DueProLogic Arduino Communications Project

This project guide describes how to assemble all the code necessary for communications between the Arduino Due and the DueProLogic. The project is called the Data Collector and uses the Due to generate a random eight bit number, transmit s the number to the DueProLogic, and the DueProLogic transmits the value to the PC for display.The project has three individual code projects:

- Arduino *.ino project
- DueProLogic FPGA project
- Data Collector Visual C# project

Table of Contents

1 Introduction and General Description

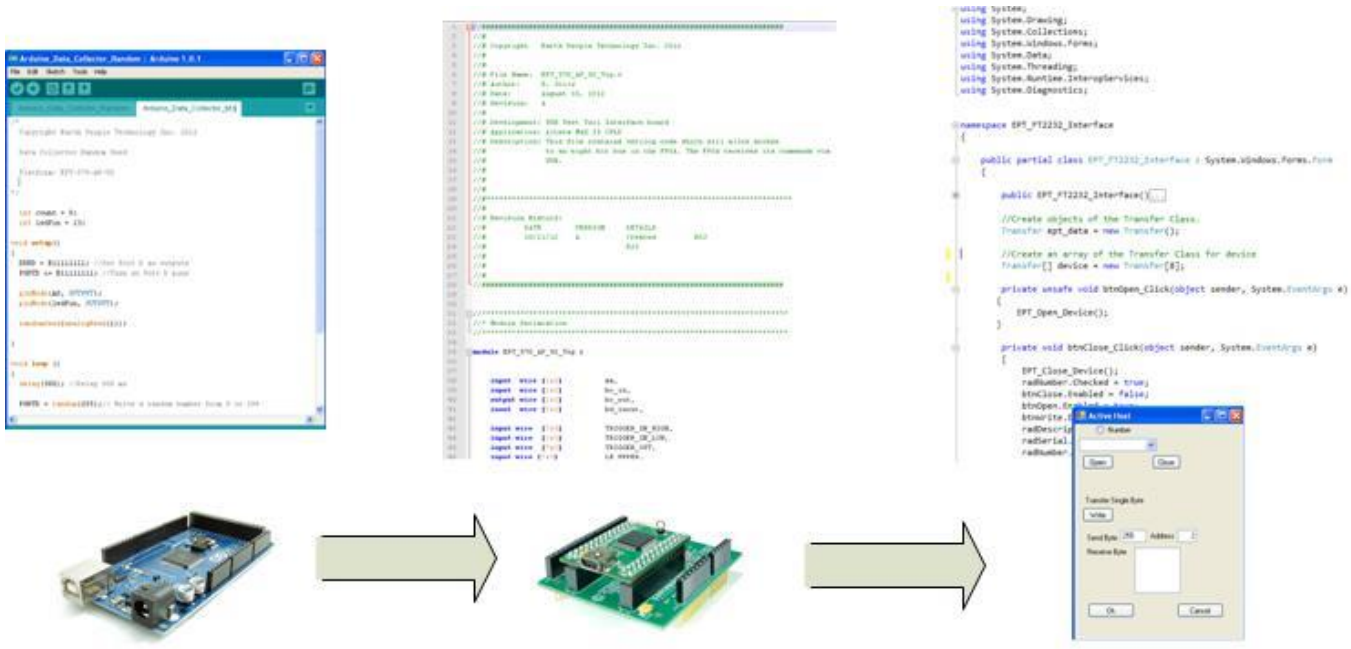
This project guide is separated into the hardware description and the code development process.

2 FPGA 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, you could use any combination of these two extremes.

2.1 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-4CE6-AF-D1 FPGA, 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.

2.1.1 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.

2.1.2 Create Data Generator

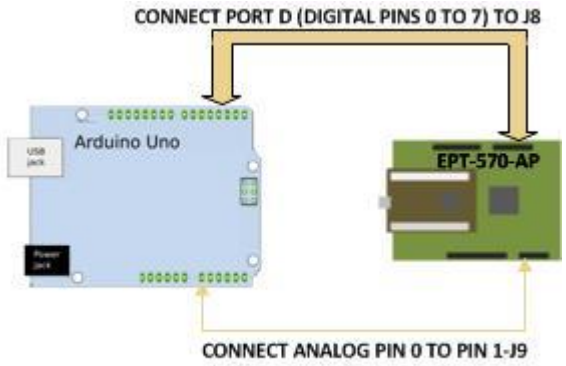
To keep the design simple, no external data source will be used. We will create a data source using the Arduino, then transmit this data to the EPT-4CE6-AF-D1 board. To create the data source, we will use the random() function. This function generates pseudo random numbers from a seed value. We will 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));
}
```

2.1.3 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-4CE6-AF-D1. There is also a one bit control line which will be used to inform the FPGA 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 Output Write Enable Register (OWER) determines whether the pin is an INPUT or OUTPUT. The Output Data Set Register (ODSR) controls whether the pin output is HIGH or LOW, and the Output Write Disable Register sets selected pins to disable write.

PORTC maps to Arduino digital pins 33 to 41 on the 2x18 connector of the Due

REG_PIOC_OWER - The Port C Write Enable Register - read/write
REG_PIOC_ODSR - The Port C Data Register - read/write
REG_PIOC_OWDR - Disable writing to all other pins on the same port

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.

```
void setup()
{

   int k ;

   //Set pins D33-D41 as outputs
     for( k = 33 ; k <= 41 ; k++ )
      pinMode( k, OUTPUT ) ; // Sets Port C2-C9 to output pins

    // Set the output pins in the Output Write Enable Register
    REG_PIOC_OWER =  0x0000037E ;

    // Disable writing to all other pins on the same port-
    REG_PIOC_OWDR = 0xFFFFFC81 ;


  pinMode(C_Enable, OUTPUT);
  pinMode(ledPin, OUTPUT);
  pinMode(LEDExt, OUTPUT);
  pinMode(inPin8, INPUT);

  randomSeed(analogRead(1));

  Serial.begin(115200);


}
```

After the setup() function executes, the PORT C is ready to be assigned the results of our random() function. And the C_Enable pin will be used to latch the value on PORT C pins into the FPGA.

2.1.4 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-4CE6-AF-D1. This output pin (from the EPT-4CE6-AF-D1) becomes an input to the Arduino and is used in conditional switch.

```
void loop ()
{

   int i ;


   //Sample the Start/Stop switch
   //from the EPT-4CE6-AF
   startStopBit = digitalRead(inPin8);

  delay(newP); //Delay in ms

   if(startStopBit)
   {
     // Write a random number from 0 to 299
     //to the input of the EPT-4CE6-AF
      new_value = random(255);
```

This code will sample the Start/Stop switch which is an output from the EPT-4CE6-AF-D1 on J10 PIN 4. On the Arduino, this is PIN 11 of the Digital pins. Each iteration of the loop() function, the startStopBit variable stores the state of DigitalPin17. 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-4CE6-AF-D1. 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 on PIN 13 of the Digital Pins. This will blink the LED on the DUE so that we can have a visual indication that the project is working.

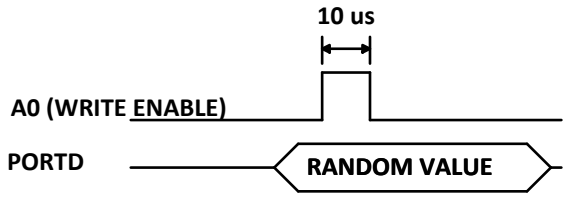
We want to add a delay so that the data from the generated displays on the Windows PC long enough for our eyes to verify that the data is updating correctly. This delay should be one second in total. So, the data will change then stay stable in the textbox for one second before changing again.

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:

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

delay(newP); //Delay in ms

//Set the LED Pin Low
digitalWrite(ledPin, LOW);
digitalWrite(LEDExt, LOW);
```

Notice that PORT C 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 C_Enable write enable signal for the FPGA has back to back instructions turning it on then off immediately. Because the DUE SAM3 chip takes approximately 160 clock cycles to execute the digitalWrite() function and affect the Pin at C_Enable, this produces a write enable pulse of 10 microseconds.



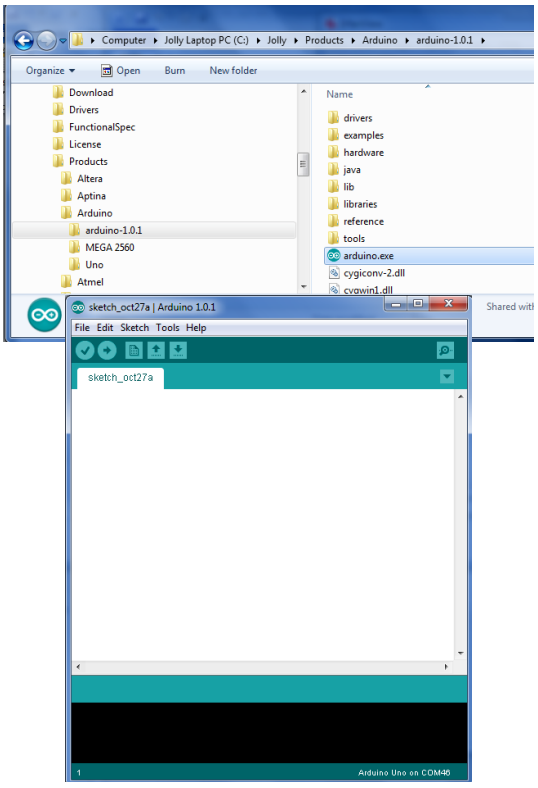
The RANDOM VALUE will be stable before the C_Enable (WRITE ENABLE) asserts thus guaranteeing a successful transfer of data from Arduino to FPGA.

2.1.5 Building Arduino Project

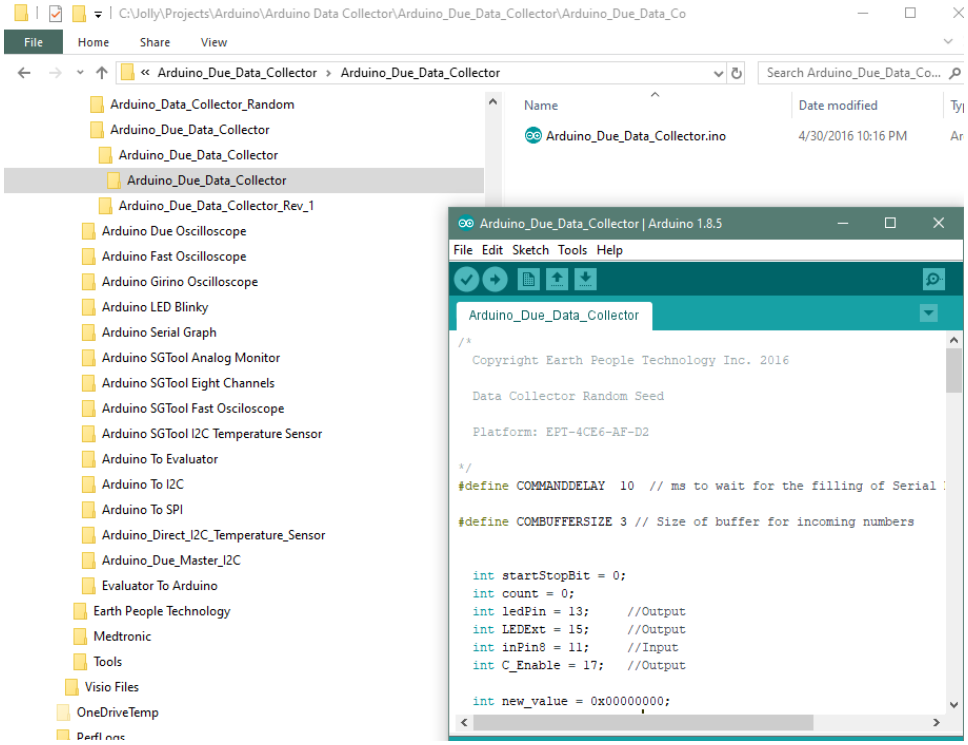
Building the Arduino project is the process of converting (compiling) the code you just wrote into machine level code that the processor can understand. The Arduino IDE is the software tool that does the compiling. The machine level code is a set of basic instructions that the processor uses to perform the functions the user code. Browse to the \Projects_Arduino\Arduino_Data_Collector_Code\ folder of the EPT FPGA Development System DVD. Copy Arduino_Due_Data_Collector.ino .

To compile your code,

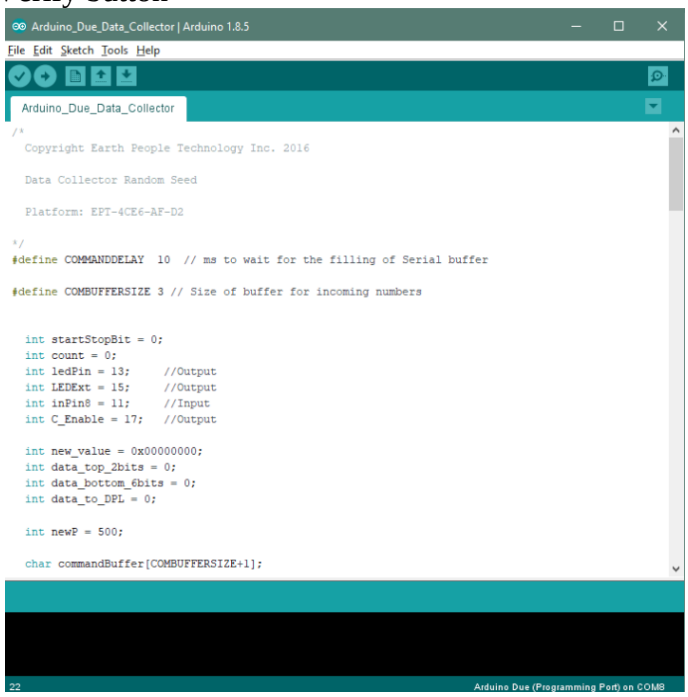
- Open up the Arduino IDE



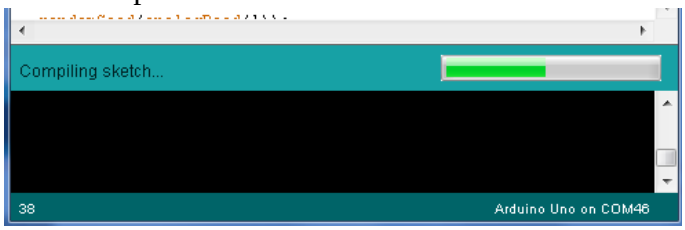
- Load your code into the Sketch.



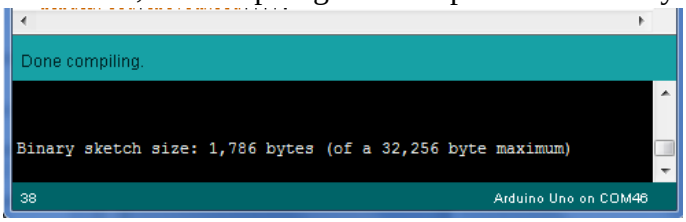
- Click the Verify button



- The sketch will compile



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



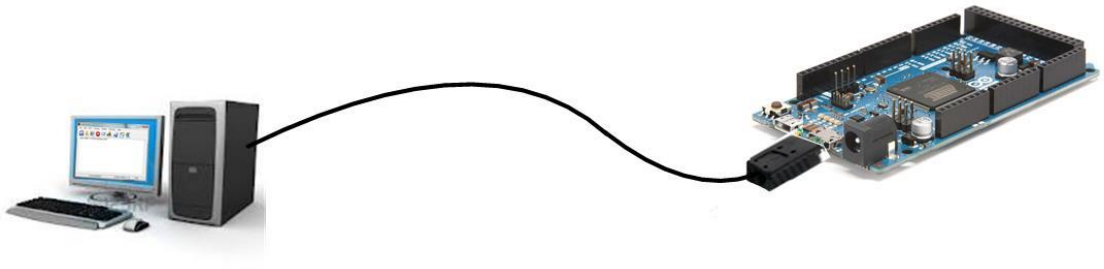
Now we are done with compiling and ready to program the Arduino

2.1.6 Programming the Arduino

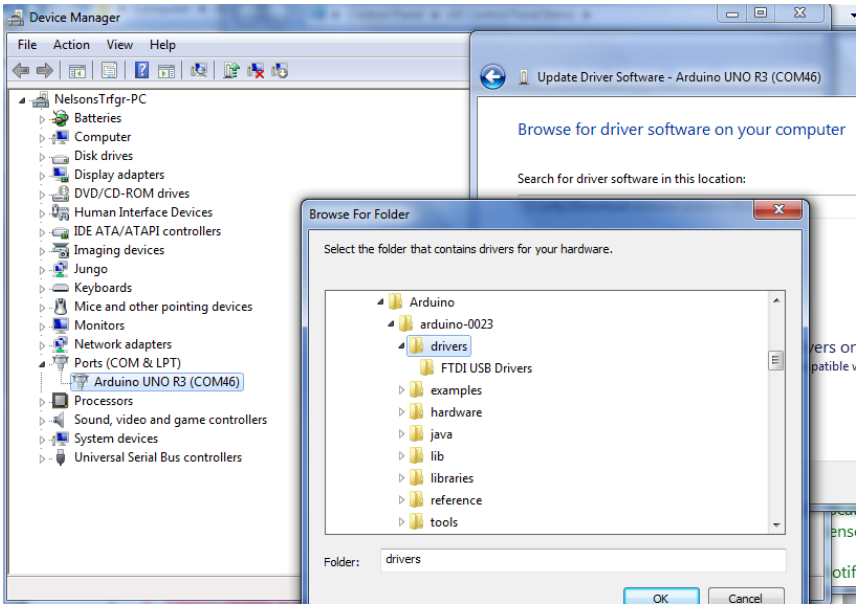
Programming the Arduino is the process of downloading the user's compiled code into the Flash memory of the Atmel ATMega328 chip. Once the code is downloaded, the Arduino IDE resets the chip and the processor starts executing out of Flash memory.

To program the Arduino

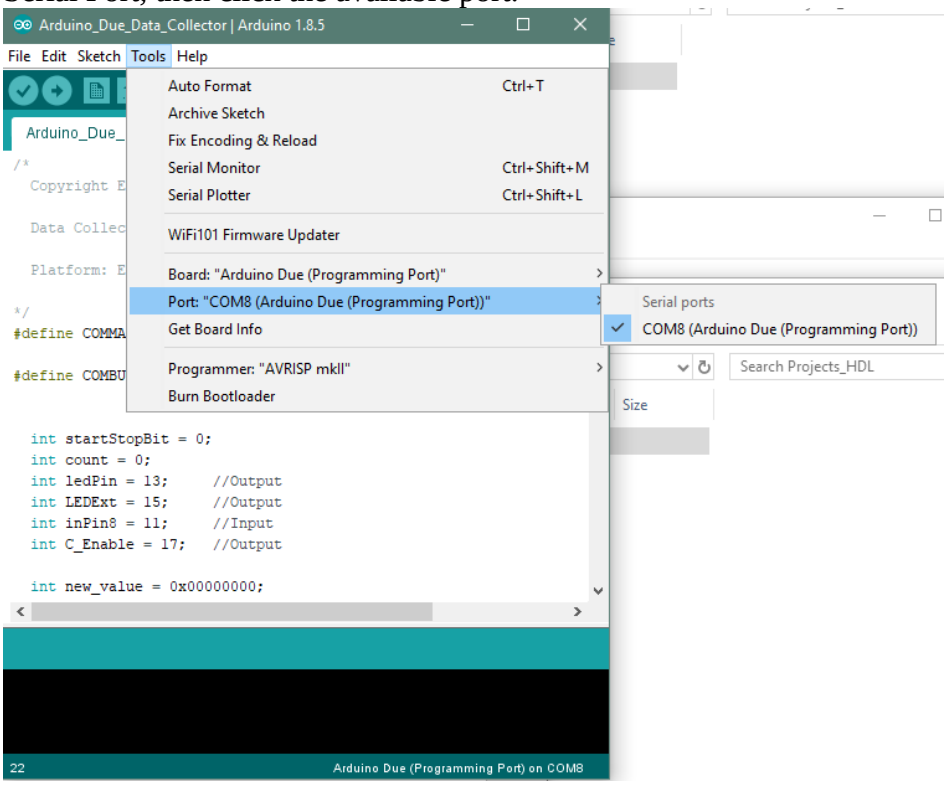
- Connect the USB cable from PC to Arduino



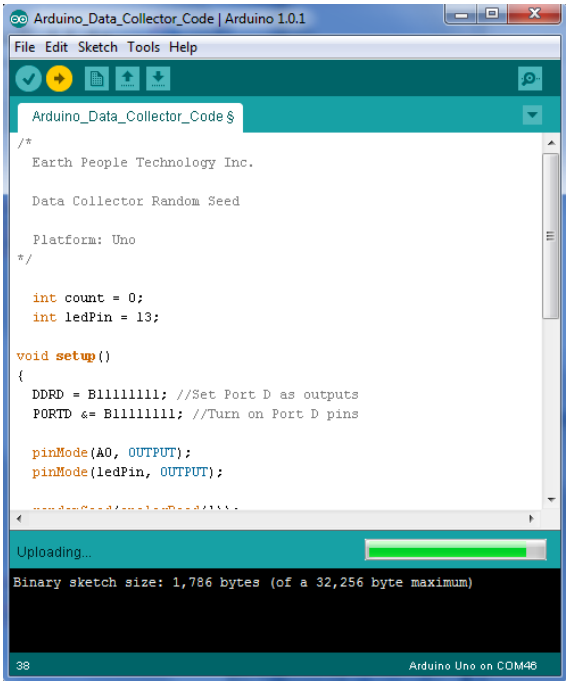
- Load the Arduino USB driver according to the manual
- Plug in your board and wait for Windows to begin it's driver installation process. After a few moments, the process will fail, despite its best efforts
- Click on the Start Menu, and open up the Control Panel.
- While in the Control Panel, navigate 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 Due (COMxx)"
- Right click on the "Arduino Due (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 Due's driver file, named **"ArduinoDUE.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.



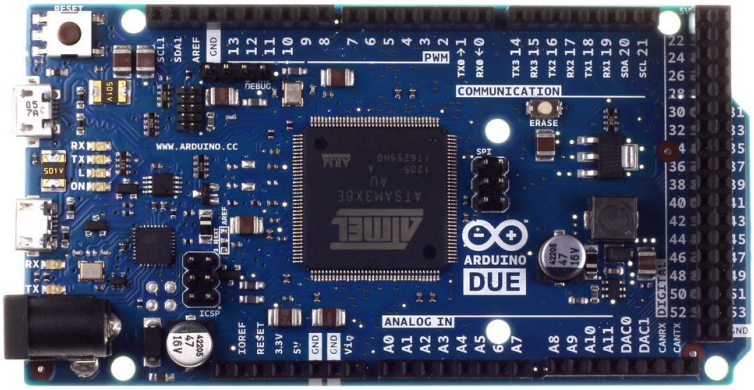
- Once the driver is loaded, we can set the COM Port. Click on Tools and select Serial Port, then click the available port.



- 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.

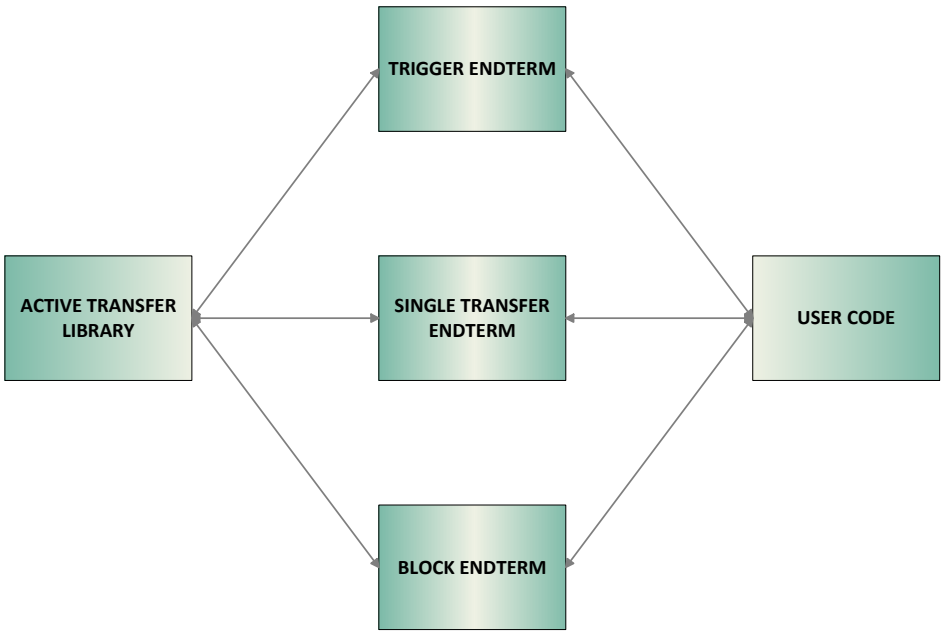


2.1.7 FPGA Active Transfer EndTerm Coding and Initiation

The EPT-4CE6-AF-D1 will accept the data collected by the Arduino and transfer it to the PC. It is designed to plug directly into the Arduino Due and there is no need for external wires to be added. The Active Transfer EndTerms are used to connect the Active Transfer Library to the user code. This makes it easy to transfer data to and from the PC via the USB. The user needs to create a state machine to control the transfer between the incoming data and the Active Transfer EndTerms. We will now go through exercise of creating the FPGA code for the Data Collector Sampler.

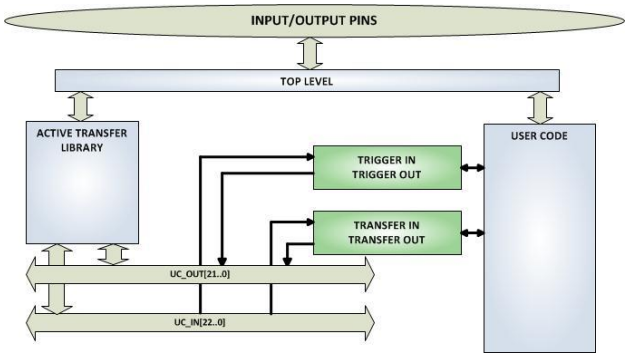
2.1.8 FPGA: Define the User Design.

In this step we will define the user's code and include EndTerms and 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 EndTerms 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 Project DVD.



We will build our FPGA project using Quartus II software from Altera. The primary file defining the user's FPGA project is named "EPT_4CE6_AF_D1_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 FPGA, the Pin Planner tool is used. This tool allows the user to match internal net names to the pins of the FPGA.

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 C_Enable (write enable) signal from the Arduino can latch the data into the active_transfer module.



FPGA: 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 FPGA. 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;
```

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.

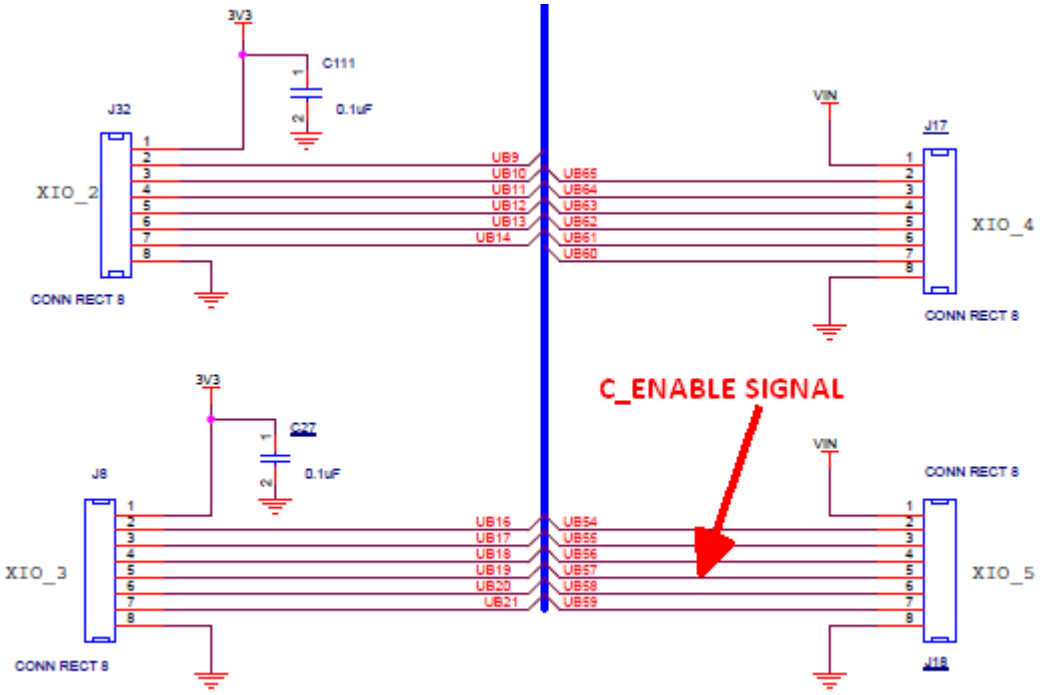
```
//LED3 is used to signify to the  user that the Start
//switch is enabled
assign              LED3 =  ~start_stop_cntrl;
```

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

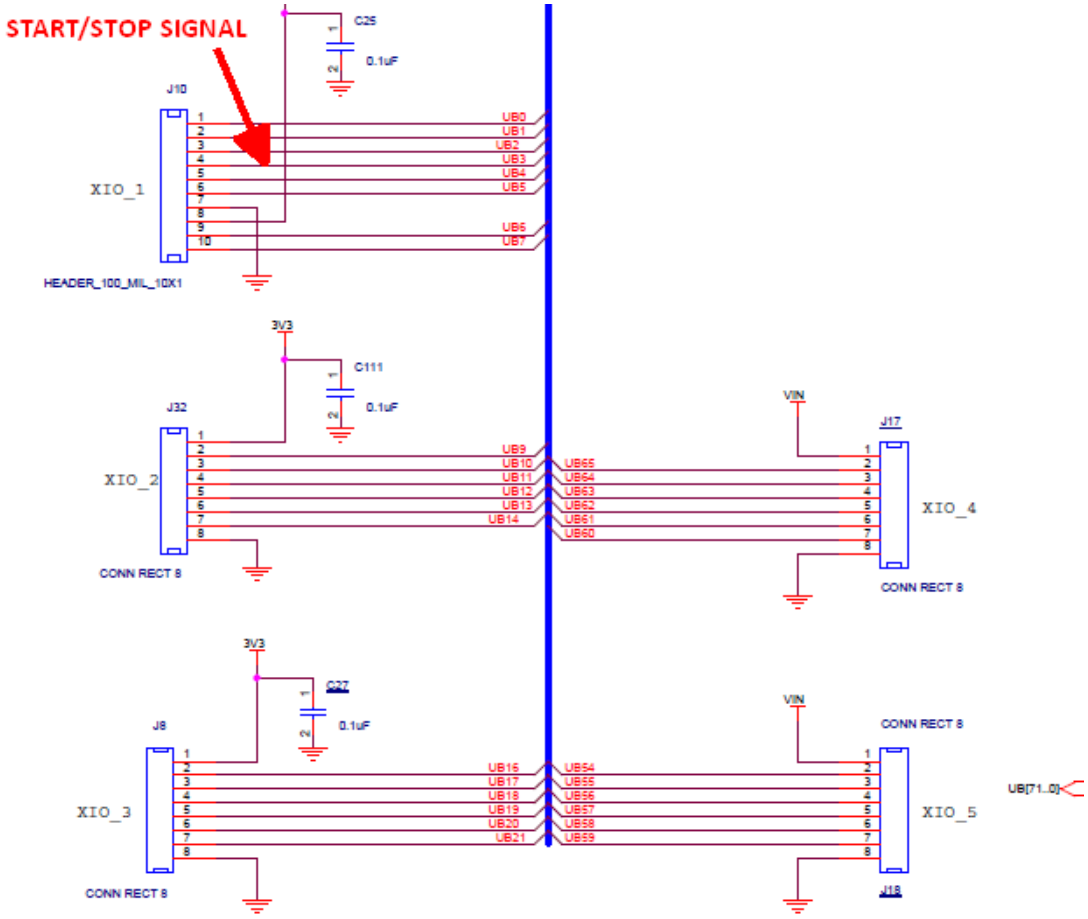
- transfer_write_reg –This is a latch register to hold the state of the C_Enable (Write Enable)
- transfer_write –This register is used to start the active_transfer single byte write to the PC.
- transfer_write_byte –This is an 8 bit register to hold the value of the Data Collection output.

This block will compare the input signal on C_Enable to a high. This signal is derived from Pin 5 of J18 which is routed to the C_Enable (Write Enable) of the Arduino DUE. 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 data_from_arduino register (this register is directly connected to pins 33-41 on the Arduino DUE) 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 C_Enable (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 C_Enable (Write Enable) to assert high.

```
//-------------------------------------------------
// 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;
    end
    else
    begin
        if(c_enable & !transfer_write_reg)
        begin
            transfer_write_reg <= 1'b1;
            transfer_write <= 1'b1;
            transfer_write_byte <= data_from_arduino;
        end
        else if(c_enable & transfer_write_reg)
        begin
            transfer_write_reg <= 1'b1;
            transfer_write <= 1'b0;
        end
        else if(!c_enable & transfer_write_reg)
        begin
            transfer_write_reg <= 1'b0;
            transfer_write <= 1'b0;
            transfer_write_byte <= 0;
        end
        else
        begin
        end
    end
 end
```



This block of code takes care of reading the random word from the Arduino using the C_Enable (Write Enable) Pin. However, because the Arduino is expecting a Start/Stop bit on Digital Pin8, the FPGA code has to provide this bit. This presents a problem, the EPT-4CE6-AF-D1 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 FPGA for the random word and the C_Enable (Write Enable) Pin. So, the third port can be used as the output port.



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 DigitalPin11 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 C_Enable (Write Enable) Pin in its conditional statement. The C_Enable (Write Enable) Pin will cause the FPGA code to enter into its first conditional statement.

Next, we add a TRANSFER_CONTROL state machine to read the Control Register from the Host PC using the active_transfer EndTerm.
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	

```
*/
active_control_register      ACTIVE_CONTROL_REG_INST
(
 .CLK                        (CLK_66),
 .RST                        (RST),
 .TRANSFER_IN_RECEIVED       (transfer_in_received),
 .TRANSFER_IN_BYTE           (transfer_in_byte),

 .CONTROL_REGISTER           (control_register)
 );
```

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.

2.1.9 FPGA: Compile/Synthesize the Project

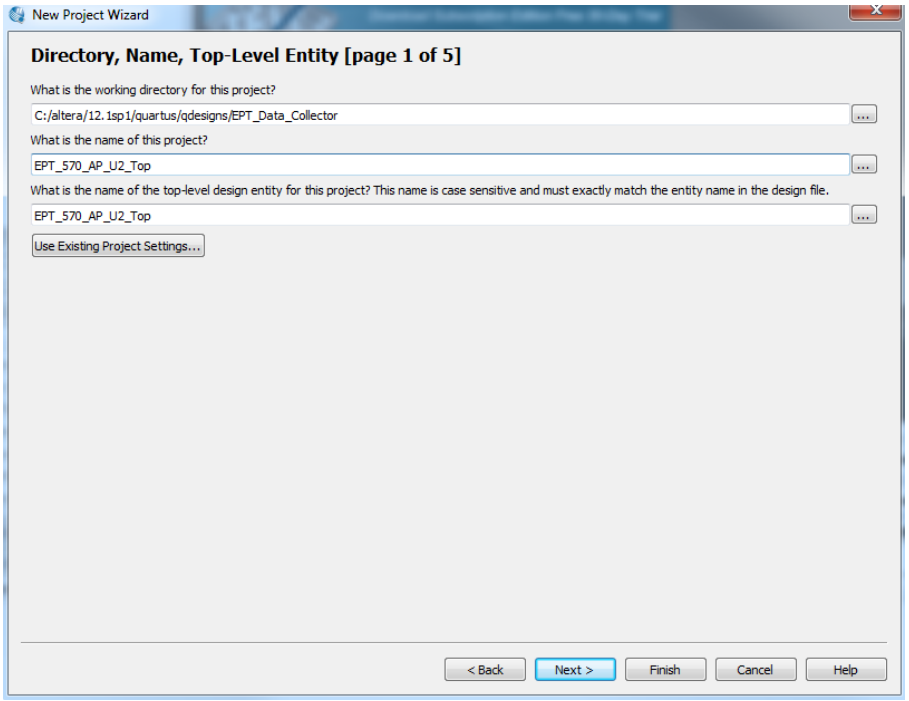
The Quartus II application will compile/ synthesize the user code, active_transfer_library, and the active EndTerms. The result of this step is a file containing the FPGA code with "*.pof". First, we need to create a project in the

Quartus II environment. Follow the directions in the section: “Compiling, Synthesizing, and Programming FPGA”.

Bring up Quartus II, then use Windows Explorer to browse to c:/altera/xxx/quartus/qdesigns create a new directory called: “EPT_Data_Collector”.

Open Quartus II by clicking on the icon . 

Under Quartus, Select File->New Project Wizard. The Wizard will walk you through setting up files and directories for your project.

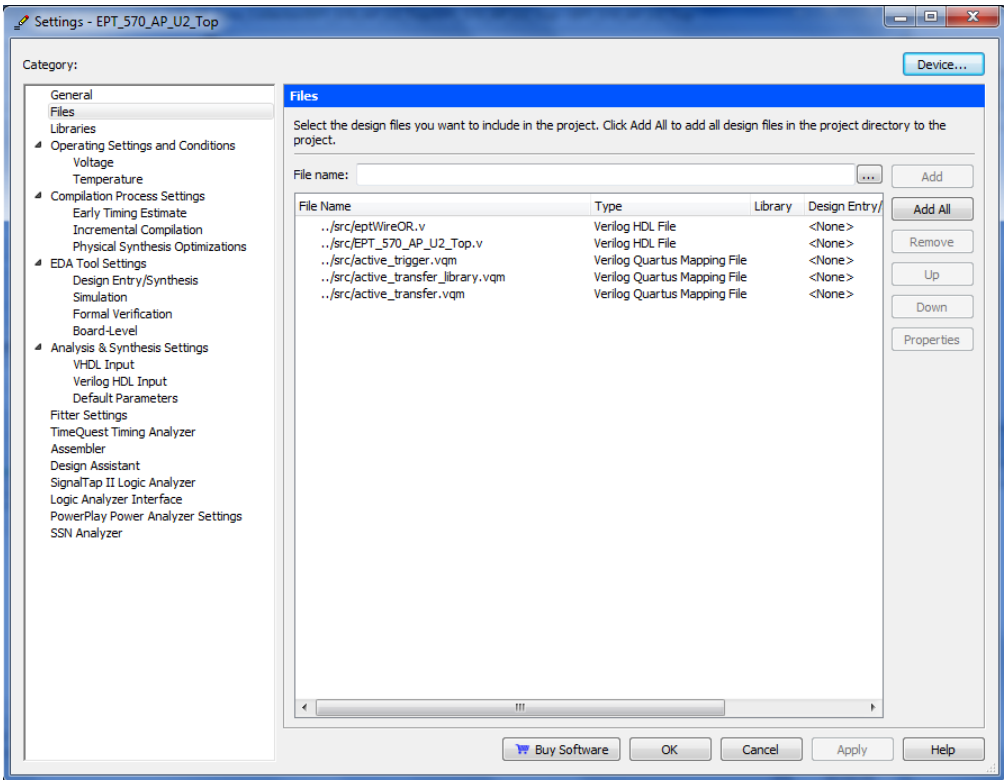


At the Top-Level Entity page, browse to the c:\altera\xxx\quartus\qdesigns\EPT_Data_Collector directory to store your project. Type in a name for your project EPT_4CE6_AF_D1

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. Browse to the \Projects_HDL\EPT_Data_Collector \ EPT_4CE6_AF_ Top folder of the EPT FPGA Development System DVD. Copy the files from the \src directory.

- Active_transfer.vqm
- Active_trigger.vqm
- Active_transfer_library.vqm
- eptWireOr.v
- EPT_4CE6_AF_Top.v

Add the files:



Continue following the instructions by adding a device and finishing the project instantiation. Then, add the Pins.

- Under Assignments, Select Import Assignments.

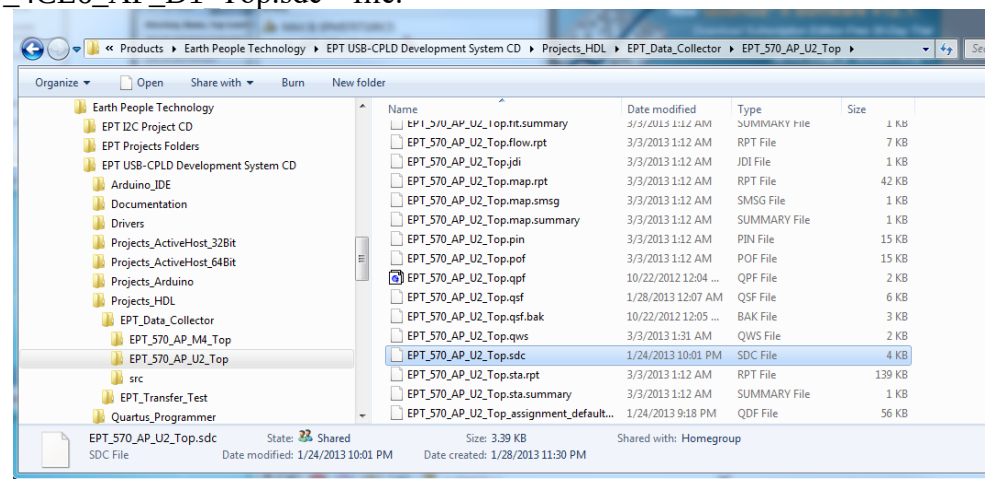
- At the Import Assignment dialog box, browse to the \Projects_HDL\EPT_Data_Collector \ EPT_4CE6_AF_D1_Top folder of the EPT FPGA Development System DVD. Select the Quartus Specification file, " EPT_4CE6_AF_D1_Top.qsf" .

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

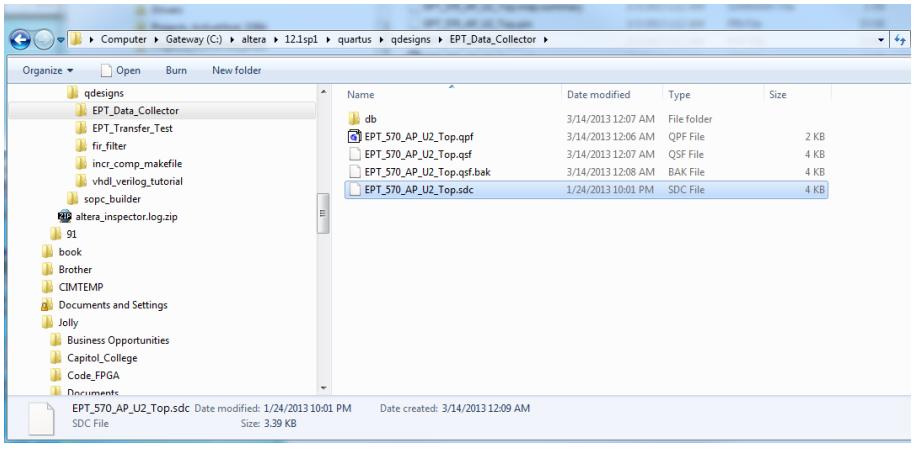
Next, we need to add the Synopsys Design Constraint file. This file contains timing constraints which forces the built in tool called TimeQuest Timing Analyzer to analyze the path of the synthesized HDL code with setup and hold times of the internal registers. It takes note of any path that may be too long to appropriately meet the timing qualifications. For more information on TimeQuest Timing Analyzer, see

http://www.altera.com/literature/hb/qts/qts_qii53018.pdf?GSA_pos=1&WT.oss_r=1&WT.oss=TimeQuest Timing Analyzer

Browse to the \Projects_HDL\EPT_Data_Collector \EPT_EPT-4CE6-AF-D1_TOP folder of the EPT FPGA Development System DVD. Select the "EPT_4CE6_AF_D1_Top.sdc" file.

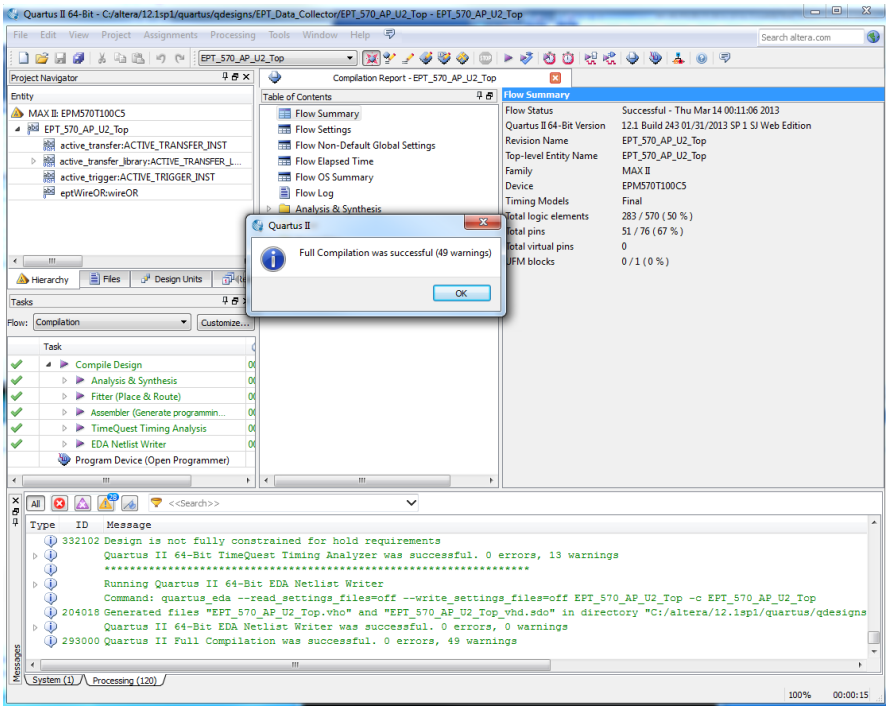


Copy the file and browse to c:\altera\xxx\quartus\qdesigns\EPT_Data_Collector directory. Paste the file.



and select the Start Compilation button.

This will cause the compile and synthesization process. After successful completion, the screen should look like the following:

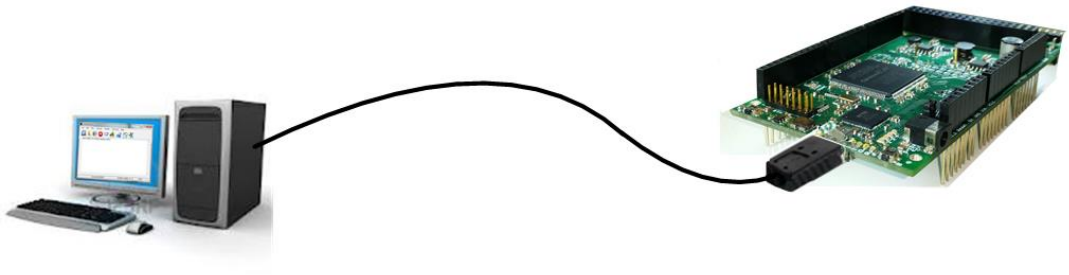


If the synthesis fails, you will see the failure message in the message window. Note that in addition to fatal errors, the compile process can produce “warnings” which do not necessarily prevent execution of the code but which should be corrected eventually.

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 FPGA.

2.1.10 FPGA: Program the DPL Flash

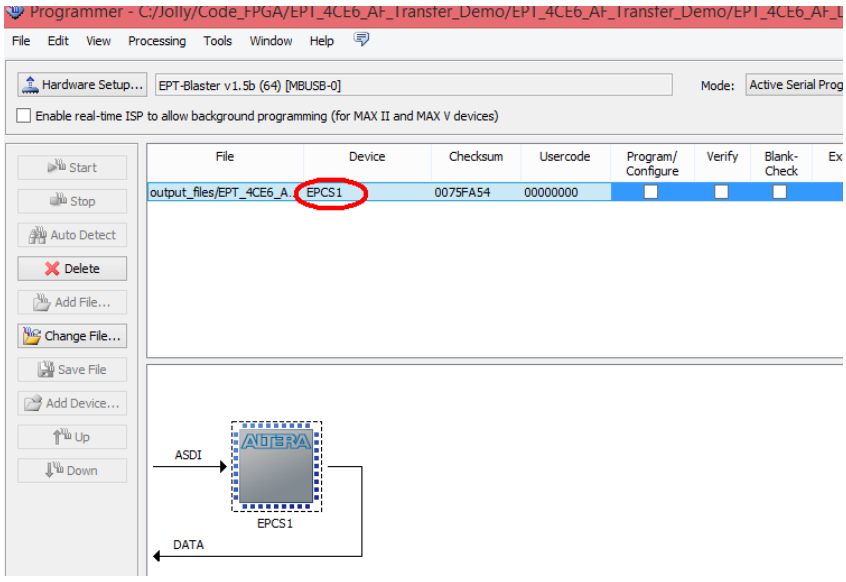
The final step is programming the “*.pof” file into the FPGA. Follow the section: “Programming the FPGA”.



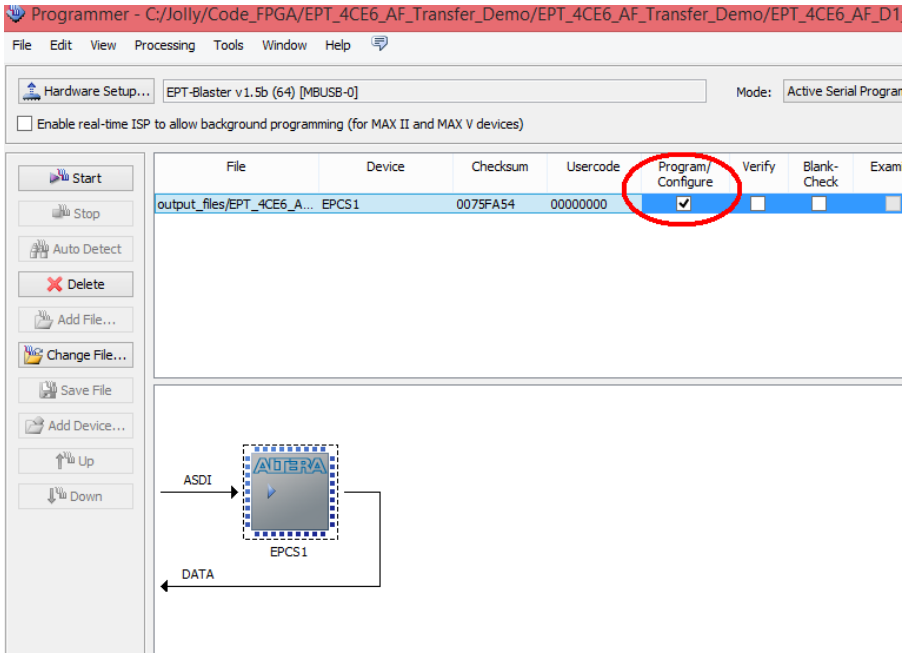
- Connect the EPT-4CE6-AF-D1 to the PC,
- Open up Quartus II,
- Open the programmer tool
- In the upper left corner of the Programmer Tool, there is a button labeled “Hardware Setup”. Verify that EPT-Blaster v1.5b” has been selected. If not, go to the section JTAG DLL Insert to Quartus II and follow the directions.
- Check the box under Program/Configure
- Click the Start button.

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

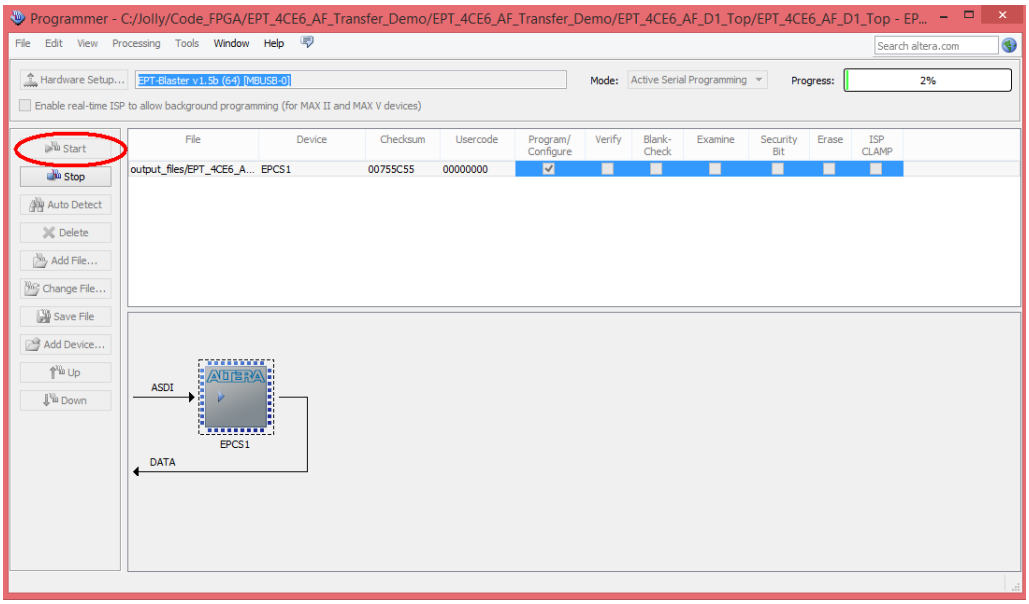
Select the EPCS1 under “Device”.



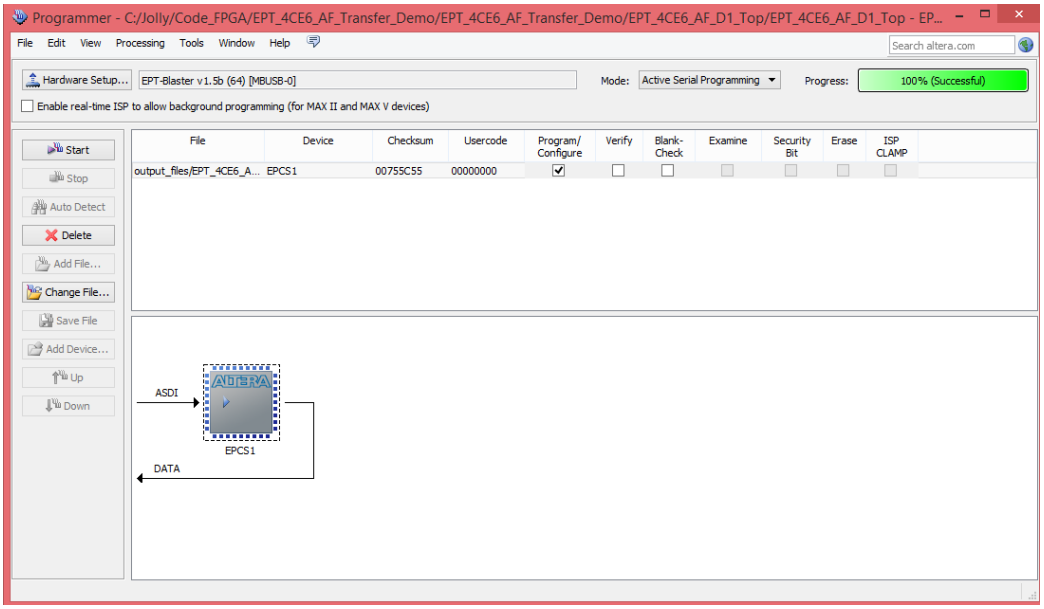
Next, selet the checkbox under the “Program/Configure” of the Programmer Tool.



Click on the Start button to to start programming the FPGA. The Progress bar will indicate the progress of programming.



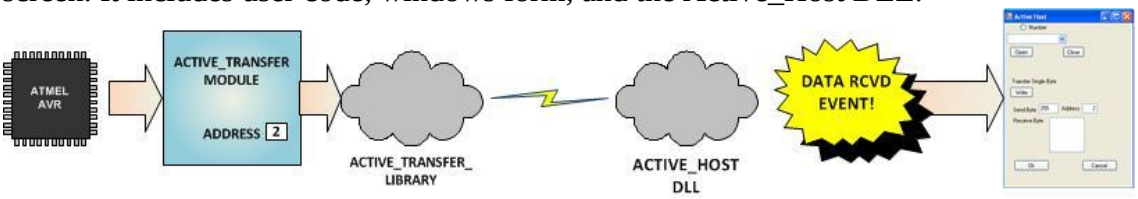
When the programming is complete, the Progress bar will indicate success.



At this point, the EPT-4CE6-AF-D1 is programmed and ready for use.

2.1.11 PC: Design the Project

The final piece of the Data Collection Sampler is the PC application. This application will fetch the data from the FPGA of the EPT-4CE6-AF-D1 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 FPGA 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 FPGA. 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-4CE6-AF-D1 Device.
- Open EPT-4CE6-AF-D1 Device.
- Start the Arduino data collection process.
- Wait for data from EPT-4CE6-AF-D1.

- Display data from EPT-4CE6-AF-D1 in textbox.

2.1.12 PC: Coding the Project

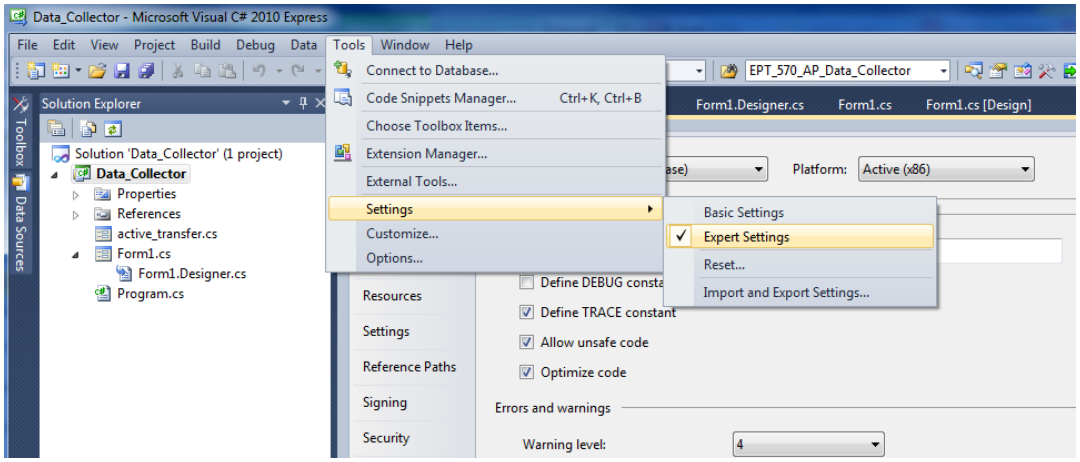
The user code is based on the .NET Framework and written in C#. The language is great for beginners as it is a subset of the C++ language. It has the look and feel of the familiar C language but adds the ease of use of classes, inheritance and method overloading. C# is an event based language which changes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

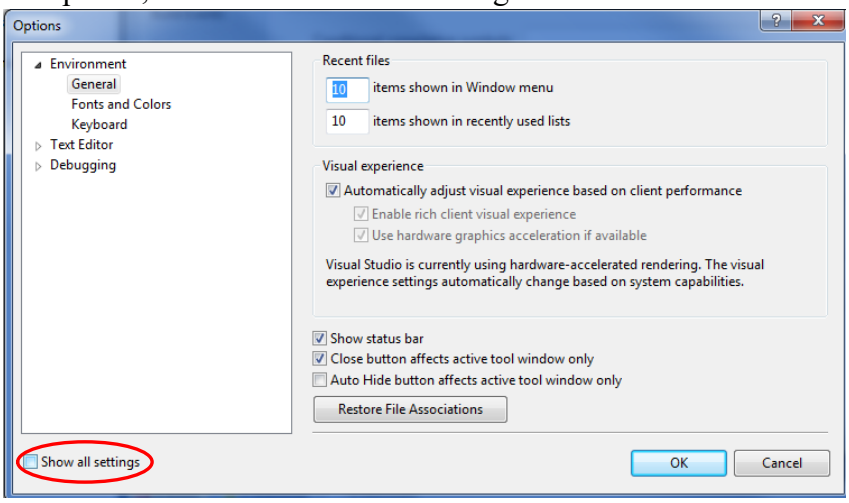
```
using System;
using System.Drawing;
using System.Collections;
using System.Windows.Forms;
using System.Data;
using System.Threading;
using System.Runtime.InteropServices;
using System.Diagnostics;


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.

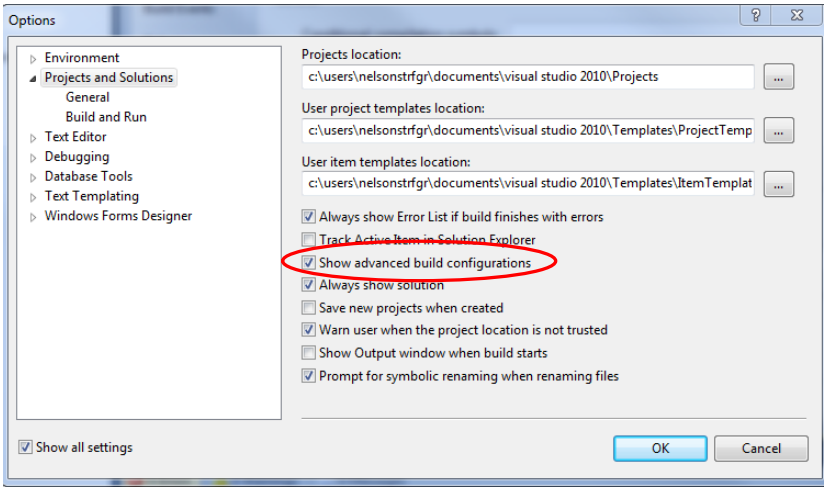
The project environment must be set up correctly in order to produce an application that runs correctly on the target platform. Visual C# Express defaults new projects to 32 bits. If you OS is a 64 bit platform, use the following directions to set up a 64 bit project. First, we need tell C# Express to produce 64 bit code if we are running on a x64 platform. Go to Tools->Settings and select Expert Settings



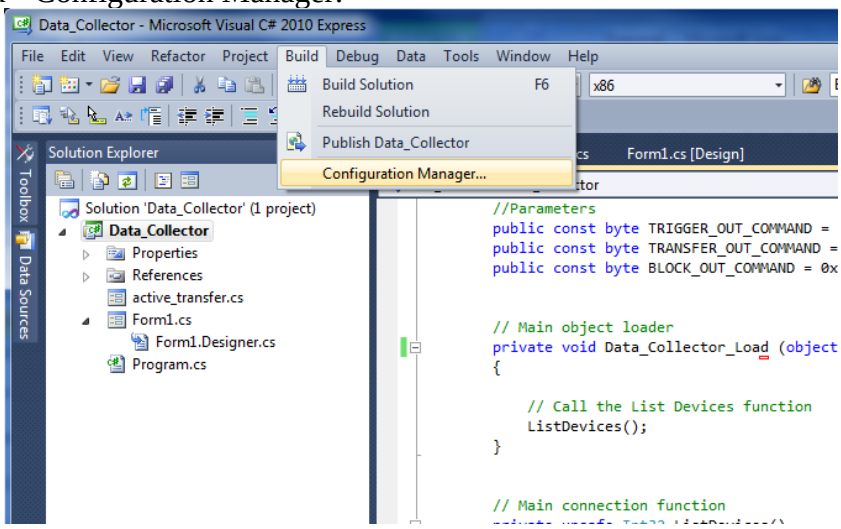
Go to Tools->Options, locate the “Show all settings” check box. Check the box.



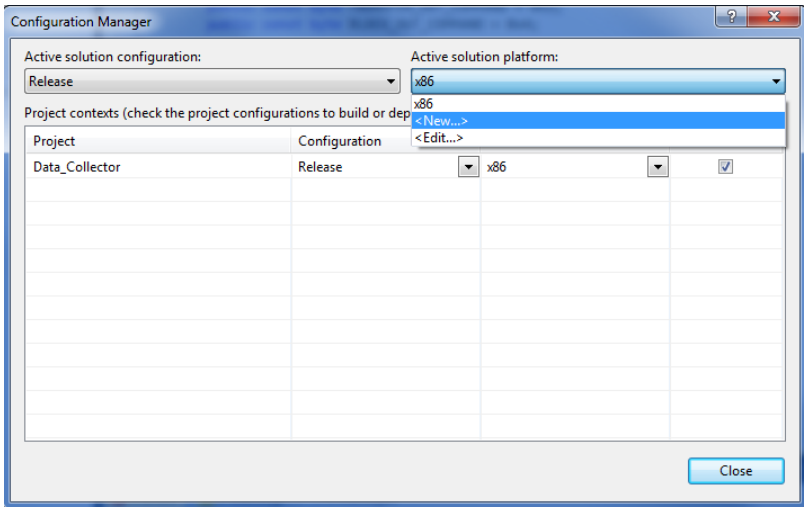
In the window on the left, go to “Projects and Solutions”. Locate the “Show advanced build configurations” check box. Check the box.



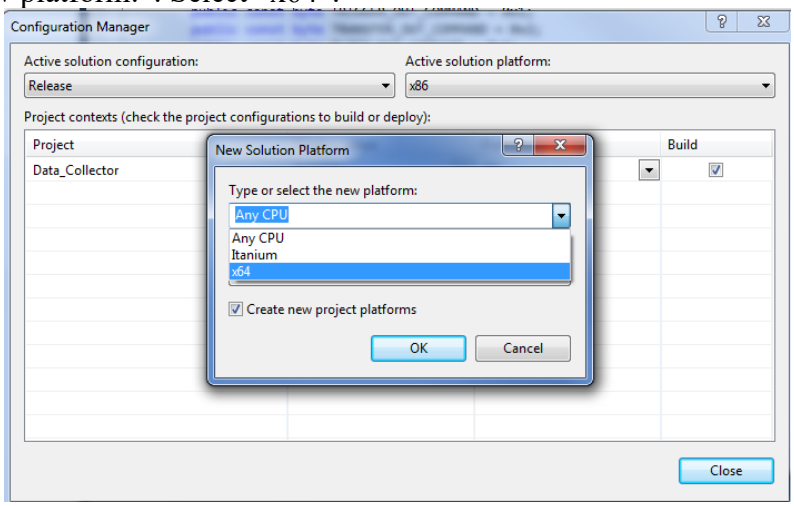
Go to Build->Configuration Manager.



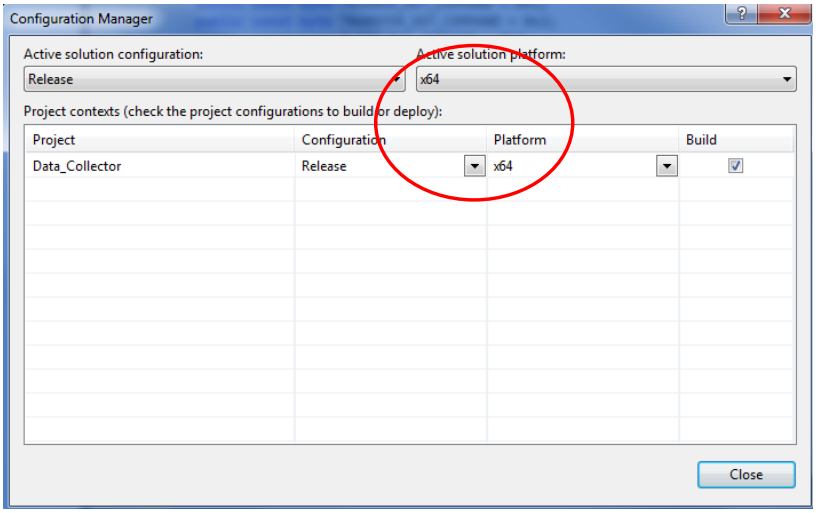
In the Configuration Manager window, locate the “Active solution platform:” label, select “New” from the drop down box.



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

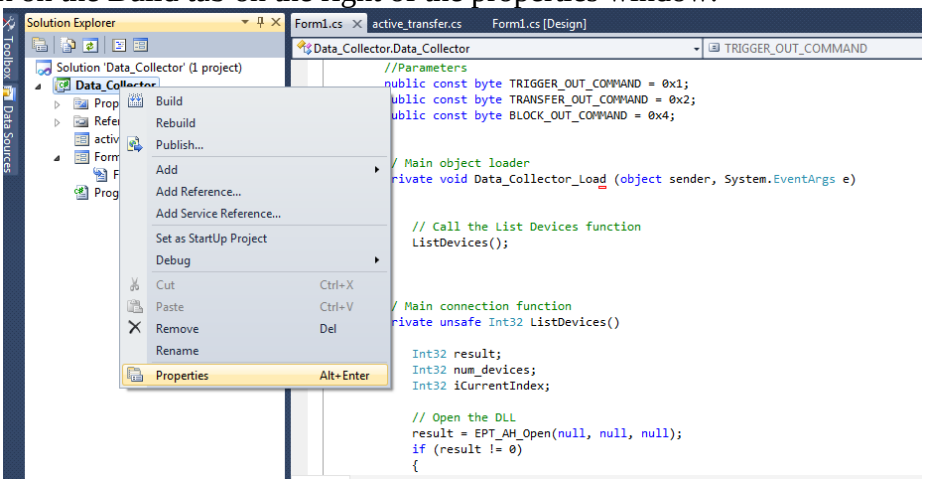


Click the Ok button. Verify that the “Active Solution Platform” and the “Platform” tab are both showing “x64”.

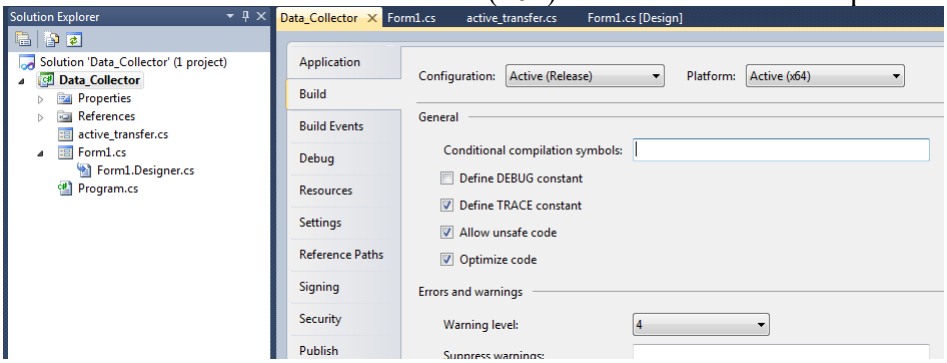


Click Close.

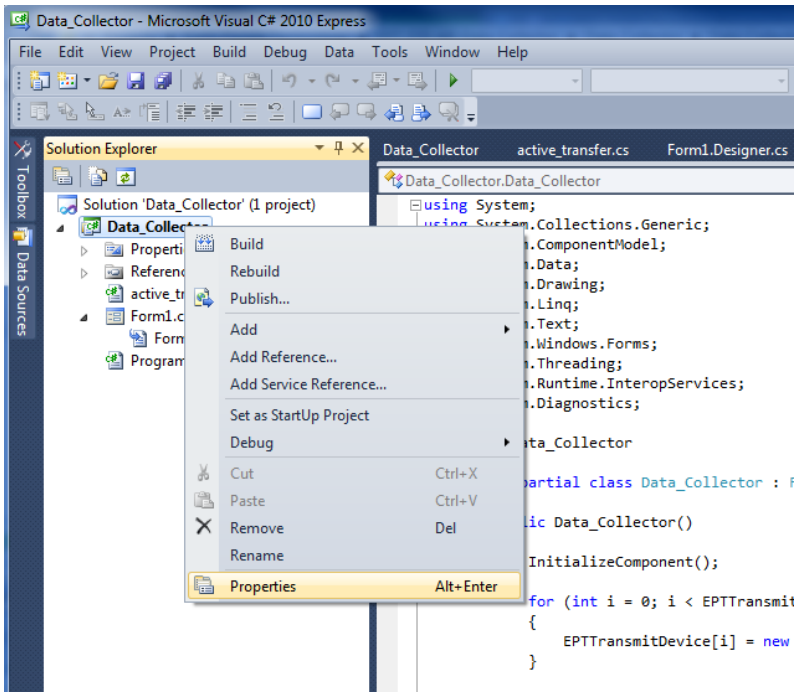
Then, using the Solution Explorer, you can right click on the project, select Properties and click on the Build tab on the right of the properties window.



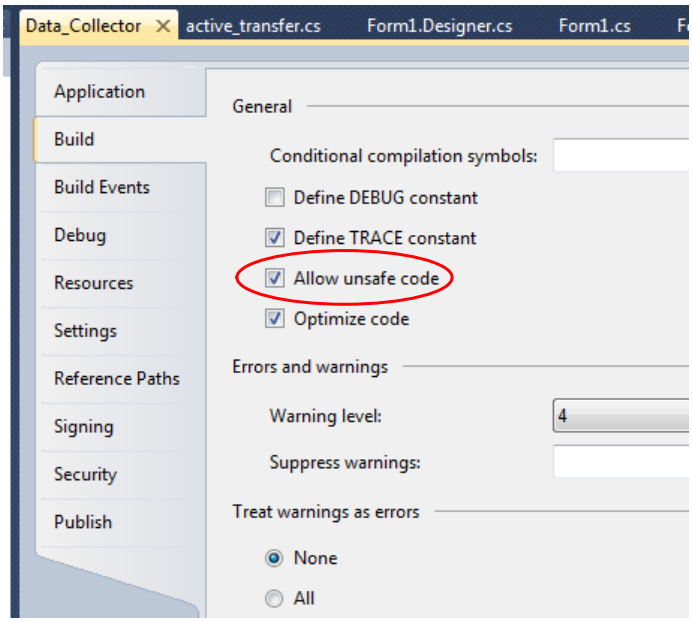
Verify that the “Platform:” label has “Active (x64)” selected from the drop down box.



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();
```

EPT-4CE6-AF-D1

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().

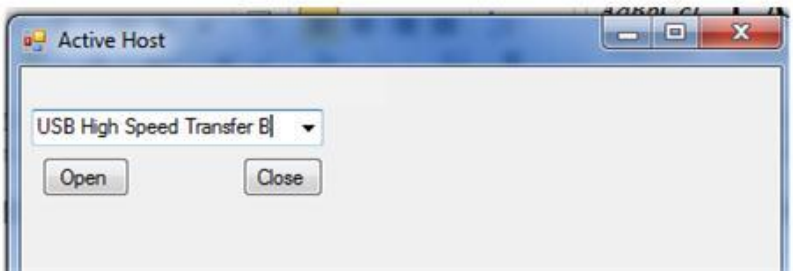
```
// Main object loader
private void Data_Collector_Load (object sender, System.EventArgs e)
   {
   // Call the List Devices function
       ListDevices();
   }
```

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 FPGA code. The btnWriteByte will set the start_stop_cntrl signal in the FPGA to one. This signal starts the Arduino Data Collector sending its random word to the FPGA.

```
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 FPGA.

```
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 FPGA using the C_Enable (WRITE_ENABLE) signal. The FPGA 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().

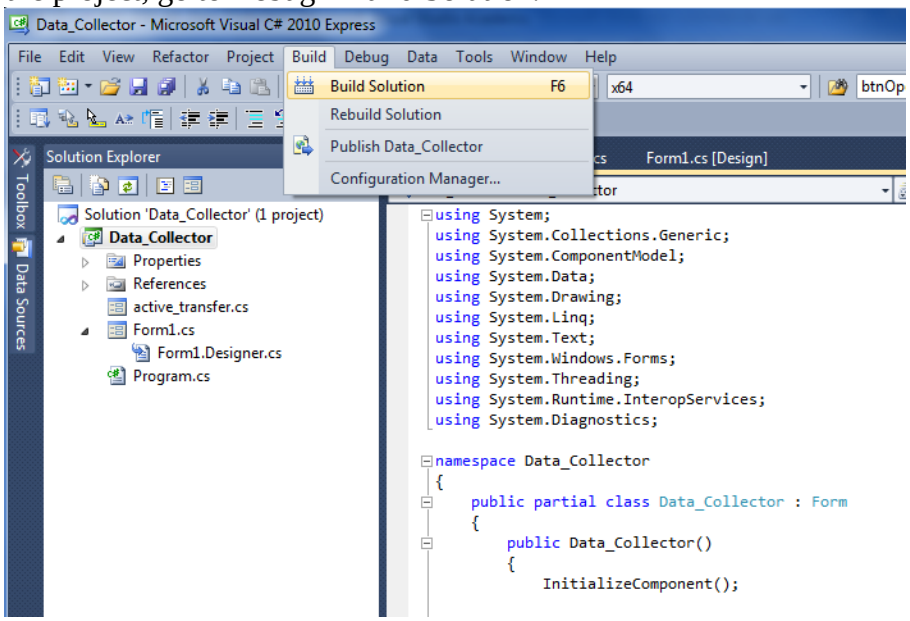
2.1.13 PC: Compiling the Active Host Application

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. Follow section: Assembling, Building, and Executing a .NET Project on the PC. Browse to the \Projects_ActiveHost_xxBit\EPT_Data_Collector \Data_Collector\ folder of the EPT FPGA Development System DVD. Copy the following files into the project.

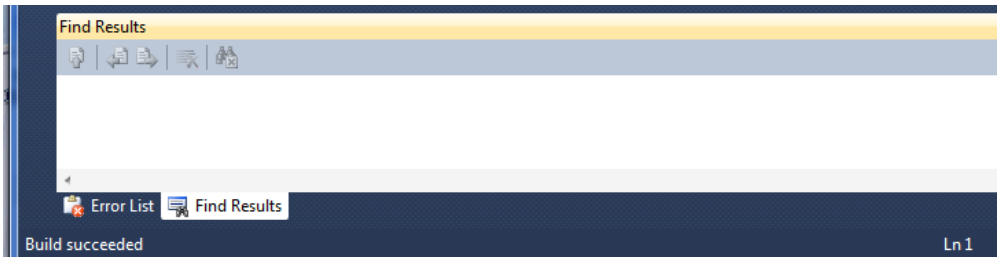
- Active_transfer_xxx.cs
- Data_Collector.csproj

- Data_Collector.csproj.user
- Form1.cs
- Form1.Designer.cs
- Program.cs

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.

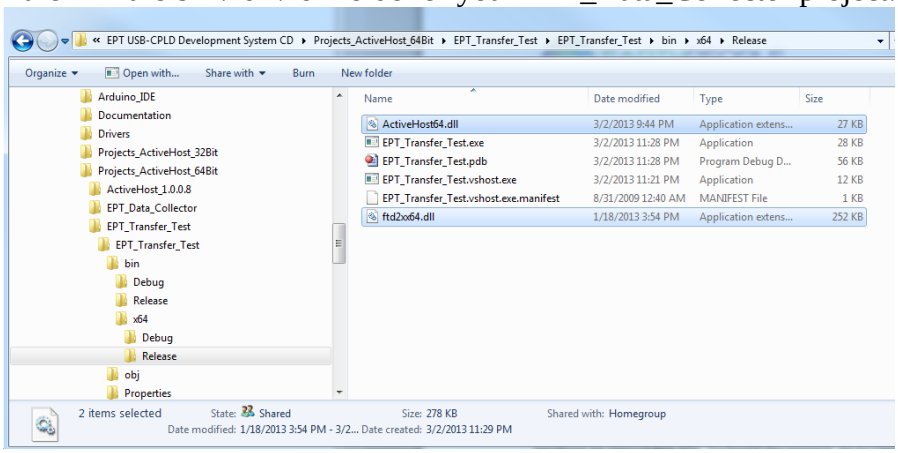
2.1.14 Adding the DLL’s to the Project

Locate the EPT FPGA Development System DVD installed on your PC. Browse to the Projects_ActiveHost folder (choose either the 32 bit or 64 bit version, depending on whether your OS is 32 or 64 bit). Open the Bin folder, copy the following files:

- ActiveHostXX.dll

- ftd2xxXX.dll

and install them in the bin\x64\x64 folder of your EPT_Data_Collector project.



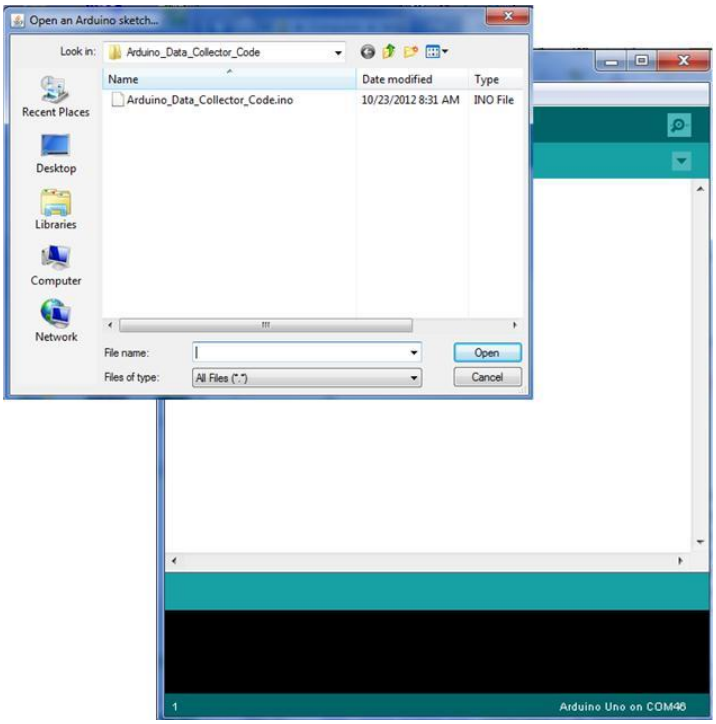
Save the project.

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.

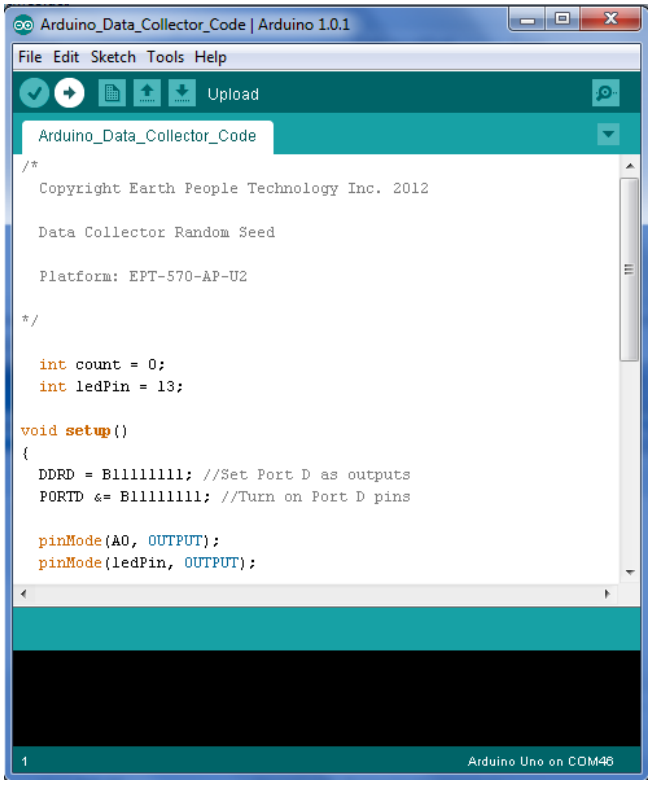
2.1.15 Connecting the Project Together

Now we will connect the Arduino, EPT-4CE6-AF-D1, 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-4CE6-AF-D1.

Next, open the Arduino IDE and select File->Open and select your sketch created earlier, Arduino_Data_Collector_Code_U2.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.

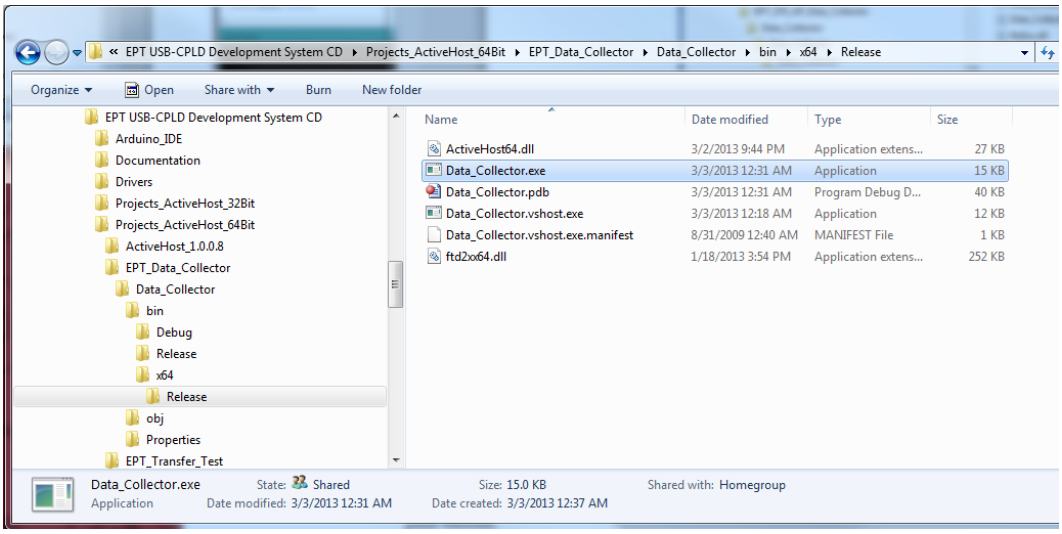


The Arduino IDE will compile the project, then transmit the machine level code into the ATMega328 SRAM to start the program. When this is complete, the Yellow L LED will blink about once per second.

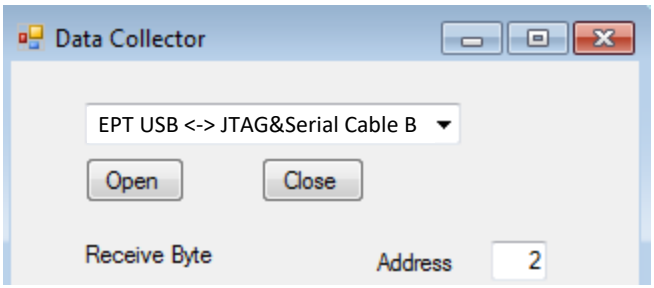
If this LED is blinking at the rate of once per second, the Arduino and the Data Collector project are ready for the EPT 570-AP-U2 code.

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

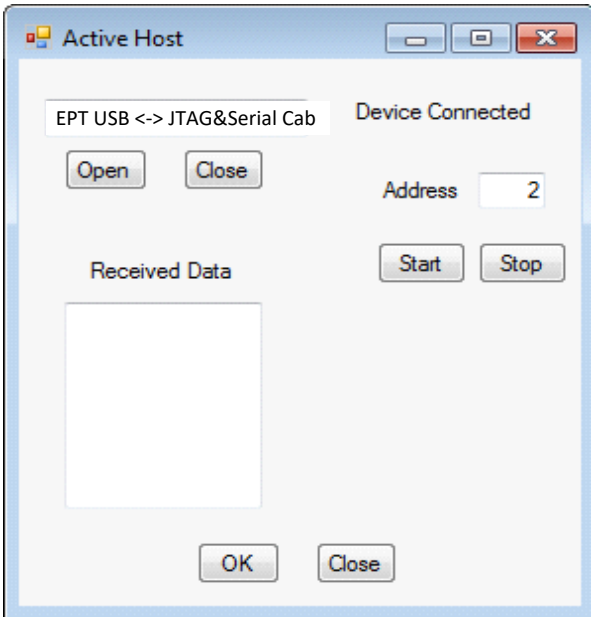
Open the EPT Data Collector on the PC by browsing to the Data Collector project folder. Locate the executable in the \bin\x64\Release folder.



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.

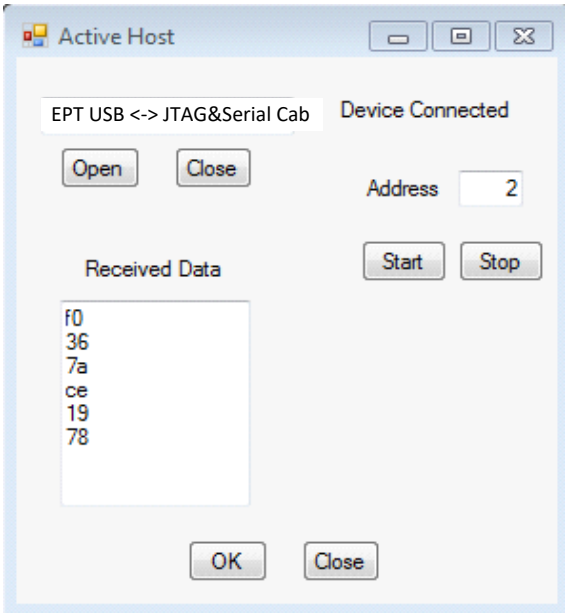


Select the EPT 570-AP device and click the Open button. If the Active Host application connects to the device, a label will indicate “Device Connected”. Next, select the address of the Active Transfer module in the FPGA. In our case it is “2”.



2.1.16 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.

APPENDIX I

List of Abbreviations and Acronyms

EPT	Earth People Technology
FIFO	First In – First Out
FTDI	Future Technology Device International
HSP	Hyper Serial Port
I2C	Inter-Integrated Circuit
JTAG	Joint Test Action Group
PC	Personal Computer
FPGA	Complex Programmable Logic Device
USB	Universal Serial Bus

APPENDIX II

Details of the Altera EPM570 FPGA