EARTH PEOPLE TECHNOLOGY, Inc

UNO DATA COLLECTOR PROJECT
User Manual

This manual describes the EPT USB-CPLD development system Data Collector Project. It goes step by step providing the instructions to build this project. The instructions include the Arduino code, CPLD code, and the Host PC coe.

Circuit designs, software and documentation are copyright © 2012, Earth People Technology, Inc

Microsoft and Windows are both registered trademarks of Microsoft Corporation. Altera is a trademark of the Altera Corporation. All other trademarks referenced herein are the property of their respective owners and no trademark rights to the same are claimed.

http://www.earthpeopletechnology.com/

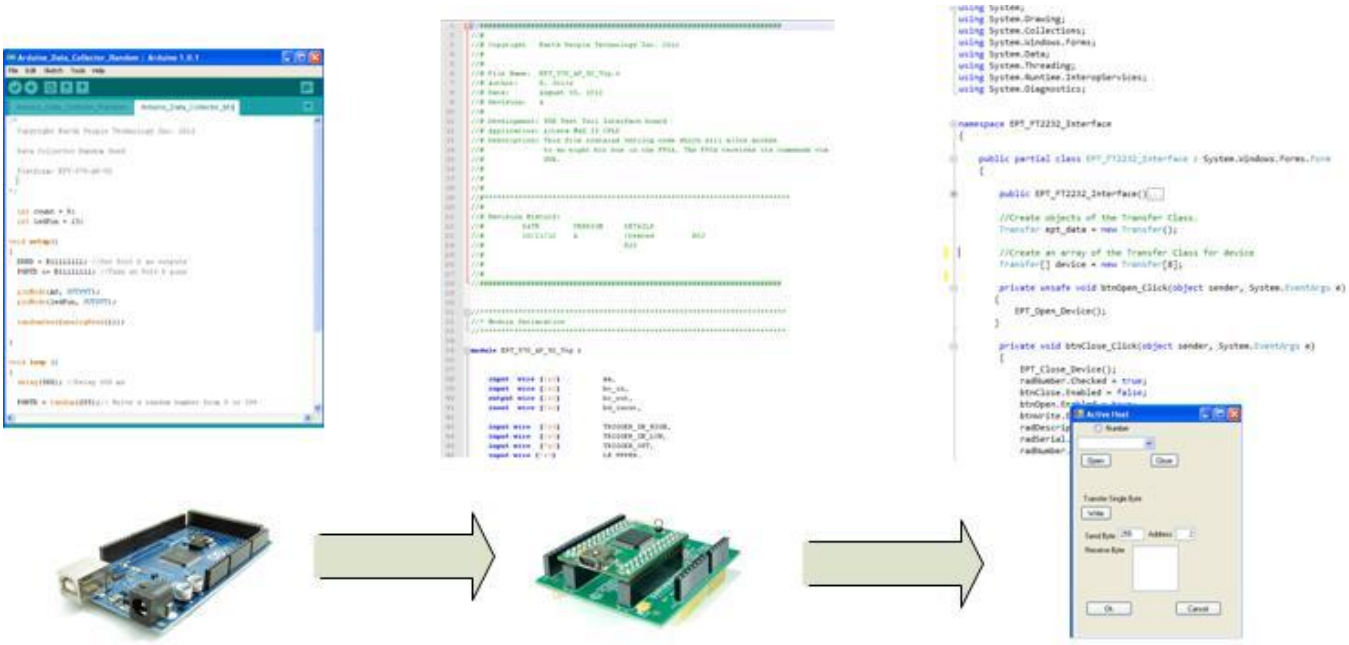
Table of Contents

1 The Development Process

There is no standard for developing embedded electronics. The best method is the one that works for the user. These methods can range from a top down approach where the design is written down first and all code is written, then compile, execute and test. Or a bottom up approach can be pursued where a small piece of the project is assembled and verified (i.e. I2C communication to a sensor). Then the next piece is assembled and verified (i.e. collect sensor data in a storage buffer) and connected to the first. And so on, until the whole design is complete. Or any infinite combination of these two extremes.

1.1 Designing a Simple Data Collection Sampler

The Data Collection Sampler is a very simple introductory project that will guide the user in the creation of an overall design using the Arduino Programming Language, Verilog HDL, and C# Language. These elements will run on the Arduino Platform, EPT-570-AP-U2 CPLD, and a Windows 7 PC respectively.



The first order of business is to layout the design. Start with the Arduino, and create a simple bit output using a random number generator. Next, use the EPT Active Transfer Library to create a byte transfer module to read the byte from the Arduino and send it to the Host PC. Finally, use EPT Active Host to accept the byte transfer from EPT Active

Transfer, and display in a textbox. This is just the hierarchical system level design. In the following sections, we will fill in the above blocks.

1.1.1 The Arduino Microcontroller Board

Using the features and capabilities of the Arduino development system, the user will develop the source code using the "Wiring" programming language and download the resulting binary code from the Processing development environment to the Flash memory of the microcontroller.

1.1.2 Create Data Generator

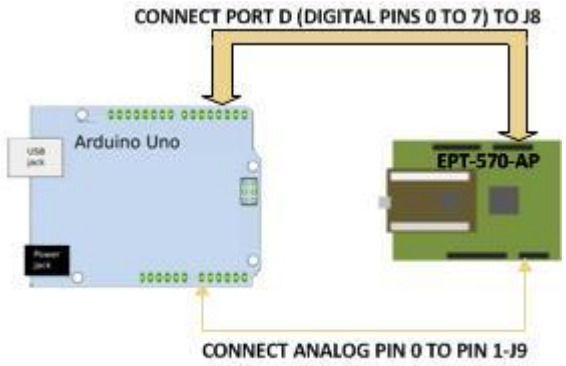
To keep the design simple, no external data source will be used. We will create a data source using the Arduino, then transmit this data to the EPT-570-AP board. To create the data source, we will use the random() function. This function generates pseudo random numbers from a seed value. We will give the randomSeed() function a fairly random input using the value from the analogRead(). This will give different values every time the random() function is called. We will limit the random number output from the function to 8 bits. The random() function will be called once per iteration of the loop() function.

The randomSeed() function must be called during the setup() function. It takes as input parameter the output of the Analog Pin 1. The output of this Pin 1 will have a small amount of random noise on it. Because of this noise, the randomSeed() function will produce a different seed every time the sketch is initialized.

```
void setup()
{
  randomSeed(analogRead(1));
}
```

1.1.3 Select I/O's for Fast Throughput on Arduino

An 8 bit port is used to connect the 8 bit byte from the random function output to the input of the EPT-570-AP. There is also a one bit control line which will be used to inform the CPLD that a byte is ready to be written to the USB.



Each port is controlled by three registers, which are also defined variables in the Arduino language. The DDR register, determines whether the pin is an INPUT or OUTPUT. The PORT register controls whether the pin is HIGH or LOW, and the PIN register reads the state of INPUT pins set to input with pinMode(). The maps of the ATmega328 chips show the ports.

DDR and PORT registers may be both written to, and read. PIN registers correspond to the state of inputs and may only be read.

PORTD maps to Arduino digital pins 0 to 7

DDRD - The Port D Data Direction Register - read/write
PORTD - The Port D Data Register - read/write
PIND - The Port D Input Pins Register - read only

The ports and pins for the Data Collection Sampler project must be initialized in the setup() function. The setup function will only run once, after each powerup or reset of the Arduino board.

```
int ledPin = 13;

void setup()
{
  DDRD = B11111111; //Set Port D as outputs
  PORTD &= B11111111; //Turn on Port D pins

  pinMode(A0, OUTPUT);

}
```

After the setup() function executes, the PORTD is ready to be assigned the results of our random() function. And the A0 pin will be used to latch the value on PORTD pins into the CPLD.

1.1.4 Coding the Arduino Data Sampler

Now that we have the data generator and the ports defined, we can add some delays in the loop() function and make a simulated data collector. Because Start and Stop buttons will be added to the C# Windows Form, the Data Collector code will need to monitor a single pin output from the EPT-570-AP. This output pin (from the EPT-570-AP) becomes an input to the Arduino and is used in conditional switch.

```
void loop ()
{

   //Sample the Start/Stop switch
   //from the EPT-570-AP
   startStopBit = digitalRead(inPin8);

  delay(500); //Delay 500 ms

   if(startStopBit)
   {
```

This code will sample the Start/Stop switch which is an output from the EPT-570-AP on J10 PIN 1. On the Arduino, this is PIN 8 of the Digital pins. Each iteration of the loop() function, the startStopBit variable stores the state of DigitalPin8. Then, a delay of 500 milliseconds is added. The delay() function pauses the program for the amount of time (in milliseconds) specified as parameter. Next, the startStopBit is checked with a conditional switch. If the bit is set, the conditional branch is entered and the random number is sent to the EPT-570-AP. If the bit is not set, the end of the loop() function is reached and it branches to the top of the loop().

We will also add an LED Pin that will blink so that we can have a visual indication that the project is working.

We want to add a delay so that the data from the generated displays on the Windows PC long enough for our eyes to verify that the data is updating correctly. This delay should be one second in total. So, the data will change then stay stable in the textbox for one second before changing again.

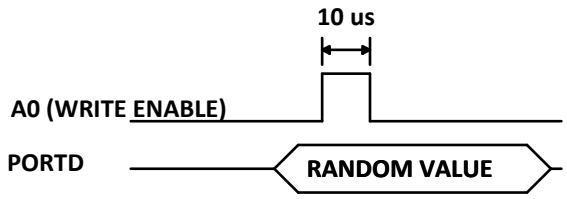
For the LED to blink correctly, it should turn on, delay for half a second then turn off and delay for half a second. If we don't use half second intervals for the LED blink, the LED will appear to not change at all. It will look like it stays on all the time or off all the time.

So, the code looks like this:

```
/*
  Copyright Earth People Technology Inc. 2012

  Data Collector Random Seed

  Platform: EPT-570-AP-U2

*/

  int startStopBit = 0;
  int count = 0;
  int ledPin = 13;
  int inPin8 = 8;

void setup()
{
  DDRD = B11111111; //Set Port D as outputs
  PORTD &= B11111111; //Turn on Port D pins

  pinMode(A0, OUTPUT);
  pinMode(ledPin, OUTPUT);
  pinMode(inPin8, OUTPUT);

  randomSeed(analogRead(1));

}

void loop ()
{

   //Sample the Start/Stop switch
   //from the EPT-570-AP
   startStopBit = digitalRead(inPin8);

  delay(500); //Delay 500 ms

   if(startStopBit)
   {
     // Write a random number from 0 to 299
     //to the input of the EPT-570-AP
      PORTD = random(255);
      //Set the Write Enable Pin High
       digitalWrite(A0, HIGH);
```

```
        //Set the LED Pin High
        digitalWrite(ledPin, HIGH);

        delay(500); //Delay 500 ms

        //Set the LED Pin Low
        digitalWrite(ledPin, LOW);
        //Set teh Write Enable Pin Low
        digitalWrite(A0, LOW);
    }

}
```

Notice that PORTD equals the return of random(255). The parameter passed to the random() function is the maximum decimal value of the return value. In our case we want the maximum value to be an 8 bit value, B11111111 = 0xff = 255(decimal). Also, note that the A0 write enable signal for the CPLD has back to back instructions turning it on then off immediately. Because the ATMega328 chip takes approximately 160 clock cycles to execute the digitalWrite() function and affect the Pin at A0, this produces a write enable pulse of 10 microseconds.



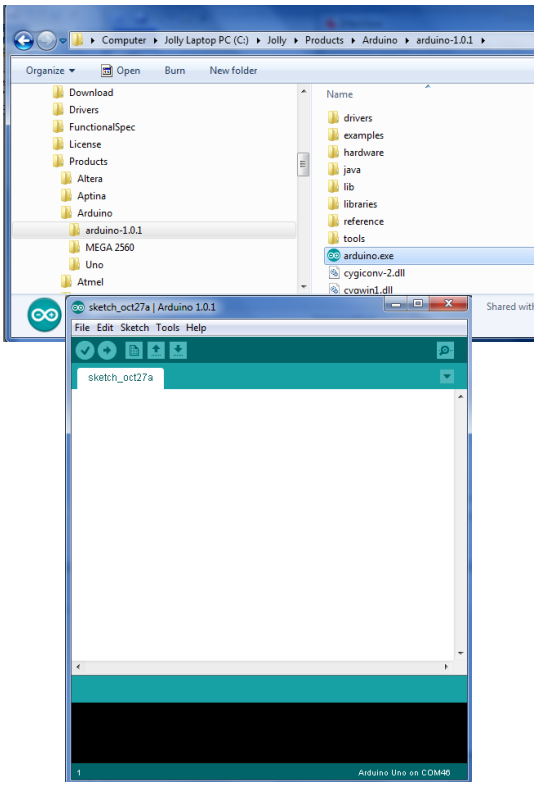
The RANDOM VALUE will be stable before the A0(WRITE ENABLE) asserts thus guaranteeing a successful transfer of data from Arduino to CPLD.

1.1.5 Building Arduino Project

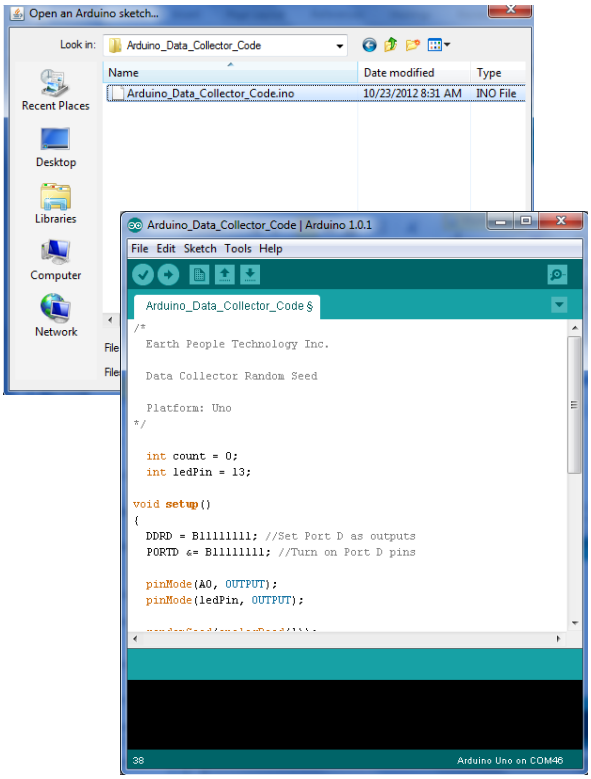
Building the Arduino project is the process of converting (compiling) the code you just wrote into machine level code that the processor can understand. The Arduino IDE is the software tool that does the compiling. The machine level code is a set of basic instructions that the processor uses to perform the functions the user code.

To compile your code,

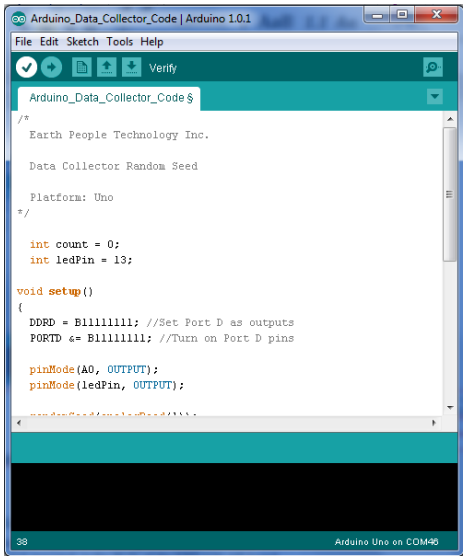
- Open up the Arduino IDE



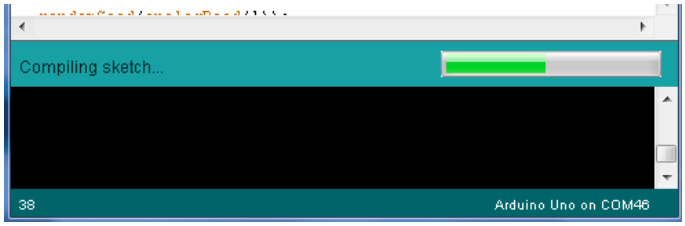
- Load your code



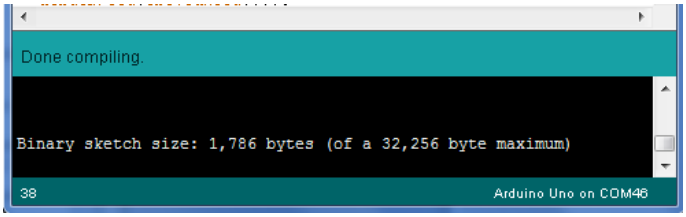
- Click the Verify button



- The sketch will compile



- If there are no errors, the compiling will complete successfully



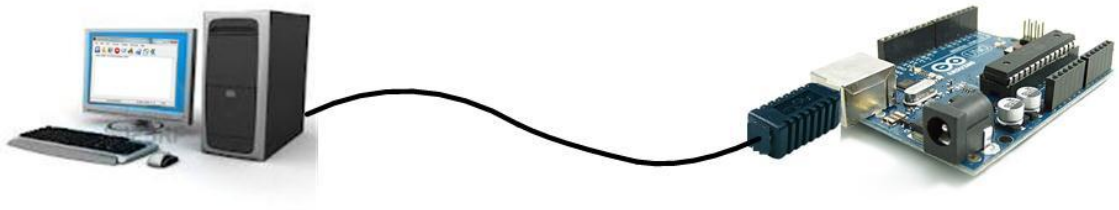
Now we are done with compiling and ready to program the Arduino

1.1.6 Programming the Arduino

Programming the Arduino is the process of downloading the user's compiled code into the Flash memory of the Atmel ATMega328 chip. Once the code is downloaded, the Arduino IDE resets the chip and the processor starts executing out of Flash memory.

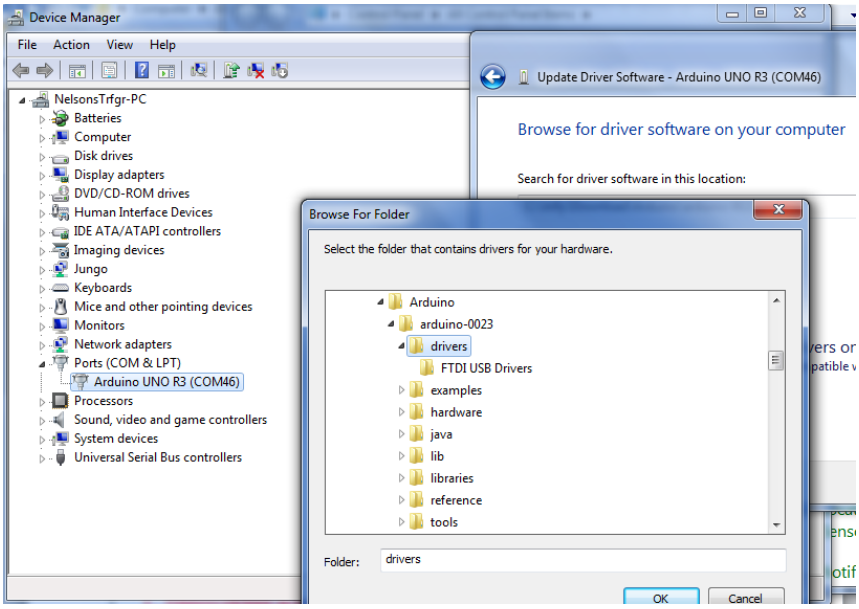
To program the Arduino

- Connect the USB cable from PC to Arduino

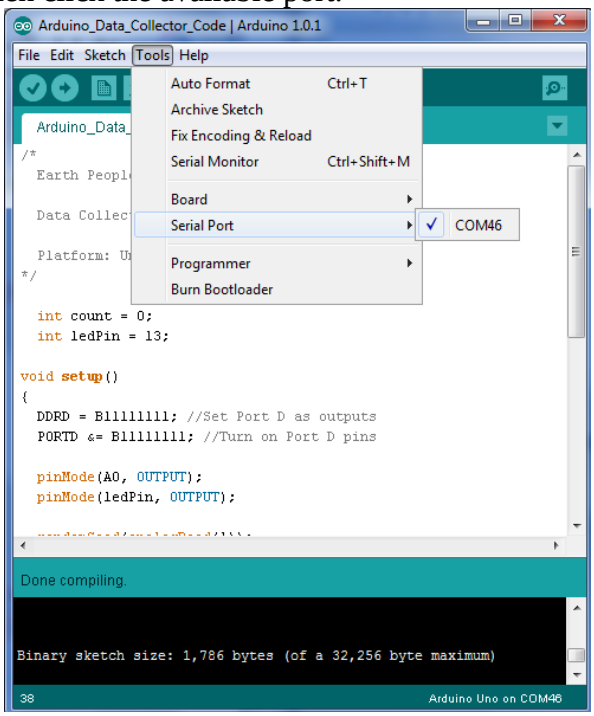


- Load the Arduino USB driver according to the manual
- Plug in your board and wait for Windows to begin it's driver installation process. After a few moments, the process will fail, despite its best efforts
- Click on the Start Menu, and open up the Control Panel.
- While in the Control Panel, navigate to System and Security. Next, click on System. Once the System window is up, open the Device Manager.
- Look under Ports (COM & LPT). You should see an open port named "Arduino UNO (COMxx)"
- Right click on the "Arduino UNO (COmxx)" port and choose the "Update Driver Software" option.
- Next, choose the "Browse my computer for Driver software" option.

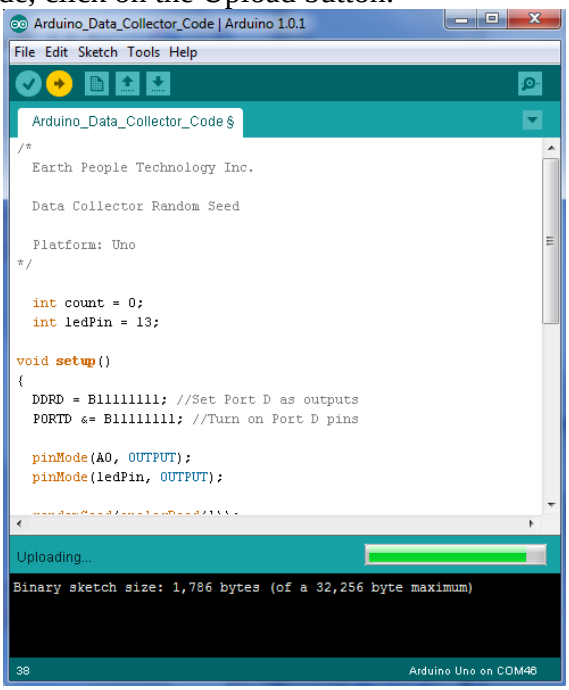
- Finally, navigate to and select the Uno's driver file, named **"ArduinoUNO.inf"**, located in the "Drivers" folder of the Arduino Software download (not the "FTDI USB Drivers" sub-directory).
- Windows will finish up the driver installation from there.



- Once the driver is loaded, we can set the COM Port. Click on Tools and select Serial Port, then click the available port.



- To load the code, click on the Upload button.



When the code has completed loading, the Arduino IDE will automatically command the processor to start executing the code. The L LED will blink at one second intervals.

1.1.7 CPLD Active Transfer Coding and Initiation

The EPT-570-AP will accept the data collected by the Arduino and transfer it to the PC. It is designed to plug directly into the Arduino Uno and there is no need for external wires to be added. The Active Transfer Library can be used to send the data to the PC. This library has been designed to make it easy to transfer data to and from the PC via the USB. The user must create some interface code between the incoming data and the Active Transfer Library. We will now go through exercise of creating the CPLD code for the Data Collector Sampler.

1.1.8 CPLD: Define the User Design.

In this step we will define the user's code and include modules from the EPT Active Transfer Library. The Active Transfer Library contains a set of files with a ".vqm" name extension which select particular operations to perform (e.g., byte transfer, block transfer, trigger).. The active_transfer_library.vqm file must be included in the top level file of the project. The function modules will connect to the active_transfer_library and provide a path to connect user code to the library. All of these files are available on the Earth People Technology website.

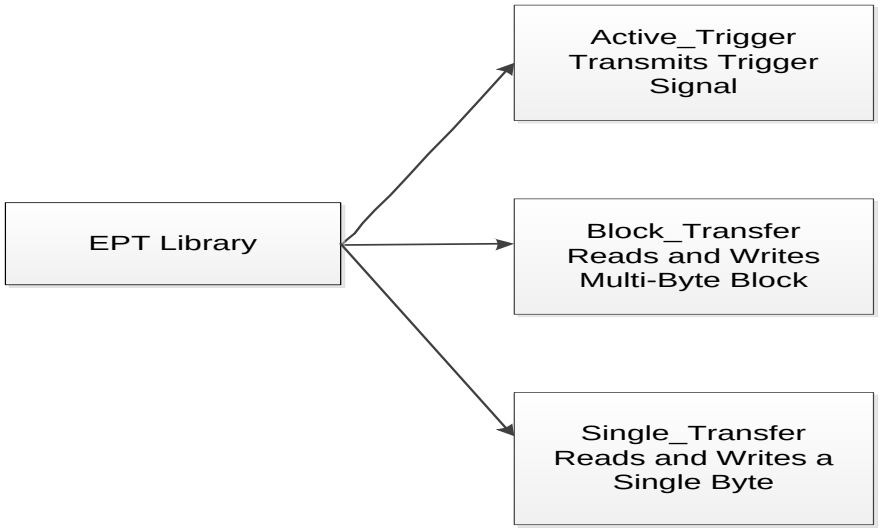
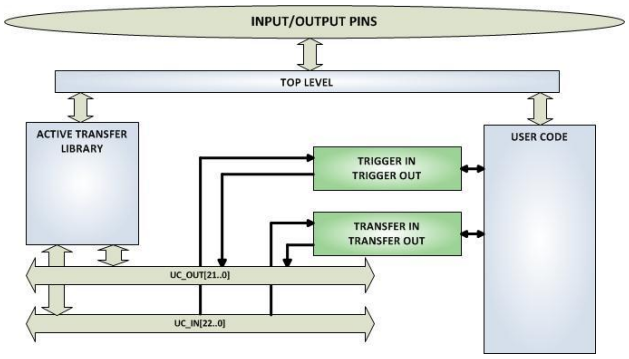


Figure 5. EPT Library EndTerm Modules

We will build our CPLD project using Quartus II software from Altera. The primary file defining the user's CPLD project is named "EPT_570_AP_U2_Top.v". It defines the user code and connects the active_transfer_library and active_transfer logic functions. In order to route the pins of the Arduino to the CPLD, the Pin Planner tool is used. This tool allows the user to match internal net names to the pins of the CPLD.

Our project needs to accept an 8 bit value on the J8 connector and a write enable on Pin 1 of J9. For this, we can use the active_transfer.vqm module as the interface to the active_transfer_library. It accepts a single byte and latches it with a single enable net. Because the active_transfer_library runs at 66 MHz we will need to write some code ensure that the slower A0 (write enable) signal from the Arduino can latch the data into the active_transfer module.



CPLD: Coding up the DesignThe first thing to do is to create a top level file for the project. The top level file will include the input and outputs for the CPLD. These are declared according to the Verilog syntax rules. We won't go through all the rules of Verilog here, but feel free to explore the language more thoroughly at

www.asic-world.com/verilog/

We need to add the inputs and outputs for active_transfer_library, user code, leds, and switches. Each port is described as input, output or inout. It is followed by the net type wire or reg. If it is a vector, the array description must be added.

```
module EPT_570_AP_U2_Top (


    input  wire [1:0]          aa,
    input  wire [1:0]          bc_in,
    output wire [2:0]          bc_out,
    inout  wire [7:0]          bd_inout,

    input wire  [1:0]          TRIGGER_IN_HIGH,//XIOH -- J10
    input wire  [5:0]          TRIGGER_IN_LOW, //AD   -- J9
    output reg  [7:0]          LB_LOWER,       //XIOH -- J10
    input wire  [7:0]          LB_UPPER,       //XIOL -- J8


    //Transceiver Control Signals
    output reg                 TR_DIR_1,
    output reg                 TR_OE_1,

    output wire                TR_DIR_2,
    output wire                TR_OE_2,

    output wire                TR_DIR_3,
    output wire                TR_OE_3,


    input wire                SW_USER_1,
    input wire                SW_USER_2,

    output reg [2:0]          LED,
    output wire               LED3
    );
```

Next, the parameter's are defined. These are used as constants in the user code.

```
//------------------------------------------------
// Parameters
//------------------------------------------------

//Header Bytes for the Transfer Loopback detection
parameter                   TRANSFER_CONTROL_BYTE1 = 8'h5A;
parameter                   TRANSFER_CONTROL_BYTE2 = 8'hC3;
parameter                   TRANSFER_CONTROL_BYTE3 = 8'h7E;

//State Machine Transfer Loopback detection
parameter                   TRANSFER_CONTROL_IDLE = 0,
                            TRANSFER_CONTROL_HDR1 = 1,
                            TRANSFER_CONTROL_HDR2 = 2,
                            TRANSFER_DECODE_BYTE   = 3,
                            TRANSFER_CONTROL_SET  = 4;

parameter                   GLOBAL_RESET_COUNT = 12'h09c8;
```

Next is the Internal Signal and Register Declarations.

```
//*****************************************************************************
//* Internal Signals and Registers Declarations
//*****************************************************************************
    wire                        CLK_66;
    wire                        RST;

    wire [23:0]                 UC_IN;
    wire [21:0]                 UC_OUT;

    //Trigger Signals
    reg [7:0]                   trigger_out;
    wire [7:0]                  trigger_in_byte;
    reg [7:0]                   trigger_in_store;

    //LED registers
    reg                         led_reset;

    //Switch registers
    reg                         switch_reset;

    //Transfer registers
    wire                        transfer_out;
    reg                         transfer_out_reg;
    wire                        transfer_in_received;
    wire [7:0]                  transfer_in_byte;
    wire  [7:0]                 transfer_out_byte;
    reg  [3:0]                  transfer_to_host_counter;
    reg  [3:0]                  transfer_to_host_state;

    //Transfer Control registers
    reg                         transfer_in_loop_back;
    reg                         transfer_in_received_reg;
    reg  [3:0]                  transfer_control_state;
    reg  [7:0]                  transfer_control_byte;

    //Transfer Write from Arduino
    reg                         transfer_write_reg;
    reg                         transfer_write;
    reg  [7:0]                  transfer_write_byte;

    //Reset signals
    wire                        reset;
    reg [11:0]                  reset_counter;
    reg                         reset_signal_reg;

    //Input/Output Signals
    reg                         start_stop_cntrl;
```

Next, add the assignments. These assignments will set the direction of the bus transceivers that interface to the Arduino I/O's. The transceivers also include an output enable bit.

```
//*****************************************************************************
//*     Signal Assignments
//*****************************************************************************
   assign            TR_DIR_2  = 1'b1; //1 = A to B; 0 = B to A
   assign            TR_OE_2   = 1'b0;

   assign            TR_DIR_3  = 1'b1; //1 = A to B; 0 = B to A
   assign            TR_OE_3   = 1'b0;

   //Clock and Reset
   assign            CLK_66 = aa[1];
   assign            RST = reset;
   assign            reset = reset_signal_reg;

   //Transfer registers
   assign            transfer_out = transfer_out_reg | transfer_write;
   assign            transfer_out_byte = transfer_write_byte;

   //LED3 is used to signify to the  user that the Start
   //switch is enabled
   assign             LED3 =  ~start_stop_cntrl;
```

The reset signal is generated by a counter that starts counting upon power up. When the counter reaches GLOBAL_RESET_COUNT.

```
//*****************************************************
//*     Reset Signal
//*****************************************************
    always @(posedge CLK_IN or negedge aa[0])
    begin
      if(!aa[0])
      begin
            reset_signal_reg <= 1'b0;
            reset_counter <= 0;
      end
      else
      begin
        if( reset_counter < GLOBAL_RESET_COUNT )
        begin
            reset_signal_reg <= 1'b0;
            reset_counter <= reset_counter + 1'b1;
        end
        else
        begin
            reset_signal_reg <= 1'b1;
        end
      end
```

The four LED's are set by the bottom four bits of the active_trigger output register. These trigger outputs can be set by using a function in the Active_Host DLL on the PC. The Data Collector project will use LED3 to indicate the state of the Start/Stop signal.

```
  //LED3 is used to signify to the  user that the Start
  //switch is enabled
  assign             LED3 =  ~start_stop_cntrl;
```

```
//-------------------------------------------------
// LED Set
//-------------------------------------------------

always @(trigger_in_byte or led_reset or LED or RST)
begin
    if(!RST)
        LED[2:0] = 3'hz;
   else if(led_reset)
        LED[2:0] = 3'hz;
   else if(trigger_in_byte[3:0])
    begin
        case(trigger_in_byte[3:0])
            3'h1:
                LED[0] = 1'b0;
            3'h2:
                LED[1] = 1'b0;
            3'h4:
                LED[2] = 1'b0;
            default:
                LED[2:0] = LED[2:0];
        endcase
    end

end
```

The two user switches are connected to the input trigger register. Pressing a switch will send a trigger to the PC to be decoded by the Active_Host DLL.

```
//-----------------------------------------------
// User Switch Trigger
//-----------------------------------------------
always @(posedge CLK_IN or negedge RST)
begin
   if(!RST)
   begin
       trigger_out <= 8'h00;
   end
  else
  begin
     if(!SW_USER_1 )
         trigger_out <= 8'h01;
     else if(!SW_USER_2 )
         trigger_out <= 8'h02;
      else if(switch_reset)
         trigger_out <= 8'h04;
      else
         trigger_out <= 8'h00;
  end
end
```

Next, we will add the transfer detection signal from the Arduino. This block will require three registers.

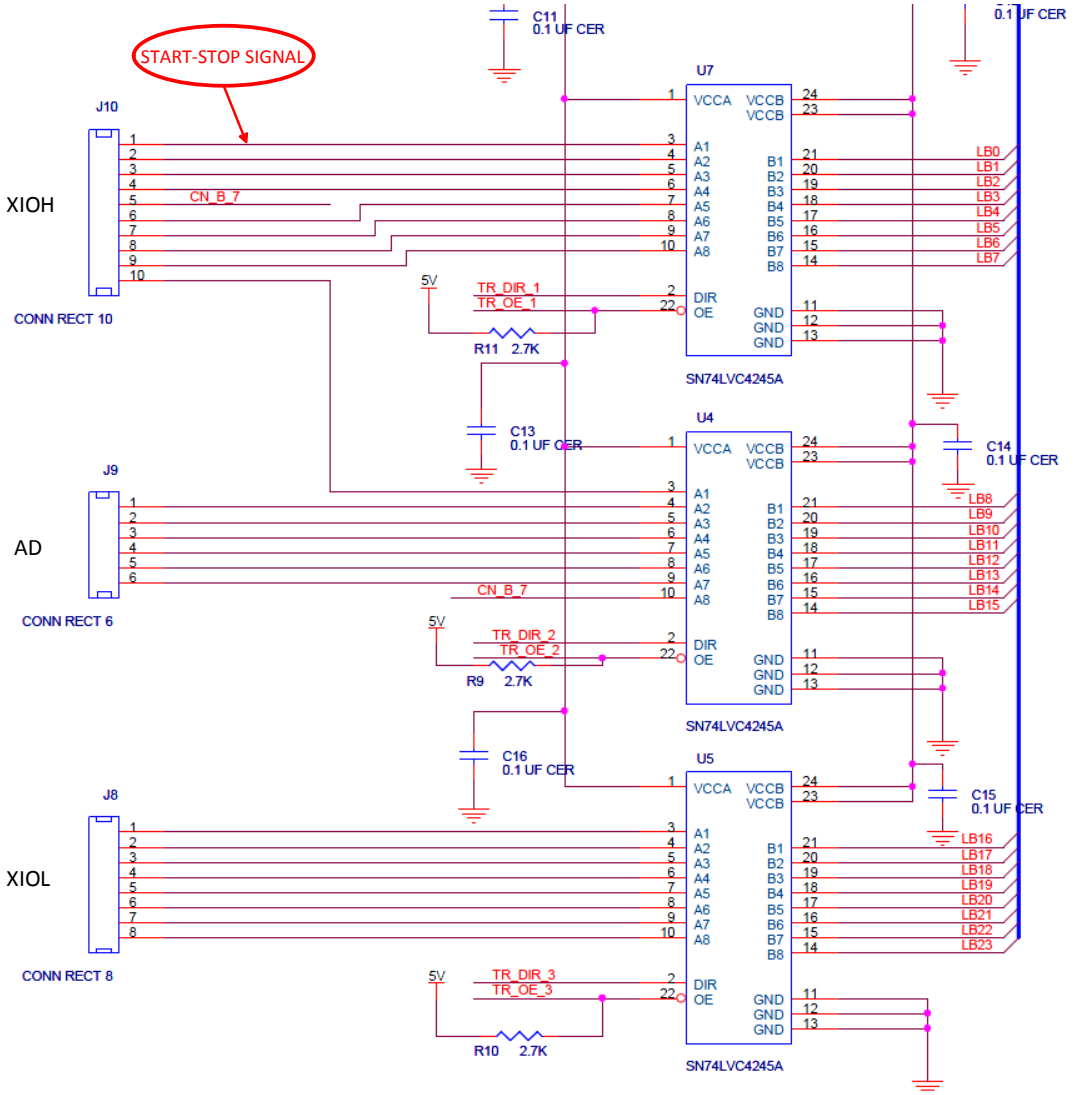
- transfer_write_reg –This is a latch register to hold the state of the A0(Write Enable)
- transfer_write –This register is used to start the active_transfer single byte write to the PC.
- transfer_write_byte –This is an 8 bit register to hold the value of the Data Collection output.

This block will compare the input signal on TRIGGER_IN_LOW[1] to a high. The TRIGGER_IN_LOW[1] pin is routed to Pin 1 of J9 which is routed to the A0(Write Enable) of the Arduino Data Collector. When this bit goes high, the priority encoder goes into statement 1 and sets transfer_write_reg and transfer_write high and latches the value on the LB_UPPER[7:0] pins to the transfer_write_byte register. By setting transfer_write_reg high, the priority encoder goes into statement 2 which will set transfer_write register to low and stay in statement 2 of the priority encoder. The back to back high and low on the transfer_write register will cause the active_transfer module to latch the value of transfer_write_byte into the active_transfer_library module and sets up the byte transfer to the PC. When the TRIGGER_IN_LOW[1] -A0(Write Enable) pin goes low, the encoder will reset transfer_write_reg and transfer_write to

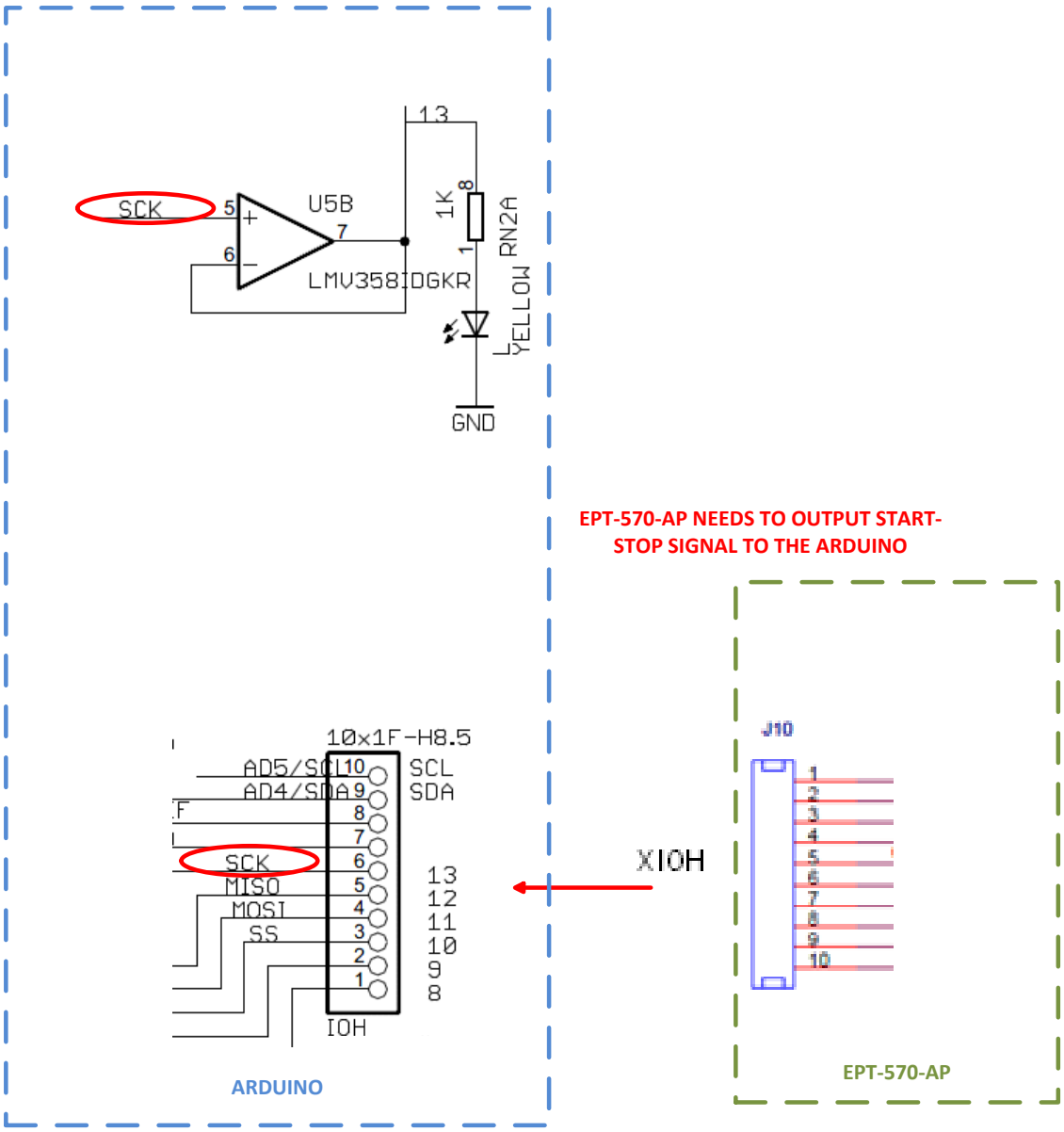
low. The encoder goes back to waiting for the TRIGGER_IN_LOW[1] -A0(Write Enable) to assert high.

```
//-----------------------------------------------
// Detect Transfer From Arduino
//-----------------------------------------------
 always @(posedge CLK_IN or negedge RST)
 begin
    if (!RST)
    begin
        transfer_write_reg <= 1'b0;
        transfer_write <= 1'b0;
        transfer_write_byte <= 0;
    end
    else
    begin
        if(TRIGGER_IN_LOW[1] & !transfer_write_reg)
        begin
            transfer_write_reg <= 1'b1;
            transfer_write <= 1'b1;
            transfer_write_byte <= LB_UPPER;
        end
        else if(TRIGGER_IN_LOW[1] & transfer_write_reg)
        begin
            transfer_write_reg <= 1'b1;
            transfer_write <= 1'b0;
        end
        else if(!TRIGGER_IN_LOW[1] & transfer_write_reg)
        begin
            transfer_write_reg <= 1'b0;
            transfer_write <= 1'b0;
            transfer_write_byte <= 0;
        end
    end
 end
```

This block of code takes care of reading the random word from the Arduino using the A0(Write Enable) Pin. However, because the Arduino is expecting a Start/Stop bit on Digital Pin8, the CPLD code has to provide this bit. This presents a problem, the EPT-570-AP has 3 eight bit bi-directional ports. Which means each port is has a direction which is either input or ouput at a given time. However, the ports can be switched between input and output at any time. Two of the three ports must be used as inputs into the CPLD for the random word and the A0(Write Enable) Pin. So, the third port can be used as the output port.



This, however, causes another problem! The Arduino XIOH connector needs to output the Amber LED state. So, if one pin on the connector needs to be an output, the EPT-570-AP port on J10 (XIOH) cannot be an output! This would interfere with the turning on and turning off of the LED.



So, we can fix this problem by noting that the 8 bit bi-directional ports on the EPT-570-AP have Output Enables that allow the CPLD to "float" the signals of the port at any time. By floating the port, we can multiplex the signals of the port. When we need to drive the signals from the EPT-570-AP port to the Arduino, we turn on the Output Enables of the port. And when we need to let the Arduino drive its signals, we turn off the Output Enables of the port.

```
//-----------------------------------------------
// Detect Transfer From Arduino
//-----------------------------------------------
 always @(posedge CLK_66 or negedge RST)
 begin
    if (!RST)
    begin
        transfer_write_reg <= 1'b0;
        transfer_write <= 1'b0;
        transfer_write_byte <= 0;
        TR_DIR_1  <= 1'b0; //1 = A to B; 0 = B to A
        TR_OE_1  <= 1'b0;
        LB_LOWER[0] <= start_stop_cntrl;
        LB_LOWER[7:1] <= 7'b0100000;
    end
```

In the reset section of the synchronous block, we turn the Direction bit to “B to A”

TR_DIR_1 <= 1‘b0;

and the Output Enable on.

TR_OE_1 <= 1’b0; (Output Enables are asserted with a zero)

The start_stop_cntrl signal is set by using the TRANSFER_CONTROL state machine in the following section. So, if the start_stop_cntrl signal is set, the Output Enable is turned on and the signal will appear on DigitalPin8 on the Arduino XIOH connector. As the Data Collector code cycles through its loop() function, it will cause the if statement to branch into its conditional statement. The Data Collector code will assert the A0(Write Enable) Pin in its conditional statement. The A0(Write Enable) Pin will cause the CPLD code to enter into its first conditional statement. This first statement turns off the Output Enables of the Port J10. With the Port turned off, the Arduino can set the LED on when it executes its code. When the A0(Write Enable) Pin is de-asserted, the Output Enable of Port J10 is turned back on and the whole process can start over.

Next, we add a TRANSFER_CONTROL state machine to read the Control Register from the Host PC using the active_transfer EndTerm. This state machine will decode the 8 bit control register only after a sequence of three 8 bit bytes in the order of 0x5a, 0xc3, 0x7e. The operation of the state machine is as follows.

- The TRANSFER_CONTROL state machine will stay in the idle state of the parallel encoder until a byte from the active_transfer transfer_to_device register receives a 0x5a.
- This will cause the transfer_control_state to be changed to TRANSFER_CONTROL_HDR1.
- The state machine will stay in the TRANSFER_CONTROL_HDR1 state until the next byte is read from the active_transfer.

- If the byte from transfer_to_device is a 0xc3, the transfer_control_state will be changed to TRANSFER_CONTROL_HDR2.
- If the byte from transfer_to_device is not a 0xc3, the transfer_control_state will go back to idle.
- In the TRANSFER_CONTROL_HDR2 state , the state machine will stay in this state until the next byte from the active_transfer is received.
- If the byte from transfer_to_device is a 0x7e, the transfer_control_state will be changed to TRANSFER_DECODE_BYTE.
- If the byte from transfer_to_device is not a 0x7e, the transfer_control_state will go back to idle.
- In the TRANSFER_DECODE_BYTE state , the state machine will stay in this state until the next byte from the active_transfer.
- The next byte transferred from active_transfer will be decoded as the Control Register.

The bits of the Control Register are defined below.

Register	Bits	Description	Assertion
Control	0	Start Stop Cntrl	High
	1	Not Used	
	2	LED Reset	High
	3	Switch Reset	High
	4	Transfer In Loop Back	High
	5	Not Used	
	6	Not Used	
	7	Not Used	
	7	Not Used	

```
//-----------------------------------------------
// State Machine: Control Register from Transfer In
//-----------------------------------------------
 always @(posedge CLK_IN or negedge RST)
 begin
      if (!RST)
      begin
          transfer_in_received_reg <= 1'b0;
          transfer_control_state <= TRANSFER_LOOPBACK_IDLE;
          transfer_in_loop_back <= 1'b0;
          led_reset <= 1'b0;
          switch_reset <= 1'b0;
      end
      else
      begin
          if(transfer_in_received & !transfer_in_received_reg)
          begin
              transfer_in_received_reg <= 1'b1;
              case(transfer_control_state)
              TRANSFER_CONTROL_IDLE:
                  if((transfer_in_byte == TRANSFER_CONTROL_BYTE1))
                      transfer_control_state <= TRANSFER_CONTROL_HDR1;
                  else if((transfer_in_byte != TRANSFER_CONTROL_BYTE1))
                      transfer_control_state <= TRANSFER_CONTROL_IDLE;
                  else
                      transfer_control_state <= TRANSFER_CONTROL_IDLE;
              TRANSFER_CONTROL_HDR1:
                  if((transfer_in_byte == TRANSFER_CONTROL_BYTE2))
                      transfer_control_state <= TRANSFER_CONTROL_HDR2;
                  else if((transfer_in_byte != TRANSFER_CONTROL_BYTE2))
```

```
                    transfer_control_state <= TRANSFER_CONTROL_IDLE;
                else
                    transfer_control_state <= TRANSFER_CONTROL_HDR1;
            TRANSFER_CONTROL_HDR2:
                if((transfer_in_byte == TRANSFER_CONTROL_BYTE3))
                    transfer_control_state <= TRANSFER_DECODE_BYTE;
                else if((transfer_in_byte != TRANSFER_CONTROL_BYTE3))
                    transfer_control_state <= TRANSFER_CONTROL_IDLE;
                else
                    transfer_control_state <= TRANSFER_CONTROL_HDR2;
            TRANSFER_DECODE_BYTE:
            begin
                transfer_in_loop_back <= transfer_in_byte[0];
                led_reset <= transfer_in_byte[2];
                switch_reset <= transfer_in_byte[3];
                transfer_loopback_state <= TRANSFER_LOOPBACK_SET;
            end
            TRANSFER_CONTROL_SET:
            begin
                transfer_control_state <= TRANSFER_CONTROL_IDLE;
            end
            endcase
        end
        else if(!transfer_in_received & transfer_in_received_reg)
            transfer_in_received_reg <= 1'b0;
    end
end
```

Next, up is the instantiation for the active_transfer_library. The ports include the input and output pins and the two buses that connect the active modules. These buses are the input UC_IN[23:0] and output UC_OUT[21:0].

```
//-----------------------------------------------
// Instantiate the EPT Active Transfer Library
//-----------------------------------------------

active_transfer_library    ACTIVE_TRANSFER_LIBRARY_INST
(
.aa                        (aa),
.bc_in                     (bc_in),
.bc_out                    (bc_out),
.bd_inout                  (bd_inout),

.UC_IN                     (UC_IN),
.UC_OUT                    (UC_OUT)

);
```

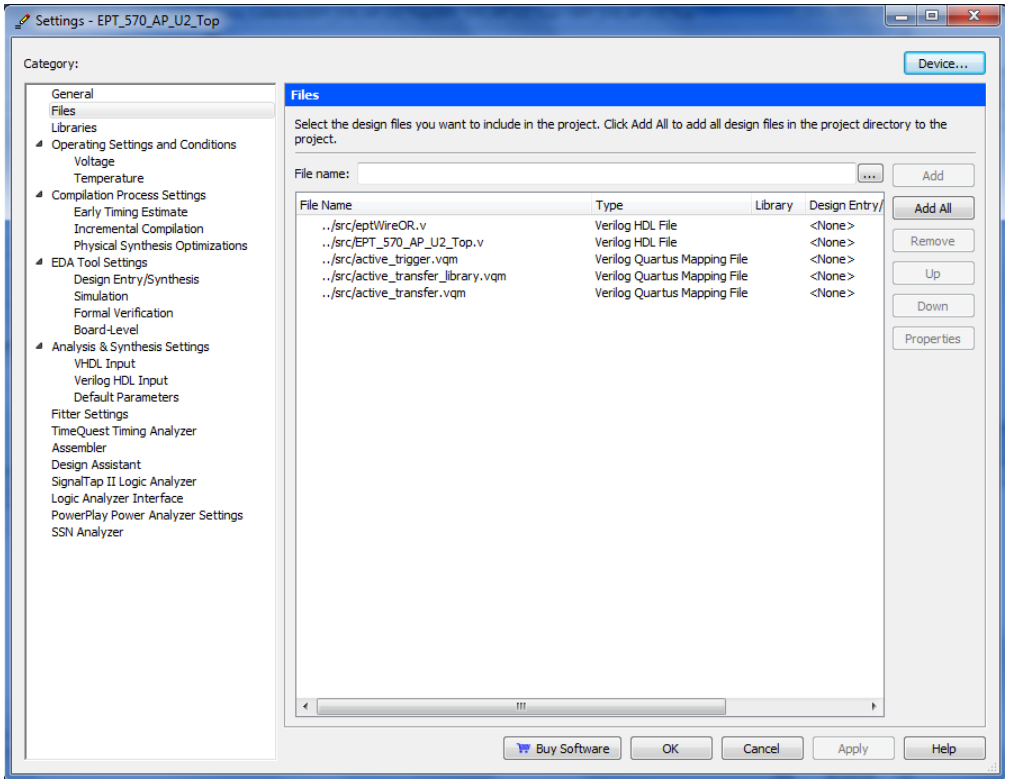
Finally, we instantiate the Active EndTerms. For the Data Collection project, we only need active_transfer and active_trigger EndTerms. The uc_out port for both modules must be shared. Since they both drive this bus, a bus wide wired-or circuit is used so that they don't drive each other. The active_transfer EndTerm has a port for the address (uc_addr). This allows the PC to address up to 8 different modules. Just add a three bit address to this port and the PC must add this same address to communicate with this module.

```
   //------------------------------------------------
   // Instantiate the EPT Active Modules
   //------------------------------------------------
wire [22*2-1:0]  uc_out_m;
eptWireOR # (.N(2)) wireOR (UC_OUT, uc_out_m);
    active_trigger              ACTIVE_TRIGGER_INST
    (
     .uc_clk                    (CLK_IN),
     .uc_reset                  (RST),
     .uc_in                     (UC_IN),
     //.uc_out                    (UC_OUT),
     .uc_out                    (uc_out_m[ 0*22 +: 22 ]),

     .trigger_to_host           (trigger_out),
     .trigger_to_device         (trigger_in_byte)

    );

    active_transfer             ACTIVE_TRANSFER_INST
    (
     .uc_clk                    (CLK_IN),
     .uc_reset                  (RST),
     .uc_in                     (UC_IN),
     //.uc_out                    (UC_OUT),
     .uc_out                    (uc_out_m[ 1*22 +: 22 ]),

     .start_transfer            (transfer_out),
     .transfer_received         (transfer_in_received),

     .uc_addr                   (3'h2),

     .transfer_to_host          (transfer_out_byte),
     .transfer_to_device        (transfer_in_byte)
    );
```

Next, we are ready to compile and synthesize.

1.1.9 CPLD: Compile/Synthesize the Project

The Quartus II application will compile/ synthesize the user code, active_transfer_library, and the active EndTerms. The result of this step is a file containing the CPLD code with "*.pof" name First, we need to create a project in the Quartus II environment. Follow the directions in the section: "1.7 Compiling, Synthesizing, and Programming CPLD". Follow the steps up to Add Files. At the Add Files box, click on the Browse button and navigate to the project Data Collector install folder in the dialog box. Add the files:



Continue following the instructions by adding a device and finishing the project instantiation. Then, bring up the Pin Planner.

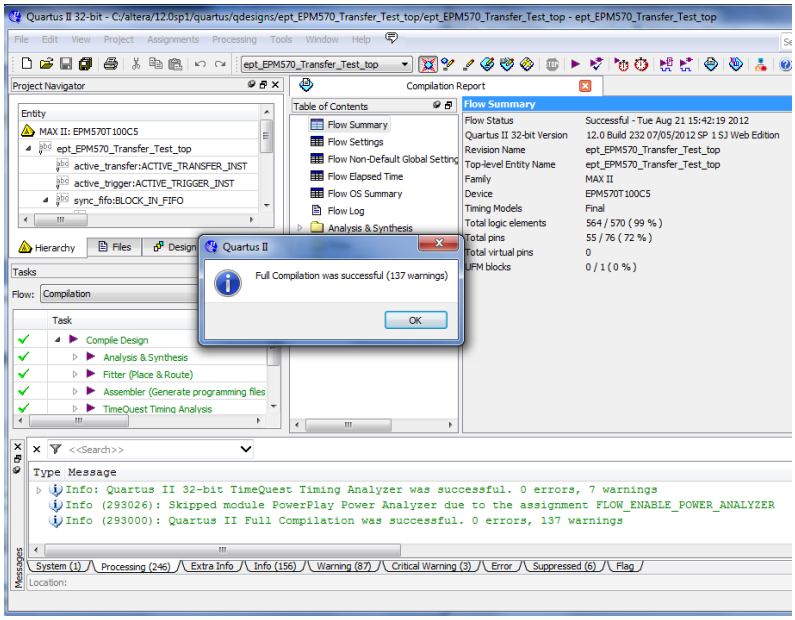
- Under Assignments, Select Import Assignments.

- At the Import Assignment dialog box, browse to the folder with the Quartus Specification File on the EPT USB-CPLD Development System CD. Select the "EPT_570_AP_U2_Top.qsf" file.

- Click Ok. Under Assignments, Select Pin Planner. Verify the pins have been imported correctly.

Exit the Pin Planner, and select the Start Compilation button.

Click Ok then re-run the Compile process. After successful completion, the screen should look like the following:

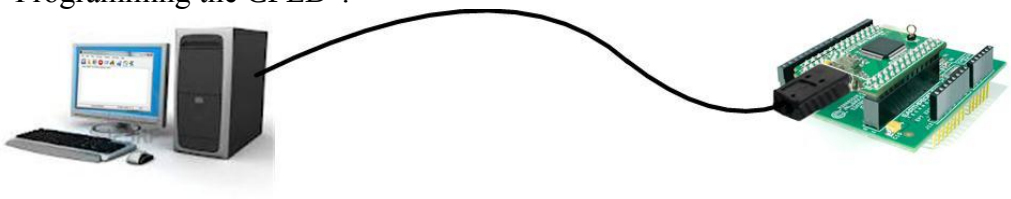


If the synthesis fails, you will see the failure message in the message window. Note that in addition to fatal errors, the compile process can produce “warnings” which do not necessarily prevent execution of the code but which should be corrected eventually.

At this point the project has been successfully compiled, synthesized and a programming file has been produced. See the next section on how to program the CPLD.

1.1.10 CPLD: Program the CPLD

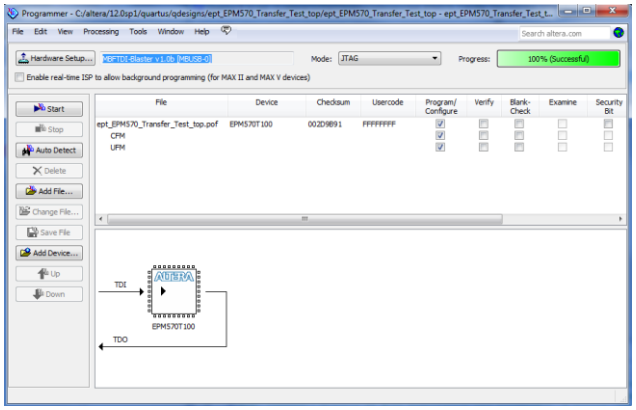
The final step is programming the “*.pof” file into the CPLD. Follow the section: “Programming the CPLD”.



- Connect the EPT-570-AP to the PC,
- Open up Quartus II,
- Open the programmer tool

- In the upper left corner of the Programmer Tool, there is a button labeled “Hardware Setup”. Verify that EPT-Blaster v1.3” has been selected. If not, go to the section JTAG DLL Insert to Quartus II and follow the directions.
- Check the box under Program/Configure
- Click the Start button.

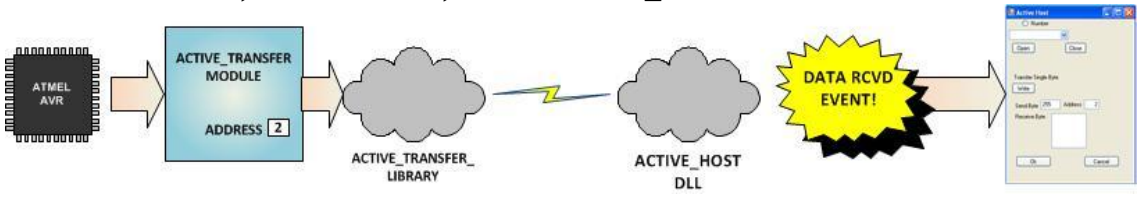
When the programming is complete, the Progress bar will indicate success.



At this point, the EPT-570-AP is programmed and ready for use.

1.1.11 PC: Design the Project

The final piece of the Data Collection Sampler is the PC application. This application will fetch the data from the CPLD of the EPT-570-AP and display it on the screen. It includes user code, windows form, and the Active_Host DLL.



The Active_Host DLL is designed to transfer data from the CPLD when it becomes available. The data will be stored into local memory of the PC, and an event will be triggered to inform the user code that data is available from the addressed module of the CPLD. This method, from the user code on the PC, makes the data transfer transparent. The data just appears in memory and the user code will direct the data to a textbox on the Windows Form.

The Data Collector project will perform the following functions.

- Find EPT-570-AP Device.
- Open EPT-570-AP Device.
- Start the Arduino data collection process.
- Wait for data from EPT-570-AP.
- Display data from EPT-570-AP in textbox.

1.1.12 PC: Coding the Project

The user code is based on the .NET Framework and written in C#. The language is great for beginners as it is a subset of the C++ language. It has the look and feel of the familiar C language but adds the ease of use of classes, inheritance and method overloading. C# is an event based language which changes the method of writing code for this project. See the section "1.1 Assembling, Building, and Executing a .NET Project on the PC" for a better description of event based language programming.

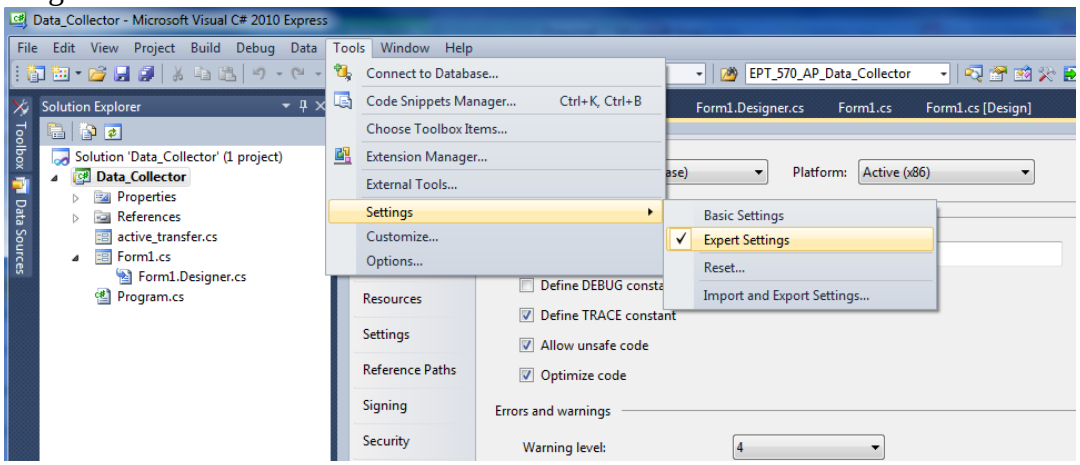
To start the project, follow the section "1.1 Assembling, Building, and Executing a .NET Project on the PC". Use the wizard to create project called "Data_Collector". When the wizard completes, the C# Express main window will look like the following.

```
using System;
using System.Drawing;
using System.Collections;
using System.Windows.Forms;
using System.Data;
using System.Threading;
using System.Runtime.InteropServices;
using System.Diagnostics;


namespace Data_Collector
{

    public partial class Data_Collector : System.Windows.Forms.Form
    {

        public Data_Collector()...
```

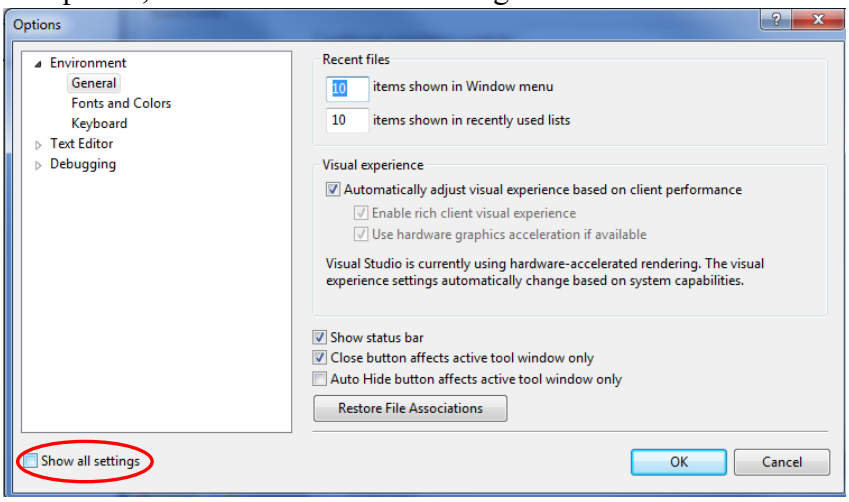
These statements setup the namespace and the class for the project. There are several other files that are created by the wizard such as Form1.Designer.cs, Program.cs, Form1.resx. We don't need to go into these support files, we will just focus on the Form1.cs as this is where all the user code goes.

The project environment must be set up correctly in order to produce an application that runs correctly on the target platform. First, we need tell C# Express to produce 64 bit

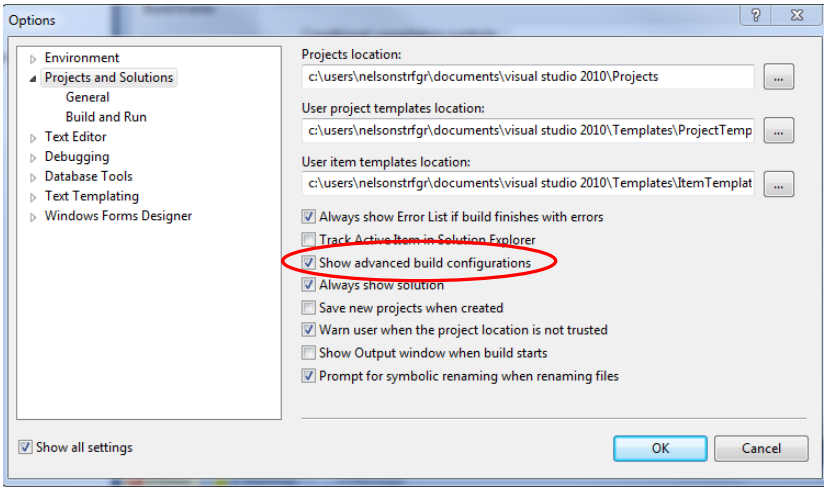
code if we are running on a x64 platform. Go to Tools->Settings and select Expert Settings



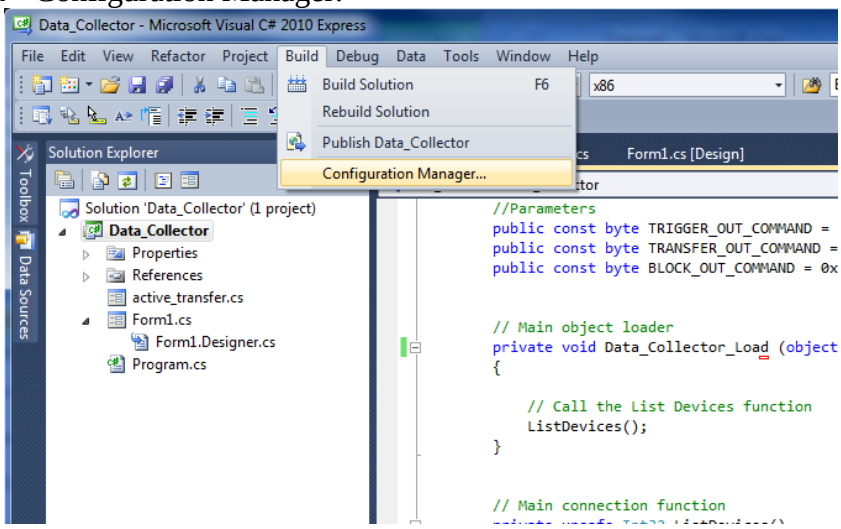
Go to Tools->Options, locate the "Show all settings" check box. Check the box.



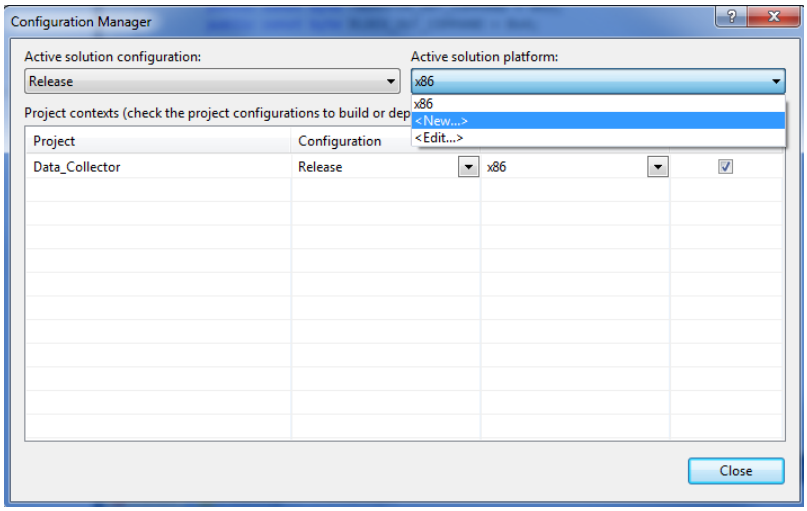
In the window on the left, go to "Projects and Solutions". Locate the "Show advanced build configurations" check box. Check the box.



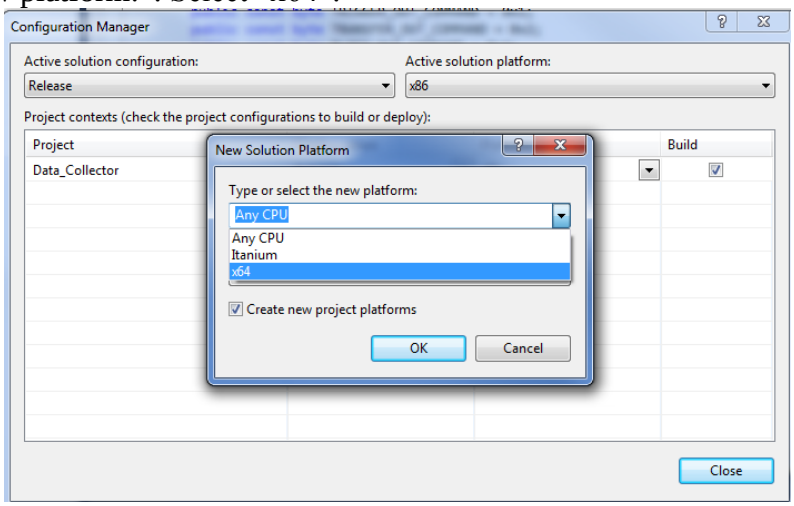
Go to Build->Configuration Manager.



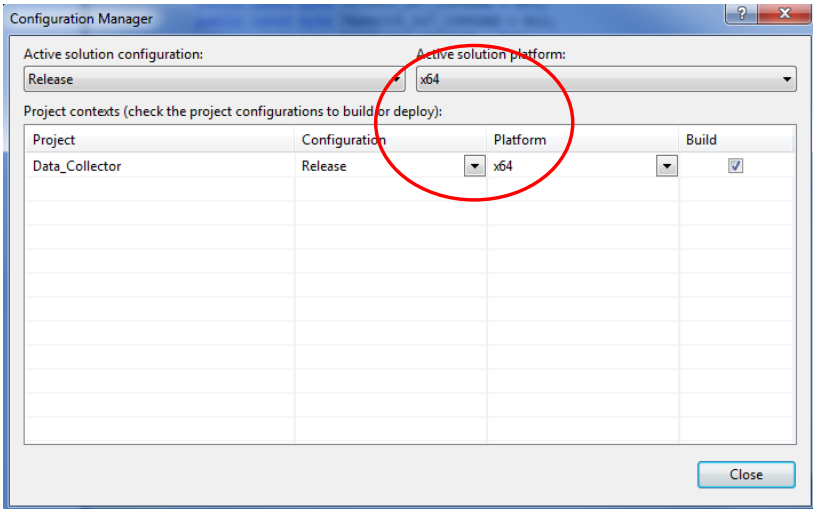
In the Configuration Manager window, locate the “Active solution platform:” label, select “New” from the drop down box.



In the New Solution Platform window, click on the drop down box under "Type or select the new platform:". Select "x64".

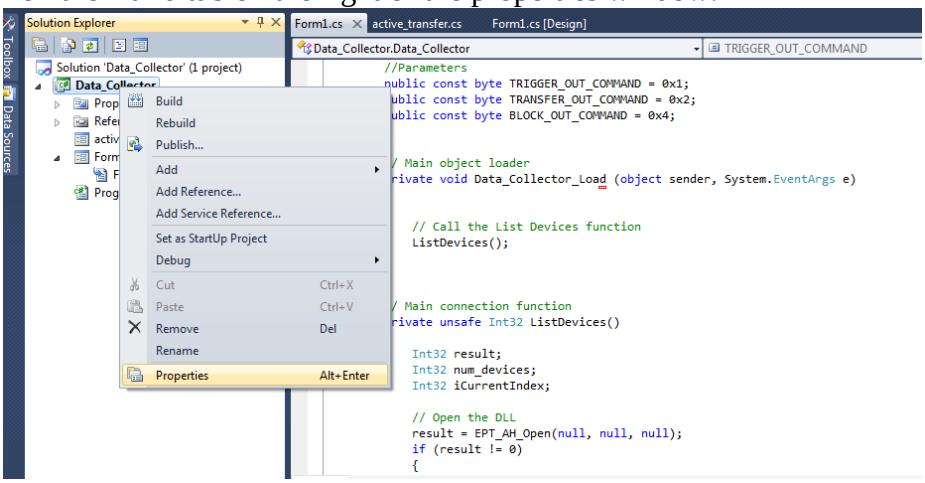


Click the Ok button. Verify that the "Active Solution Platform" and the "Platform" tab are both showing "x64".

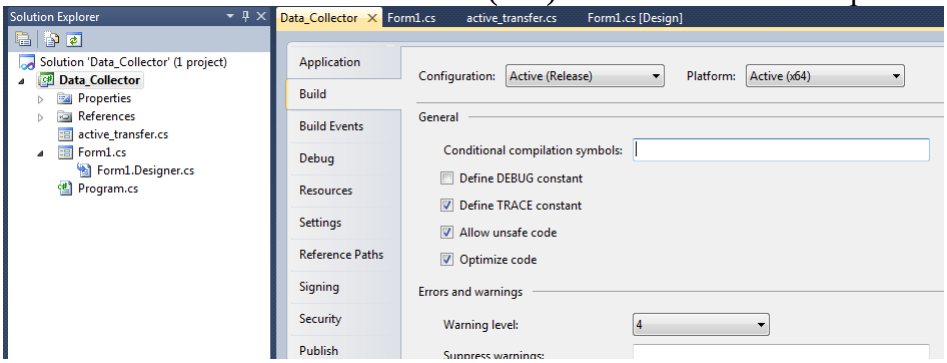


Click Close.

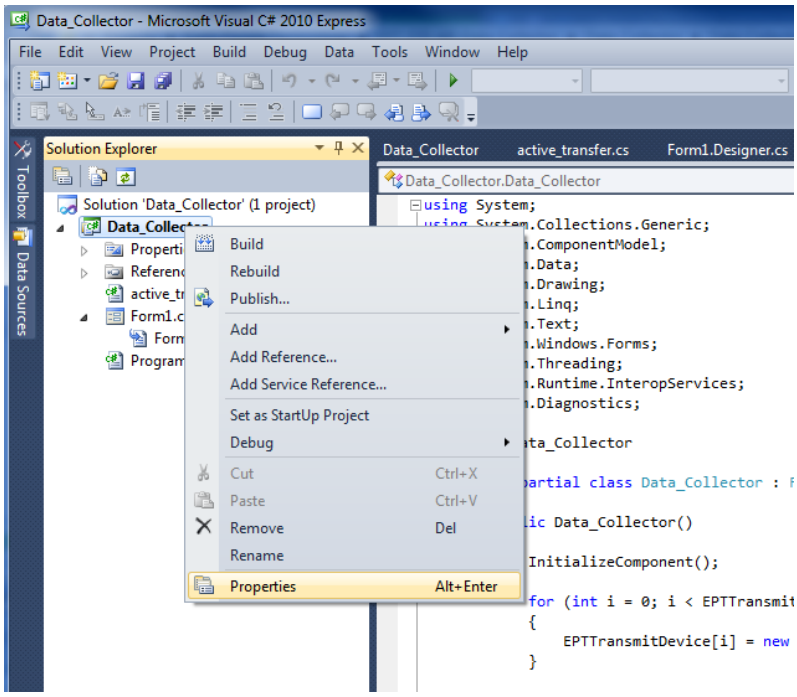
Then, using the Solution Explorer, you can right click on the project, select Properties and click on the Build tab on the right of the properties window.



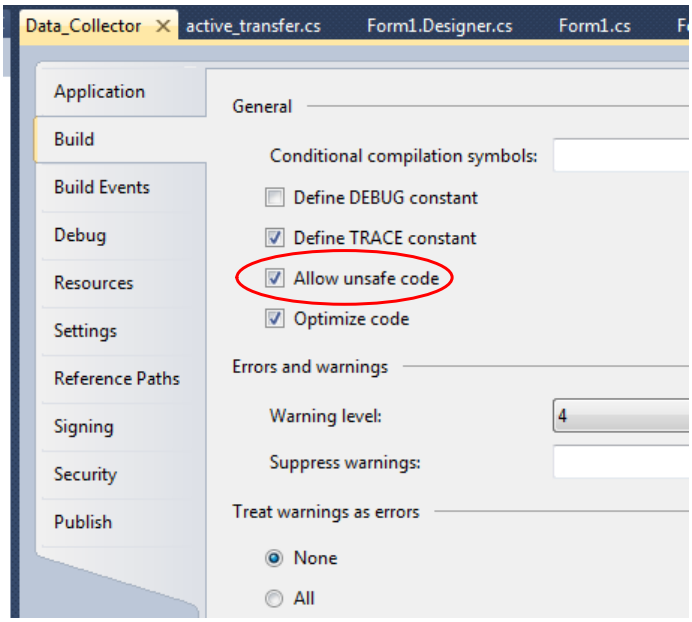
Verify that the “Platform:” label has “Active (x64)” selected from the drop down box.



Next, unsafe code needs to be allowed so that C# can be passed pointer values from the Active Host. Right click on the “Data Collector” project in the Solution Explorer. Select Properties.



Click on the Build tab and locate the "Allow unsafe code" check box. Check the box



Now we are ready to start coding.

Next, we add two classes for our device. One class stores the information useful for our device for Transmit to the EndTerms such as, address of module, length of transfer etc.

```
//Create an array of the Transfer Class for device
Transfer[] EPTTransmitDevice = new Transfer[8];
```

The next class is used to store parameters for receiving data from the device.

```
//Create a Receive object of the Transfer Class.
Transfer EPTReceiveData = new Transfer();
```

The first function called when the Windows Form loads up is the Data_Collector_Load(). This function is called automatically upon the completion of the Windows Form, so there is no need to do anything to call it. Once this function is called, it in turn calls the ListDevices().

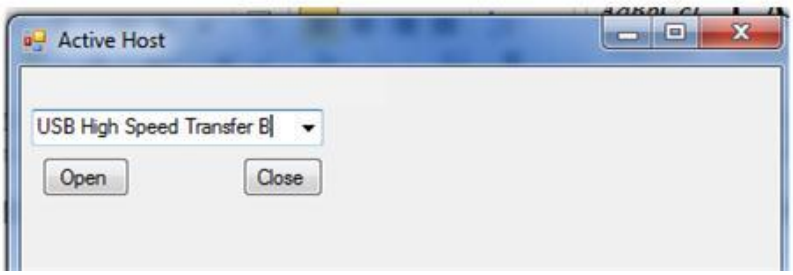
```
// Main object loader
private void Data_Collector_Load (object sender, System.EventArgs e)
   {
   // Call the List Devices function
       ListDevices();
   }
```

The ListDevices() function calls the EPT_AH_Open() function to load up the ActiveHost Dll. Next, it calls EPT_AH_QueryDevices() which searches through the registry files to determine the number of EPT devices attached to the PC. Next, EPT_AH_GetDeviceName() is called inside a for loop to return the ASCII name of each device attached to the PC. It will automatically populate the combo box, cmbDevList with all the EPT devices it finds.

```
// List Devices function
private unsafe Int32 ListDevices ()
   {
   Int32 result;
   Int32 num_devices;
   Int32 iCurrentIndex;

   // Open the DLL
   result = EPT_AH_Open(null, null, null);
   if (result != 0)
      {
      MessageBox.Show("Could not attach to the ActiveHost library");
      return 0;
      }

   // Query connected devices
   num_devices = EPT_AH_QueryDevices();

   //Prepare the Combo box for population
   iCurrentIndex = cmbDevList.SelectedIndex;
   cmbDevList.Items.Clear();

   // Go through all available devices
   for (device_index = 0; device_index < num_devices; device_index++)
      {
          String str;
          str = Marshal.PtrToStringAnsi((IntPtr)EPT_AH_GetDeviceName(device_index));
          cmbDevList.Items.Add(str);
      }
   return 0;
}
```

The user will select the device from the drop down combo box. This value can be sent to the OpenDevice() function using the button Click of the Open button.

```
// Open the device
if (EPT_AH_OpenDeviceByIndex(device_index) == false)
   {
   printf("Could not open device %s\n", EPT_AH_GetDeviceName(device_index));
   exit(0);
   }
```



The device_index variable is used to store the index of the device selected from the combo box. This variable is passed into the EPT_AH_OpenDeviceByIndex(). This

process is started by the user clicking on the "Open" button. If the function is successful, the device name is displayed in the label, labelDeviceCnt. Next, the device is made the active device and the call back function is registered using the RegisterCallBack() function. Finally, the Open button is grayed out and the Close button is made active.

```
   // Open the device
public unsafe Int32 OpenDevice()
{
    device_index = (int)cmbDevList.SelectedIndex;
    if (EPT_AH_OpenDeviceByIndex(device_index) == 0)
    {
        String message = "Could not open device " +
            Marshal.PtrToStringAnsi((IntPtr)EPT_AH_GetDeviceName(device_index)) + ", " +
            Marshal.PtrToStringAnsi((IntPtr)EPT_AH_GetDeviceSerial(device_index));
        MessageBox.Show(message);
        return 0;
    }
    else
    {
        labelDeviceCnt.Text = "Connected to device " +
            Marshal.PtrToStringAnsi((IntPtr)EPT_AH_GetDeviceName(device_index)) + ", " +
            Marshal.PtrToStringAnsi((IntPtr)EPT_AH_GetDeviceSerial(device_index));
    }

    // Make the opened device the active device
    if (EPT_AH_SelectActiveDeviceByIndex(device_index) == 0)
    {
        String message = "Error selecting device: %s " +
            Marshal.PtrToStringAnsi((IntPtr)EPT_AH_GetLastError());
        MessageBox.Show(message);
        return 0;
    }

    // Register the read callback function
    RegisterCallBack();
    btnOpenDevice.Enabled = false;
    btnCloseDevice.Enabled = true;
    return 0;
}
```

Next, the callback function is populated. This function will be called from the Active Host dll. When the EPT Device has transferred data to the PC, the callback function will do something with the data and command.

```
// Actual callback function which will read messages coming from the EPT device
unsafe void EPTReadFunction(Int32 device_id, Int32 device_channel, byte command, byte payload,
{
    byte* message = data;

    // Select current device
    EPT_AH_SelectActiveDeviceByIndex(device_id);

    //Add command and device_channel to the receive object
    EPTReceiveData.Command = ((command & COMMAND_DECODE) >> 3);
    EPTReceiveData.Address = device_channel;

    //Check if the command is Block Receive. If so,
    //use Marshalling to copy the buffer into the receive
    //object
    if (EPTReceiveData.Command == BLOCK_OUT_COMMAND)
    {
        EPTReceiveData.Length = data_size;
        EPTReceiveData.cBlockBuf = new Byte[data_size];

        Marshal.Copy(new IntPtr(message), EPTReceiveData.cBlockBuf, 0, data_size);
    }
    else
    {
        EPTReceiveData.Payload = payload;
    }
    EPTParseReceive();
}
```

Because the callback function communicates directly with the dll and must pass pointers from the dll to the C#, marshaling must be used. Marshaling is an advanced topic and will not be covered in this manual.

When EPTReadFunction() callback is called and passed parameters from the Active Host dll, it populates the EPTReceiveData object. It then calls EPTParseReceive() function. This function uses a case statement to call the TransferOutReceive() function.

```
private void EPTParseReceive()
{
    switch (EPTReceiveData.Command)
    {
        case TRANSFER_OUT_COMMAND:
            TransferOutReceive();
            break;
        default:
            break;
    }
}
```

TransferOut Receive() creates a string from the EPTReceiveData.Payload parameter. Then sends the string to the textbox, tbDataBytes.

```
public void TransferOutReceive()
{
    string WriteRcvChar = "";
    WriteRcvChar = String.Format("{0}", (int)EPTReceiveData.Payload);
    tbDataBytes.AppendText(WriteRcvChar + " ");
}
```

Controls such as buttons are added to the Form1.cs[Design] window which allow turning on and off signals. These include

- btnWriteByte
- btnTransferReset
- btnOk
- btnClose
- btnResetBlock

Refer to section 1.6.4 Adding Controls to the Project for details about using the ToolBox to place controls on a design. The btnWriteByte click event calls the EPT_AH_SendTransferControlByte(). This function is used to turn on/off bits in the Control Register in the CPLD code. The btnWriteByte will set the start_stop_cntrl signal in the CPLD to one. This signal starts the Arduino Data Collector sending its random word to the CPLD.

```
private void btnWriteByte_Click(object sender, EventArgs e)
{
    int address_to_device;
    address_to_device = Convert.ToInt32(tbAddress.Text);
    EPT_AH_SendTransferControlByte((char)2, (char)1);
}
```

The btnTransferReset sets the start_stop_cntrl bit in the Control Register to zero. This action will cause the Arduino Data Collector to stop sending the random word to the CPLD.

```
private void btnTransferReset_Click(object sender, EventArgs e)
{
    int address_to_device;
    address_to_device = Convert.ToInt32(tbAddress.Text);
    EPT_AH_SendTransferControlByte((char)address_to_device, (char)0);
}
```

The btnResetBlock button will clear the tbDataBytes textblock. The Clear() method is inherited from the textbox class.

```
private void btnResetBlock_Click(object sender, EventArgs e)
{
    tbDataBytes.Clear();
}
```

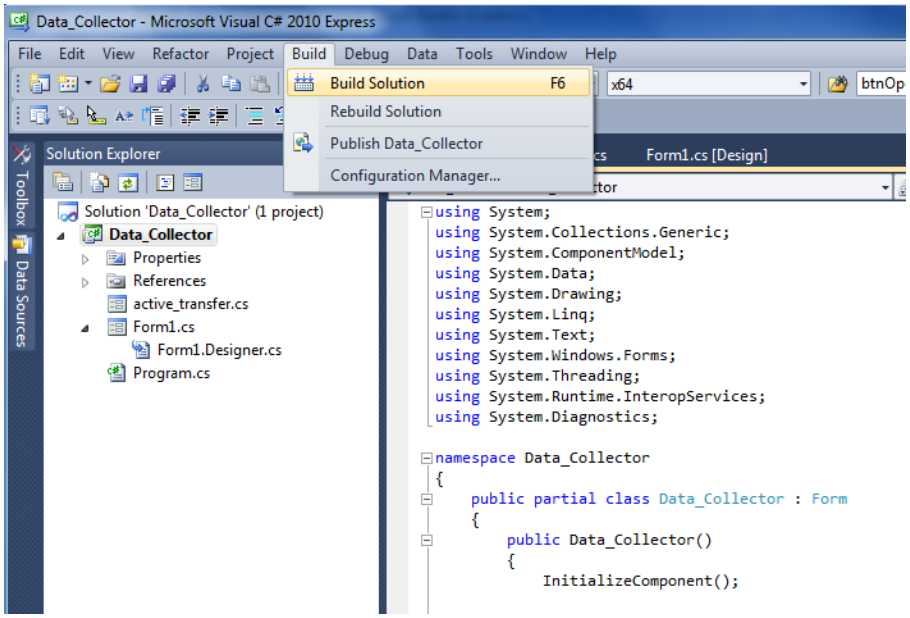
The btnOk and btnClose buttons are used to end the application. It calls the function EPT_AH_CloseDeviceByIndex() to remove the device from the Active Host dll. The buttons btnOpen and btnClose have their Enabled parameter set to true and false respectively. The Enabled parameter controls whether the button is allowed to launch an event or not. If it is not enabled, the button is grayed out. At the end of each click event, the Application.Exit() method is called. This exits the form.

```
private void btnOk_Click(object sender, EventArgs e)
{
    EPT_AH_CloseDeviceByIndex(device_index);
    btnOpenDevice.Enabled = true;
    btnCloseDevice.Enabled = false;

    lblDeviceConnected.Text = "";
    Application.Exit();
}

private void btnCancel_Click(object sender, EventArgs e)
{
    EPT_AH_CloseDeviceByIndex(device_index);
    btnOpenDevice.Enabled = true;
    btnCloseDevice.Enabled = false;

    lblDeviceConnected.Text = "";
    Application.Exit();
}
```

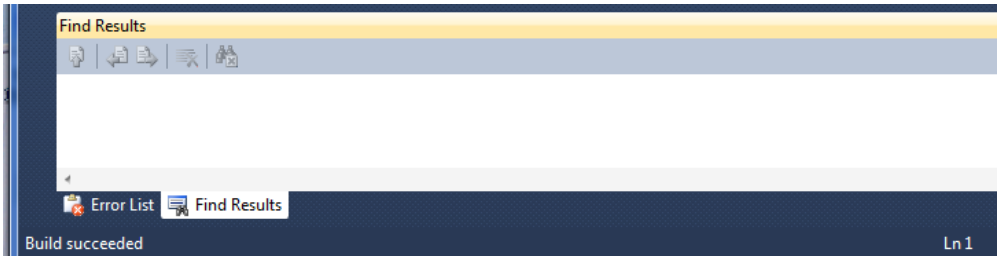
This is all that is needed for the Data Collector project. The Arduino will generate a random 8 bit word. It then transmits that word to the CPLD using the A0 (WRITE_ENABLE) signal. The CPLD transmits the 8 bit word to the PC using the ACTIVE TRANSFER module of the Active_Transfer Library. The dll reads the 8 bit word into local memory. It then calls the Callback function, EPTReadFunction. The 8 bit is finally displayed to screen using the MessageBox.Show().

1.1.13 PC: Compiling the Active Host Application

Building the Data_Collector project will compile the code in the project and produce an executable file. It will link all of the functions declared in the opening of the Data_Collector Class with the Active Host dll. The project will also automatically link the FTD2XX.dll to the object code. To build the project, go to Debug->Build Solution.



The C# Express compiler will start the building process. If there are no errors with code syntax, function usage, or linking, then the environment responds with "Build Succeeded".

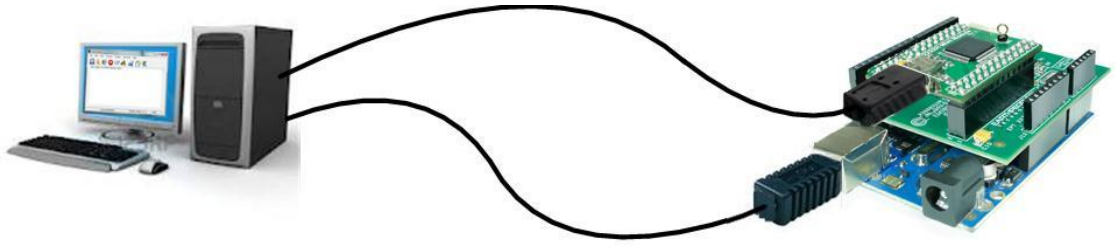


If the build fails, you will have to examine each error in the "Error List" and fix it accordingly. If you cannot fix the error using troubleshooting methods, post a topic in the Earth People Technology Forum. All topics will be answered by a member of the technical staff as soon as possible.

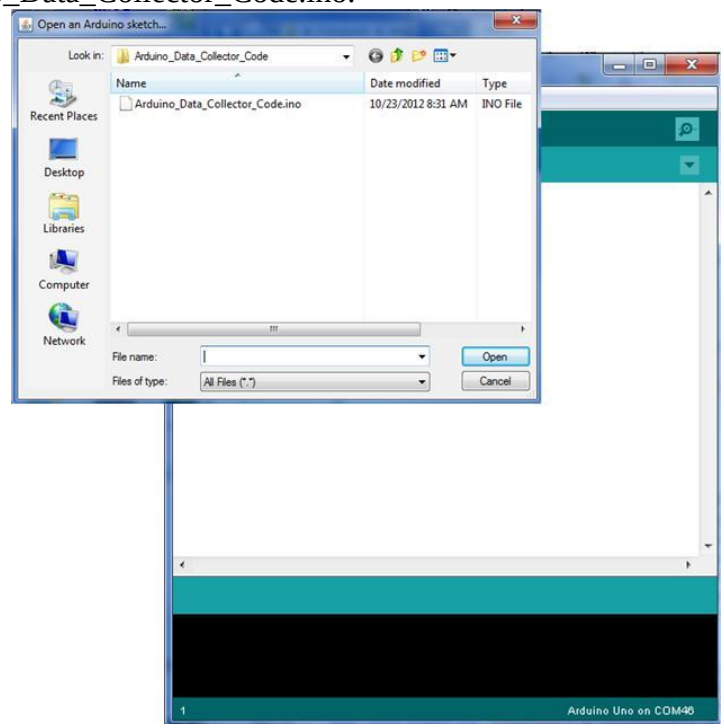
At this point, the environment has produced an executable file and is ready for testing. Next, we will connect everything together and see it collect data and display it.

1.1.14 Connecting the Project Together

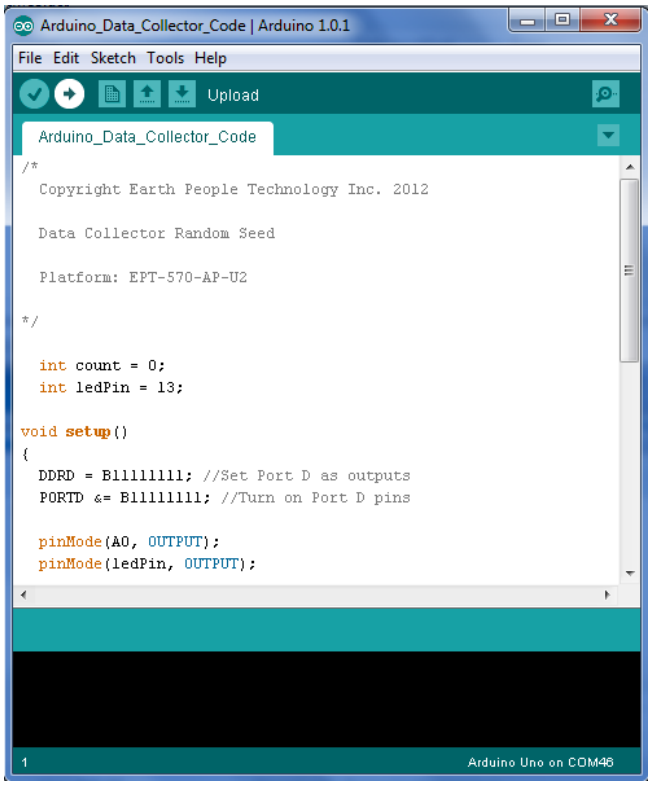
Now we will connect the Arduino, EPT 570-AP-U2, and the PC to make a Data Collector. First, connect a USB cable from a USB port on the PC to the Arduino. Second, connect a USB cable from a open USB port on the PC to the EPT 570-AP-U2.



Next, open the Arduino IDE and select File->Open and select your sketch created earlier, Arduino_Data_Collector_Code.ino.



Select the file and click Open. The sketch will now populate the Arduino IDE window. Compile and Download the sketch into the Arduino microcontroller using the Upload button.

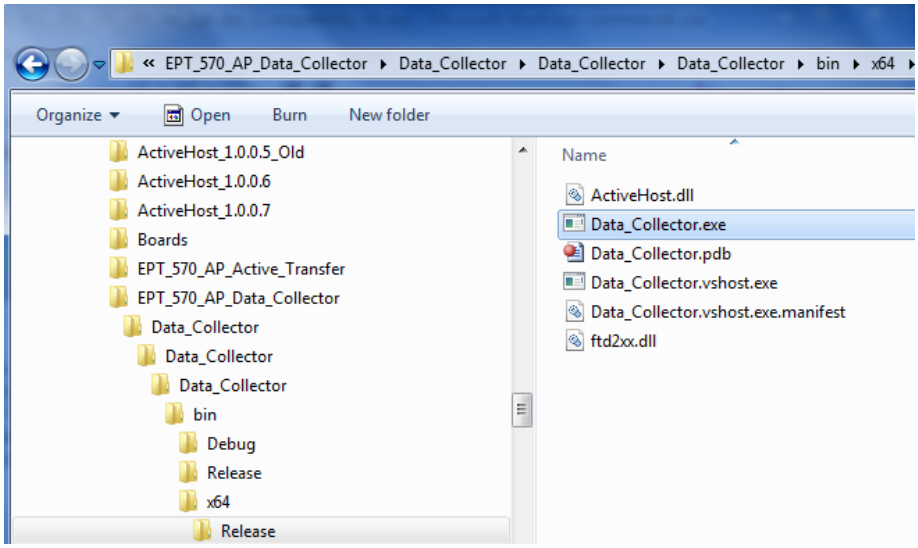


The Arduino IDE will compile the project, then transmit the machine level code into the ATMega328 SRAM to start the program. When this is complete, the Yellow L LED will blink about once per second.

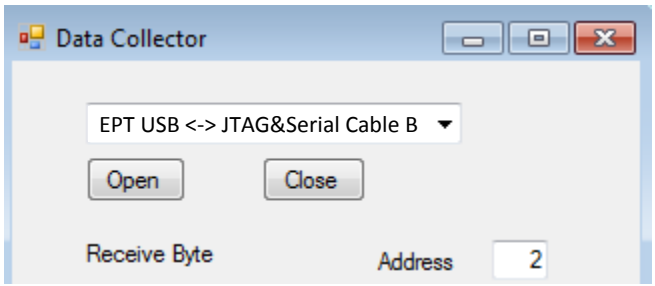
If this LED is blinking at the rate of once per second, the Arduino and the Data Collector project are ready for the EPT 570-AP-U2 code.

The CPLD should already be programmed with its Data Collector Project. If it isn't, follow the instructions in section 3.1.10.

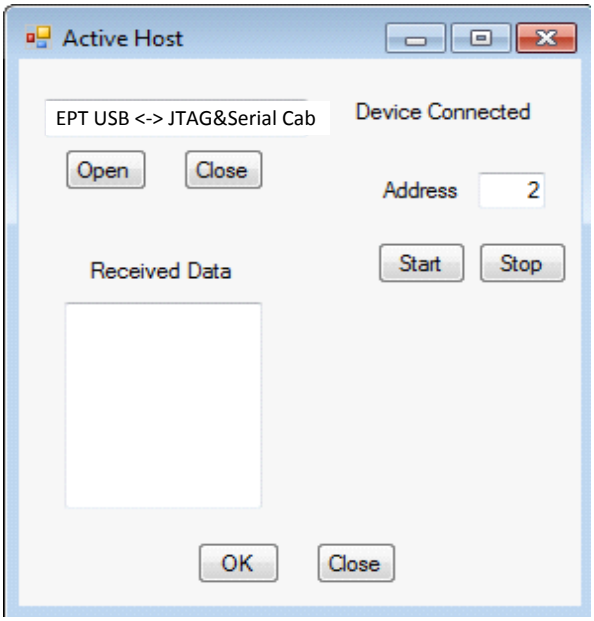
Open the EPT Data Collector on the PC by browsing to the Data Collector project folder. Locate the executable in the \bin\x64\Release folder.



Initiate the application by double clicking the application icon in the \Release folder of the project. The application will open and automatically load the Active Host dll. The application will locate the EPT 570-AP-U2 device. Next, the combo box at the top will be populated with the name of the device.

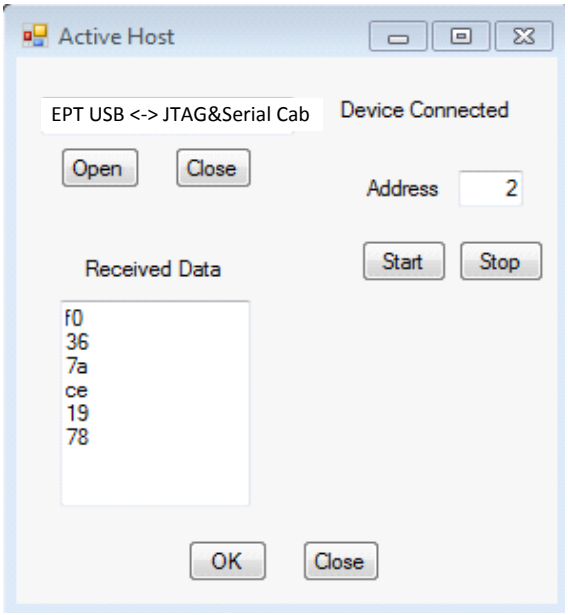


Select the EPT 570-AP device and click the Open button. If the Active Host application connects to the device, a label will indicate “Device Connected”. Next, select the address of the Active Transfer module in the CPLD. In our case it is “2”.



1.1.15 Testing the Project

To test our Data Collector project, just click on the Start button. As soon as the device connects, the data from the Arduino will appear in the received data textBox.



And that's all there is to the Data Collector Project. It's up to the user to use this project as a base to create much larger projects. You can easily make a volt meter using this project by turning off the Random number generator in the Arduino and reading the

Analog Pins. Also, reformat the textBox display that it shows in decimal instead of the Hexadecimal display.