U1 pin 43 (PCLK2).
- 2) The second, the hole for U15 pin 2 must be drilled out and then the socket pin is connected to the +12 volt supply at J7 pin 3.

6 Hardware

6.1 Front and Back View



6.2 Cables

There are three different cables used with the IO board. There is the ADC 34 pin ribbon cable, the HV cable described in section 4 and the PCI-DIO-32HS cable.

7 Verilog Code

7.1 Listing

```
//*****
module glastlatiox(
clk,sw1,led1,spare1,spare2,spare3,
req1_n,ack1_n,pclk1,ack2_n,pclk2,
dac_csa_n,dac_csb_n,dac_wr_n,hvinhibit,
reg_d,fi_d,adc_d,
rclk,wclk,ren_n,wen_n,
fwft,ld_n,mrs_n,prs_n,rt_n,sen_n,oe_n,
ef_n,/*pae_n,*/hf_n,paf_n,ff_n
,adc0_acept,adc0_endata,adc0_cdt,adc0_enc,adc0_ready,adc0_inb//,adc0_bf,adc0_blld,adc0_bcb
,adc1_acept,adc1_endata,adc1_cdt,adc1_enc,adc1_ready,adc1_inb//,adc1_bf,adc1_blld,adc1_bcb
,adc2_acept,adc2_endata,adc2_cdt,adc2_enc,adc2_ready,adc2_inb//,adc2_bf,adc2_blld,adc2_bcb
,adc3_acept,adc3_endata,adc3_cdt,adc3_enc,adc3_ready,adc3_inb//,adc3_bf,adc3_blld,adc3_bcb
);

//*****
// OPERATIONAL
wire GSR;
input clk;
input sw1;
output led1;
reg led1;
```

```
output spare1;
output spare2;
output spare3;
// PCI IO
output req1_n;
input ack1_n;
input pclk1;
input ack2_n;
input pclk2;
// DAC
output dac_csa_n;
output dac_csb_n;
output dac_wr_n;
// HV
output hv inhibit;
// REGISTER
inout [15:0] reg_d;
reg [15:0] reg_d;
// FIFO
output [15:0] fi_d;
reg [15:0] fi_d;
output rclk;
output wclk;
output fwft;
output ld_n;
output mrs_n;
output prs_n;
output ren_n;
output rt_n;
output sen_n;
output wen_n;
output oe_n;
input ef_n;
wire or_n;
wire ir_n;
//input pae_n;
input hf_n;
input paf_n;
input ff_n;
reg wen_n;
//reg wclk;
// ADC COMMON
input [12:0] adc_d;
wire [12:0] adc_d_n;
// ADC0
output adc0_acept;
output adc0_endata;
input adc0_cdt;
output adc0_enc;
input adc0_ready;
input adc0_inb;
//input adc0_bf;
//input adc0_billd;
//input adc0_bcb;
// ADC1
output adc1_acept;
output adc1_endata;
input adc1_cdt;
output adc1_enc;
input adc1_ready;
input adc1_inb;
//input adc1_bf;
//input adc1_billd;
//input adc1_bcb;
// ADC2
output adc2_acept;
output adc2_endata;
input adc2_cdt;
output adc2_enc;
input adc2_ready;
input adc2_inb;
//input adc2_bf;
//input adc2_billd;
//input adc2_bcb;
// ADC3
output adc3_acept;
output adc3_endata;
input adc3_cdt;
output adc3_enc;
input adc3_ready;
input adc3_inb;
```

```

//input adc3_bf;
//input adc3_blld;
//input adc3_bccb;

//*****
reg [15:0] reg_0; // (r ) data
reg [ 5:0] reg_1; // (rw0) csr
reg [15:0] reg_2; // (rw1) mask
reg [15:0] reg_3; // (r ) live time low
reg [15:0] reg_4; // (r ) live time high
reg [12:0] reg_5; // (r ) time low
reg [12:0] reg_6; // (r ) time high
reg [15:0] reg_7; // (r ) acq
reg [15:0] reg_8; // (rw2) dac0
reg [15:0] reg_9; // (rw3) dac1
wire dodata;
wire done;
wire GSR_done;
wire adc0_rdy;
wire adc1_rdy;
wire adc2_rdy;
wire adc3_rdy;

//*****

assign GSR = sw1;
assign spare1 = 1'b0;
assign spare2 = 1'b0;
assign spare3 = 1'b0;

assign or_n = ef_n;
assign ir_n = ff_n;
assign adc_d_n = ~adc_d;

//*****
// ***** Write DAC *****

assign dac_wr_n = (ack2_n == 1'b1 && pclk2 && reg_d[15] == 1'b1) ? 1'b0 : 1'b1;
assign dac_csa_n = (ack2_n == 1'b1 && pclk2 && reg_d[15:13] == 3'b100) ? 1'b0 : 1'b1;
assign dac_csb_n = (ack2_n == 1'b1 && pclk2 && reg_d[15:13] == 3'b110) ? 1'b0 : 1'b1;

// ***** Write Register *****

always @(posedge pclk2 or posedge GSR)
if (GSR) begin reg_1 = 6'b000000; reg_2 = 16'h0000; reg_8 = 16'h0000; reg_9 = 16'h0000; end
else
begin
  case (reg_d[15:14])
    2'b00:   reg_1 = reg_d[5:0];
    2'b01:   reg_2 = reg_d;
    2'b10:   reg_8 = reg_d;
    2'b11:   reg_9 = reg_d;
  endcase
end

// ***** Read Register *****

always @((ack2_n or reg_2[13:10] or reg_0 or reg_1 or reg_2 or reg_3 or reg_4 or reg_5 or reg_6 or reg_7 or reg_8
or reg_9 or adc3_rdy or adc2_rdy or adc1_rdy or adc0_rdy or ir_n or hf_n or or_n))
begin
  case ({ack2_n,reg_2[13:10]})
    5'b00000: reg_d = reg_0;
    5'b00001: reg_d = {2'b00,adc3_rdy,adc2_rdy,adc1_rdy,adc0_rdy,ir_n,~hf_n,~or_n,1'b0,reg_1[5:0]};
    5'b00010: reg_d = reg_2;
    5'b00011: reg_d = reg_3;
    5'b00100: reg_d = reg_4;
    5'b00101: reg_d = {3'b000,reg_5};
    5'b00110: reg_d = {3'b000,reg_6};
    5'b00111: reg_d = reg_7;
    5'b01000: reg_d = reg_8;
    5'b01001: reg_d = reg_9;
  default:   reg_d = 16'hZZZZ;
  endcase
end

assign hv inhibit = ~reg_2[9];

// ***** Read FIFO *****

```

```

assign oe_n = 1'b0;
assign rt_n = 1'b1;
assign req1_n = ((or_n == 1'b0) && (reg_1[0] == 1'b1 || reg_2[8] == 1'b1)) ? 1'b0 : 1'b1;
assign ren_n = ((or_n == 1'b0) && (reg_1[0] == 1'b1 || reg_2[8] == 1'b1) && (ack1_n == 1'b0)) ? 1'b0 : 1'b1;
assign rclk = pclk1;

// *****
// *****
// *****
// *****
// ***** Enable ADCs *****

assign adc0_enc = ~(reg_1[0] & reg_2[0]);
assign adc1_enc = ~(reg_1[0] & reg_2[1]);
assign adc2_enc = ~(reg_1[0] & reg_2[2]);
assign adc3_enc = ~(reg_1[0] & reg_2[3]);

// ***** Event Detection *****

wire adc0_ecdt;
wire adc1_ecdt;
wire adc2_ecdt;
wire adc3_ecdt;
wire adc_or_ecdt;
reg new_event;

assign adc0_ecdt = (~adc0_cdt & reg_2[0]);
assign adc1_ecdt = (~adc1_cdt & reg_2[1]);
assign adc2_ecdt = (~adc2_cdt & reg_2[2]);
assign adc3_ecdt = (~adc3_cdt & reg_2[3]);

assign adc_or_ecdt = (adc0_ecdt | adc1_ecdt | adc2_ecdt | adc3_ecdt);

always @(posedge adc_or_ecdt or posedge GSR_done)
  if (GSR_done) begin new_event = 1'b0; led1 = 1'b1; end
  else begin new_event = 1'b1; led1 = 1'b0; end

// ***** Coincidence Interval Timer *****

reg [7:0] ctim_max;
reg [7:0] ctim_cnt;
wire ctim_done;
wire ctim_early;

always @(reg_1)
  case (reg_1[4:3])
    2'b00: ctim_max = 20; // 20 = 2usec
    2'b01: ctim_max = 40; // 40 = 4usec
    2'b10: ctim_max = 80; // 80 = 8usec
    2'b11: ctim_max = 160; // 160 = 16usec
  endcase

assign ctim_done = (ctim_cnt >= ctim_max);
assign ctim_early = (ctim_cnt >= 40);

always @(posedge clk or posedge GSR_done)
  if (GSR_done) ctim_cnt = 0;
  else if (!ctim_done && new_event) ctim_cnt = ctim_cnt + 1;

// ***** Coincidence Detection *****

wire adc0_coin;
wire adc1_coin;
wire adc2_coin;
wire adc3_coin;
wire adc_and_coin;
reg itis_early;
wire adc_and_coin_and_not_early;
reg valid_event;
reg reject_event;

assign adc0_coin = (~adc0_cdt & reg_2[4]) | ~reg_2[4];
assign adc1_coin = (~adc1_cdt & reg_2[5]) | ~reg_2[5];
assign adc2_coin = (~adc2_cdt & reg_2[6]) | ~reg_2[6];
assign adc3_coin = (~adc3_cdt & reg_2[7]) | ~reg_2[7];

assign adc_and_coin = (adc0_coin & adc1_coin & adc2_coin & adc3_coin);

always @(posedge ctim_early or posedge GSR_done)
  if (GSR_done) begin itis_early = 1'b0; end

```

```

else           begin itis_early = adc_and_coin; end
assign adc_and_coin_and_not_early = adc_and_coin & ~itis_early;

always @(posedge ctim_done or posedge GSR_done)
  if (GSR_done) begin valid_event = 1'b0;          reject_event = 1'b0;      end
  else           begin valid_event = adc_and_coin_and_not_early; reject_event = ~adc_and_coin_and_not_early; end

// ***** Event Timeout counter *****

reg [10:0] event_timeout_cnt;
wire event_timeout_done;

assign event_timeout_done = (event_timeout_cnt >= 1000);

always @(posedge clk or posedge GSR_done)
  if (GSR_done) event_timeout_cnt = 0;
  else if (!event_timeout_done && valid_event) event_timeout_cnt = event_timeout_cnt + 1;

// ***** Invalid Data *****

wire adc0_einb;
wire adc1_einb;
wire adc2_einb;
wire adc3_einb;
wire adc_or_einb;

assign adc0_einb = (~adc0_inb & reg_2[0]);
assign adc1_einb = (~adc1_inb & reg_2[1]);
assign adc2_einb = (~adc2_inb & reg_2[2]);
assign adc3_einb = (~adc3_inb & reg_2[3]);

assign adc_or_einb = (adc0_einb | adc1_einb | adc2_einb | adc3_einb);

// ***** Ready Detection *****

wire adc_and_rdy;
reg fifo_load;

assign adc0_rdy = (~adc0_ready & reg_2[0]) | ~reg_2[0];
assign adc1_rdy = (~adc1_ready & reg_2[1]) | ~reg_2[1];
assign adc2_rdy = (~adc2_ready & reg_2[2]) | ~reg_2[2];
assign adc3_rdy = (~adc3_ready & reg_2[3]) | ~reg_2[3];

assign adc_and_rdy = (adc0_rdy & adc1_rdy & adc2_rdy & adc3_rdy);

always @(posedge adc_and_rdy or posedge GSR_done)
  if (GSR_done) fifo_load = 1'b0;
  else           fifo_load = 1'b1;

// *****
// ***** Fake FIFO Data Fixed Amount *****
// *****

reg [25:0] fake_cnt;
wire clk_fake;
wire GSR_fake;
wire fake_done;
wire fake_do;

assign clk_fake = (dodata & reg_2[8] & ~reg_1[0]);
assign GSR_fake = (GSR | ~reg_2[8] | reg_1[0]);

assign fake_done = (fake_cnt >= 100);

always @(posedge clk_fake or posedge GSR_fake)
  if (GSR_fake)      fake_cnt = 0;
  else if (!fake_done) fake_cnt = fake_cnt + 1;

assign fake_do = (reg_2[8] & ~fake_done & ~reg_1[0] & ~GSR_done);

// ***** Start State Machine *****

wire start_sm;
reg fifo_write_n;
reg dofake;

assign start_sm = (fifo_load == 1'b1 || fake_do == 1'b1);

always @(posedge start_sm or posedge GSR_done)
  if (GSR_done) begin fifo_write_n = 1'b1; dofake = 1'b0; end

```

```

else begin fifo_write_n = ((adc_or_einb & fifo_load) | ~paf_n); dofake = fake_do; end
// ***** FIFO Load State Machine *****

parameter [15:0] // synopsys enum STATE_TYPE
STATE_000 = 16'h0000,
STATE_001 = 16'h0001,
STATE_002 = 16'h0002,
STATE_003 = 16'h0004,
STATE_004 = 16'h0008,
STATE_005 = 16'h0010,
STATE_006 = 16'h0020,
STATE_007 = 16'h0040,
STATE_008 = 16'h0080,
STATE_009 = 16'h0100,
STATE_010 = 16'h0200,
STATE_011 = 16'h0400,
STATE_012 = 16'h0800,
STATE_013 = 16'h1000,
STATE_014 = 16'h2000,
STATE_015 = 16'h4000,
STATE_016 = 16'h8000;

reg [15:0] /* synopsys enum STATE_TYPE */ state;
reg [15:0] /* synopsys enum STATE_TYPE */ next_state;
// synopsys state_vector state

always @(posedge clk or posedge GSR)
if (GSR) state = STATE_000;
else state = next_state;

always @(state or reject_event or event_timeout_done or start_sm or reg_1 or reg_2)
begin
next_state = state;
case (state) // synopsys full_case
STATE_000: next_state = STATE_001;
STATE_001: next_state = STATE_002;
STATE_002: next_state = STATE_003;
STATE_003:
if (reject_event == 1'b1 || event_timeout_done == 1'b1) next_state = STATE_015;
else if (start_sm) next_state = STATE_004;
STATE_004: next_state = (reg_1[5] == 1'b1) ? STATE_005 : STATE_007;
STATE_005: next_state = STATE_006;
STATE_006: next_state = STATE_007;
STATE_007: next_state = STATE_008;
STATE_008: next_state = STATE_009;
STATE_009: next_state = STATE_010;
STATE_010: next_state = (reg_2[2] == 1'b1 || reg_2[3] == 1'b1) ? STATE_011 : STATE_015;
STATE_011: next_state = STATE_012;
STATE_012: next_state = STATE_013;
STATE_013: next_state = STATE_014;
STATE_014: next_state = STATE_015;
STATE_015: next_state = STATE_016;
STATE_016: next_state = STATE_003;
default: next_state = STATE_003;
endcase
end

always @(posedge clk or posedge GSR)
if (GSR)
begin wen_n = 1'b1;
fi_d = 16'h0000; end
else if (state == STATE_000) begin wen_n = 1'b1;
fi_d = 16'h0000; end
else if (state == STATE_001) begin wen_n = 1'b1;
fi_d = 16'h0000; end
else if (state == STATE_002) begin wen_n = 1'b1;
fi_d = 16'h0000; end
else if (state == STATE_003) begin wen_n = 1'b1;
fi_d = 16'h0000; end
else if (state == STATE_004) begin wen_n = 1'b1;
fi_d = 16'h0000; end
else if (state == STATE_005) begin wen_n = fifo_write_n;
fi_d = (dofake == 1'b1) ?
{3'b110,fake_cnt[12: 0]} : {3'b110,reg_5}; end
else if (state == STATE_006) begin wen_n = fifo_write_n;
fi_d = (dofake == 1'b1) ?
{3'b111,fake_cnt[25:13]} : {3'b111,reg_6}; end
else if (state == STATE_007) begin wen_n = 1'b1;
fi_d = {3'b000,adc_d_n}; end
else if (state == STATE_008) begin wen_n = (fifo_write_n | ~reg_2[0]);
fi_d = {3'b000,adc_d_n}; end
else if (state == STATE_009) begin wen_n = 1'b1;
fi_d = {3'b001,adc_d_n}; end
else if (state == STATE_010) begin wen_n = (fifo_write_n | ~reg_2[1]);
fi_d = {3'b001,adc_d_n}; end
else if (state == STATE_011) begin wen_n = 1'b1;
fi_d = (dofake == 1'b1) ?
{3'b010,13'b0010101010101} : {3'b010,adc_d_n}; end
else if (state == STATE_012) begin wen_n = (fifo_write_n | ~reg_2[2]);
fi_d = (dofake == 1'b1) ?
{3'b010,13'b0010101010101} : {3'b010,adc_d_n}; end
else if (state == STATE_013) begin wen_n = 1'b1;
fi_d = (dofake == 1'b1) ?
{3'b011,13'b11111111111111} : {3'b011,adc_d_n}; end
else if (state == STATE_014) begin wen_n = (fifo_write_n | ~reg_2[3]);
fi_d = (dofake == 1'b1) ?
{3'b011,13'b11111111111111} : {3'b011,adc_d_n}; end

```

```

else if (state == STATE_015) begin wen_n = 1'b1; fi_d = 16'h0000; end
else if (state == STATE_016) begin wen_n = 1'b1; fi_d = 16'h0000; end
else begin wen_n = 1'b1; fi_d = 16'h0000; end

assign wclk = ~clk;
assign adc0_endata = (state == STATE_007 || state == STATE_008) ? 1'b0 : 1'b1;
assign adc1_endata = (state == STATE_009 || state == STATE_010) ? 1'b0 : 1'b1;
assign adc2_endata = (state == STATE_011 || state == STATE_012) ? 1'b0 : 1'b1;
assign adc3_endata = (state == STATE_013 || state == STATE_014) ? 1'b0 : 1'b1;

assign dodata = (state == STATE_007) ? 1'b1 : 1'b0;
assign done = (state == STATE_016) ? 1'b1 : 1'b0;
assign GSR_done = (GSR | done);
assign adc0_acept = ~done;
assign adc1_acept = ~done;
assign adc2_acept = ~done;
assign adc3_acept = ~done;

// ***** Setup FIFO *****
assign fwft = 1'b1;
assign sen_n = 1'b1;
assign ld_n = (state == STATE_000 || state == STATE_001 || state == STATE_002) ? 1'b0 : 1'b1;

assign mrs_n = (state == STATE_001) ? 1'b0 : 1'b1;
assign prs_n = (state == STATE_003 && reg_1[1] == 1'b1) ? 1'b0 : 1'b1;

// ***** 1msec/1kHz clock *****
reg [15:0] jj;
wire jj_done;
reg dtclk;
wire edtclk;

assign jj_done = (jj == 5000) ? 1'b1 : 1'b0;

always @(posedge clk or posedge GSR)
  if (GSR) jj = 0;
  else jj = (jj_done) ? 0 : jj + 1;

always @(posedge jj_done or posedge GSR)
  if (GSR) dtclk = 1'b0;
  else dtclk = ~dtclk;

assign edtclk = (reg_1[0] & dtclk);

// ***** Live Time Counter *****
wire reset_livetime;
wire not_dead;

assign reset_livetime = (GSR | reg_1[2]);
assign not_dead = ~adc_or_ecdt;

always @(posedge edtclk or posedge reset_livetime)
  if (reset_livetime) {reg_4,reg_3} = 32'h00000000;
  else if (not_dead) {reg_4,reg_3} = {reg_4,reg_3} + 1;

// ***** Event Time Counter *****
always @(posedge edtclk or posedge GSR)
  if (GSR) {reg_6,reg_5} = 26'h00000000;
  else {reg_6,reg_5} = {reg_6,reg_5} + 1;

//*****
endmodule
//*****

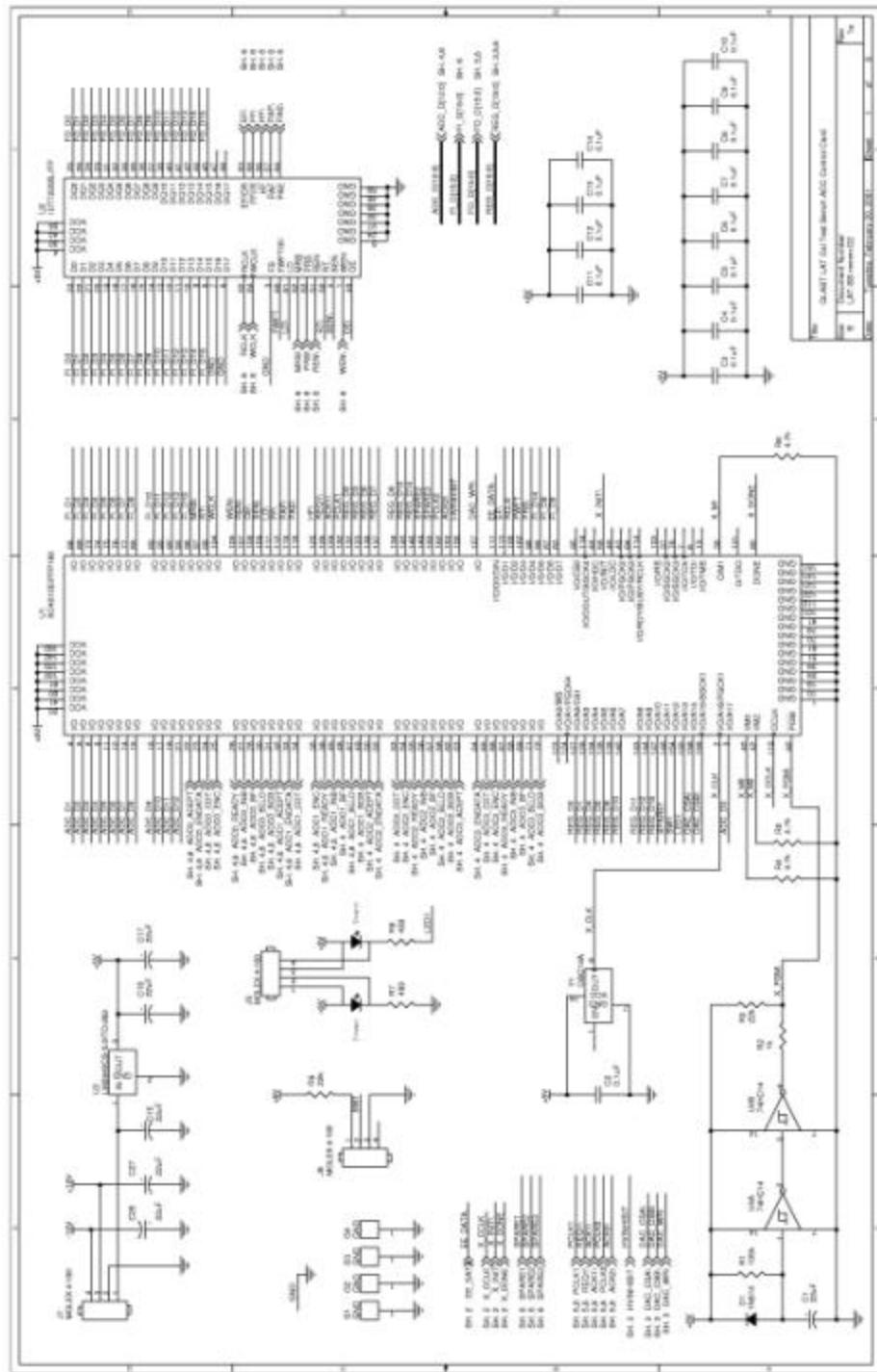
```

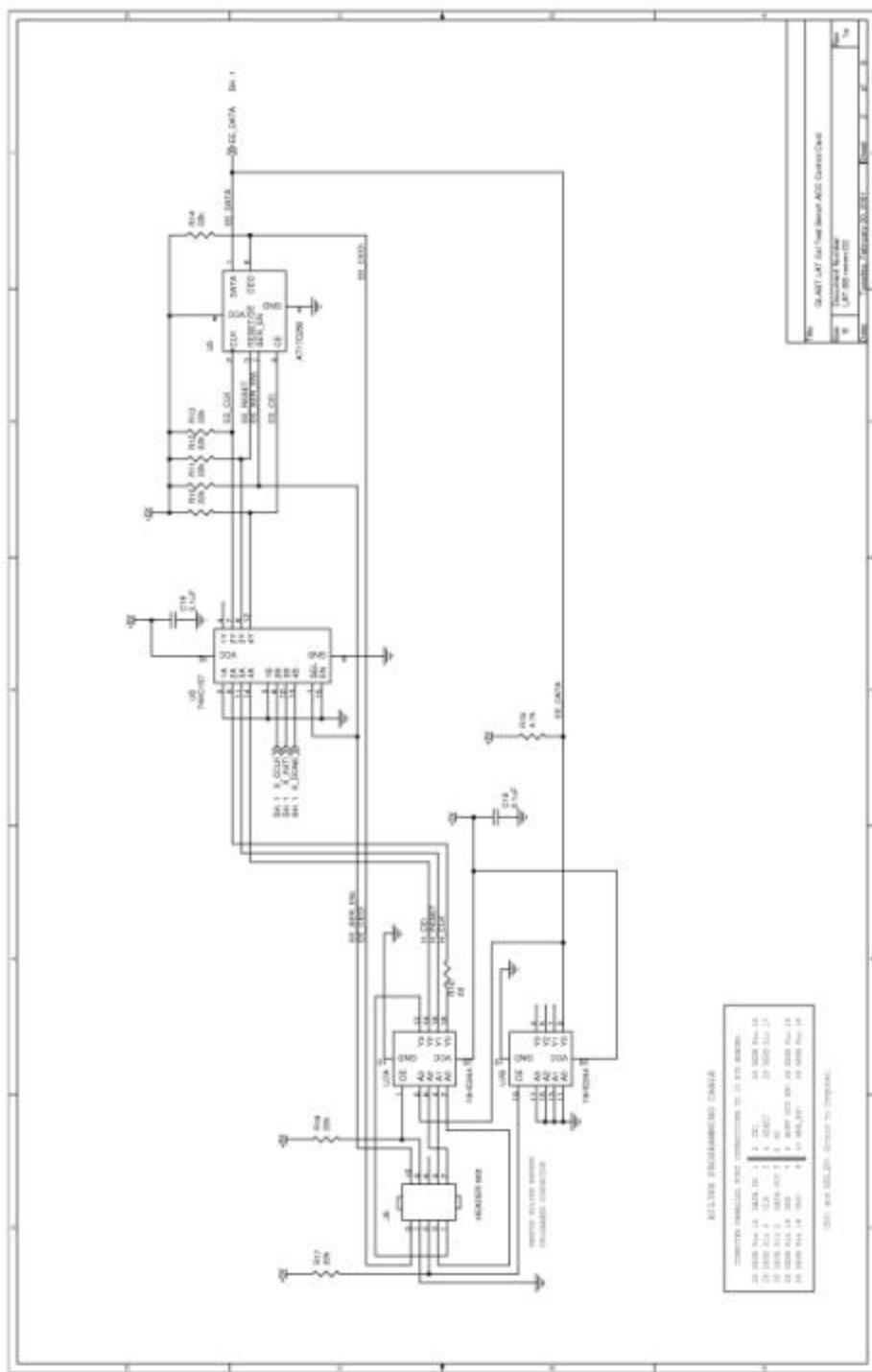
8 Parts List

Part	http://www.digikey.com	Model Number	Digi-Key Item Number
22uF			PCS5226CT-ND
0.1uF			311-1179-1-ND
1N4148			
Header 17x2			MHB34K-ND
PCI-DIO-32HS 68-P			2-174225-5-ND
Header 5x2			MHB10K-ND
MOLEX 4 pin	http://www.molex.com	22-23-2041	WM4202-ND
MOLEX 4 socket		22-01-3047	WM2002-ND
MOLEX crimp sockets		08-50-0114	WM2200-ND
MOLEX break hdr vert		22-28-4360	22-28-4360-ND
DB9 PLUG RT			A2096-ND
2N2222			2N2222AMS-ND
100k			P100KFCT-ND
1k			P1.00KFCT-ND
22k			P22.1KFCT-ND
4.7k			P4.75KFCT-ND
499			P499FCT-ND
22			P22.1FCT-ND
10k			P10.0KFCT-ND
Header 20x2			MHB40K-ND
XC4010E	http://www.xilinx.com/		122-1103-ND
IDT72285L10PF	http://www.idt.com/		
LM2940CS-5.0/TO-263			LM2940CS-5.0-ND
74HC14			296-1577-5-ND
AT17C256			AT17C256-10PC-ND
74HC157			296-1578-5-ND
74HC244			296-1582-5-ND
AD7247			
LT1028			LT1028ACN8-ND

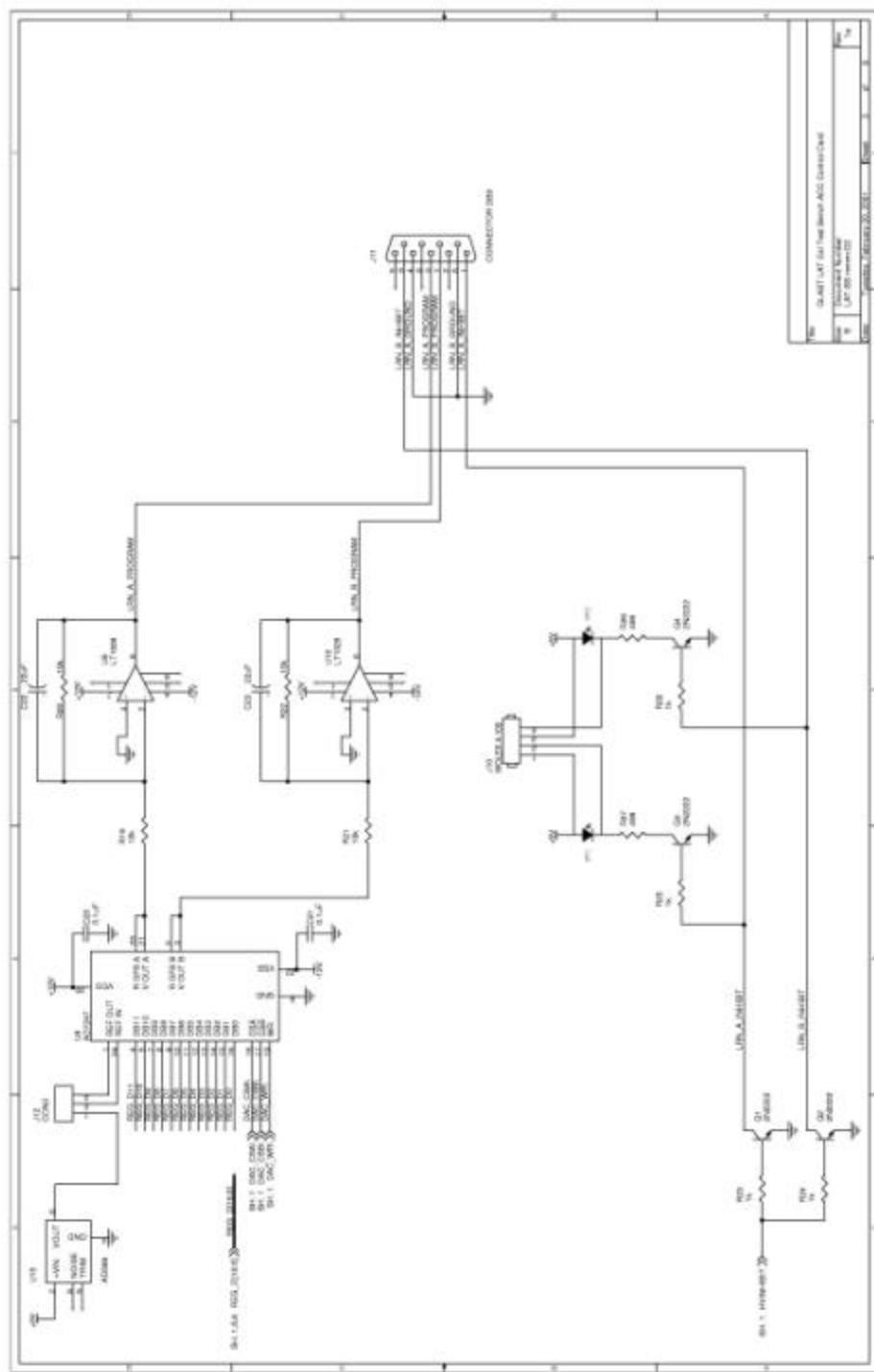
PACDN005			
AD586			
OSC 10MHZ			MX045-10.0000
SOCKET DIP 8			A400-ND
SOCKET DIP 14			A401-ND
SOCKET DIP 16			A402AE-ND
SOCKET DIP 20			A404-ND
SOCKET DIP 24			A405-ND
9 Pin Cable (Cut in half)			AE1016-ND
34 Pin w/ Pol Key			AKC34G-ND
PN:GLAST LAT IO BD	http://www.alltekcircuit.com/		ALLTEK CIRCUIT
LED 8MM RED			67-1167-ND
LED 8MM GRN			67-1168-ND
LED 8MM YEL			67-1169-ND
	http://www.mcmaster.com/		McMaster Item Number
#4-40 Hex Jack Screws			92710A209
M2.5 x 5mm Pan Head Screw			92000A103
#4-40 Flat Head Screw			91771A111
	http://www.ortec-online.com/		ORTEC
	http://instruments.perkinelmer.com/		PerkinElmer instruments
Blank NIM Module, single			400A
Winchester connector	http://litton-wed.com		M9PLSH16

9 Schematic

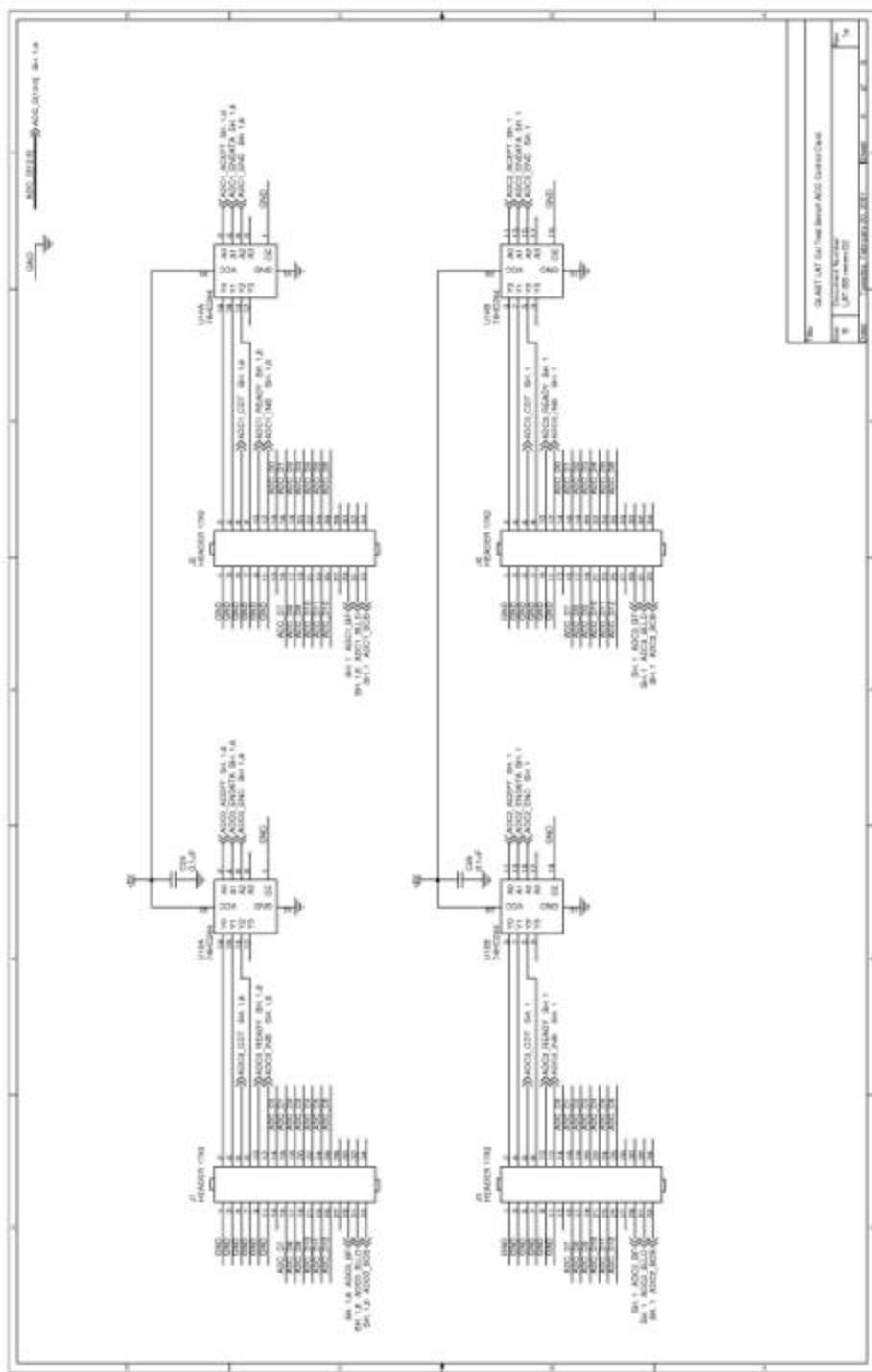




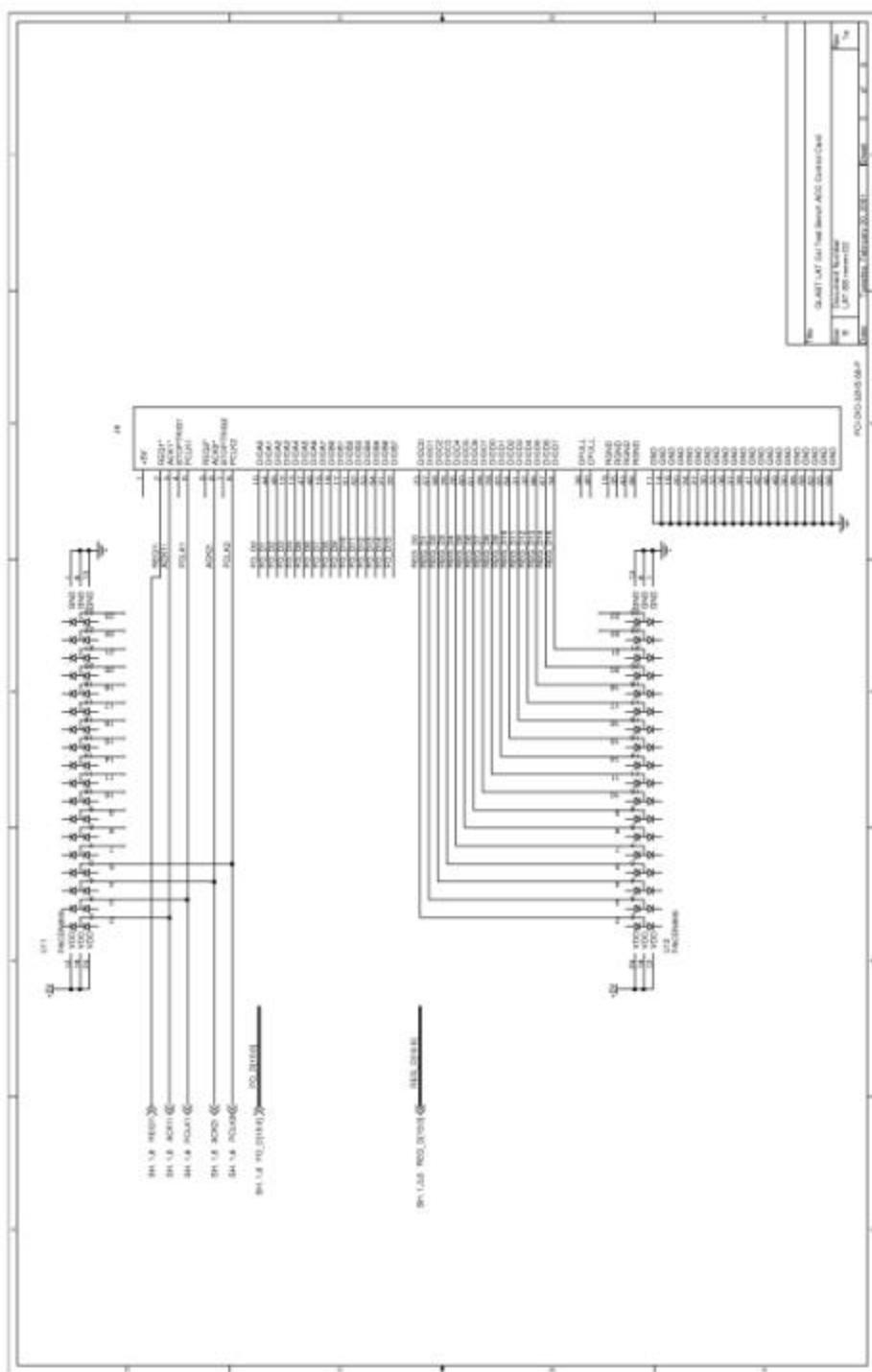
Hard copies of this document are for REFERENCE ONLY and should not be considered the latest revision.



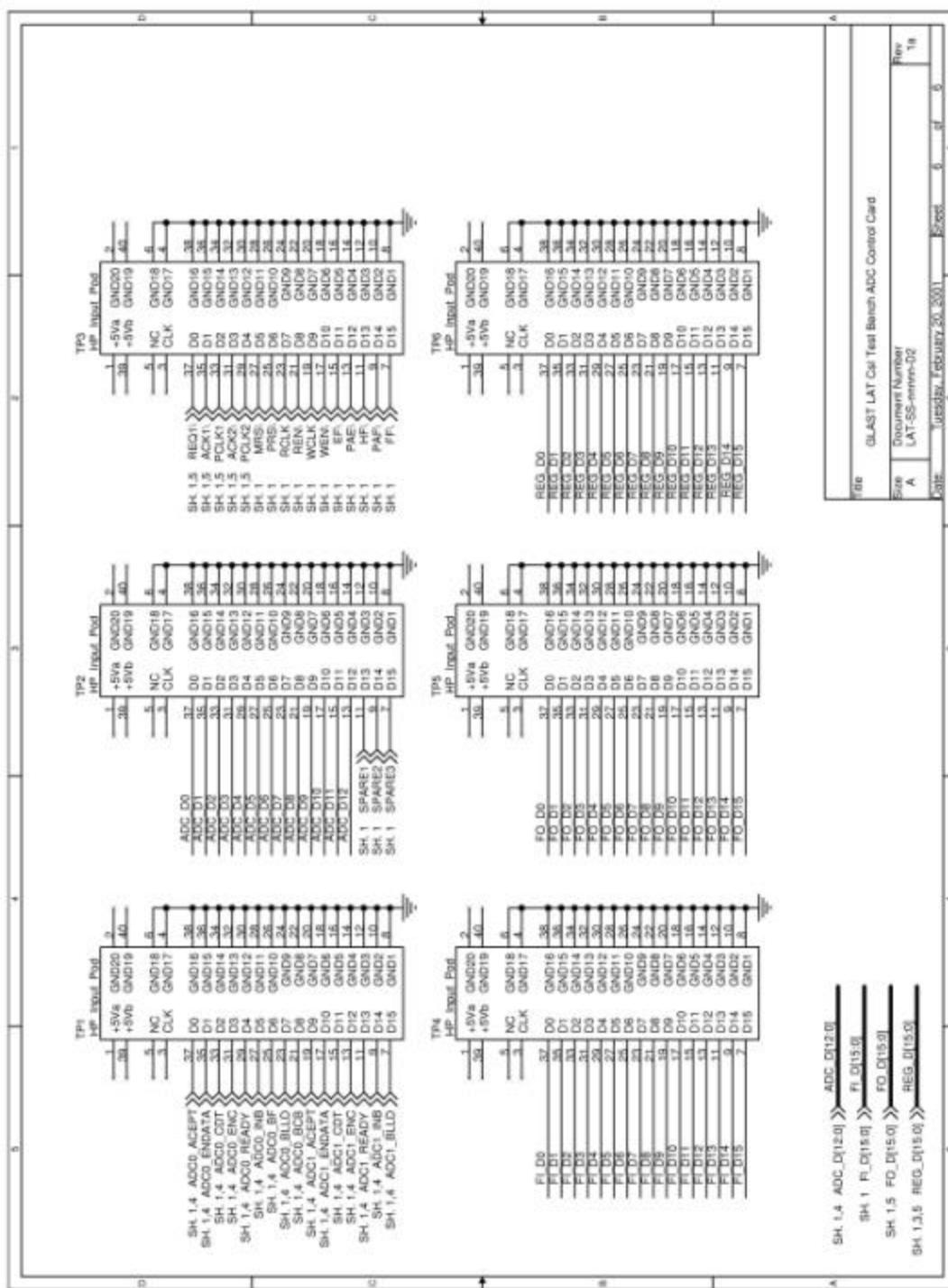
Hard copies of this document are for REFERENCE ONLY and should not be considered the latest revision.



Hard copies of this document are for REFERENCE ONLY and should not be considered the latest revision.



Hard copies of this document are for REFERENCE ONLY and should not be considered the latest revision.



Hard copies of this document are for REFERENCE ONLY and should not be considered the latest revision.

10 Assembly Drawing

