ELEX 7660

Thermicon

Prepared by:	Prep Date
Chris Barreiro Stewart	13-Apr-18
Jing Huang	13-Apr-18

Table of Contents

Table of Figures

1 Introduction

1.1 Motivation

Modularity is a hot topic in system design. As component configurations and communication protocols are normalized, modern electronics have become increasingly modular and compatible with each other. At the on-set of this project, we wanted to explore modular design by building a simple composite system using FPGAs that includes a modular component.

An obvious field that might benefit from modularity is data acquisition. A "read-out" source becomes increasingly versatile as more sensors are made compatible. The Thermicon project showcases this by interchangeably gathering various environmental data from a wireless source. Not only will this project exhibit the end-user convenience conferred through modular design, it will also provide us with insight into wireless digital communication and protocols in the context of FPGA programming.

1.2 Inspiration and other projects

A simple fact. Everything is better with Bluetooth. The past 20 years Bluetooth has literally blown up, with Bluetooth connectivity driving innovation and creating new markets. As soon to be engineers, we decided to explore the world of Bluetooth, which marked the start of the Thermicon development. Our completely original idea further progressed and took shape after research led us to bomb disposal robots. Robots go where humans fear to tread, thus we have designed a Bluetooth controlled device that can be sent to remote locations to collect and relay environmental data back to its operator.

As research continued we quickly came across many projects that collectively had all the major aspects of our Thermicon idea. The first project that we found with a controller and vehicle combination consisted of only basic directional controls, it was a conventional RC car built entirely on an FPGA [1]. Another project that posed very useful for us was a Bluetooth car powered by an FPGA that was controllable by any smart phone [2]. This project used an HC05 Bluetooth module which led to the decision to use an HC 06 module for our Thermicon project. The last project we came across that was useful for our project, was a previous ELEX 7660 project from BCIT. A Game Controller Using an FPGA, which utilized UART communication protocol to transfer data between devices [5]. All these projects were a starting point for the Thermicon development process.

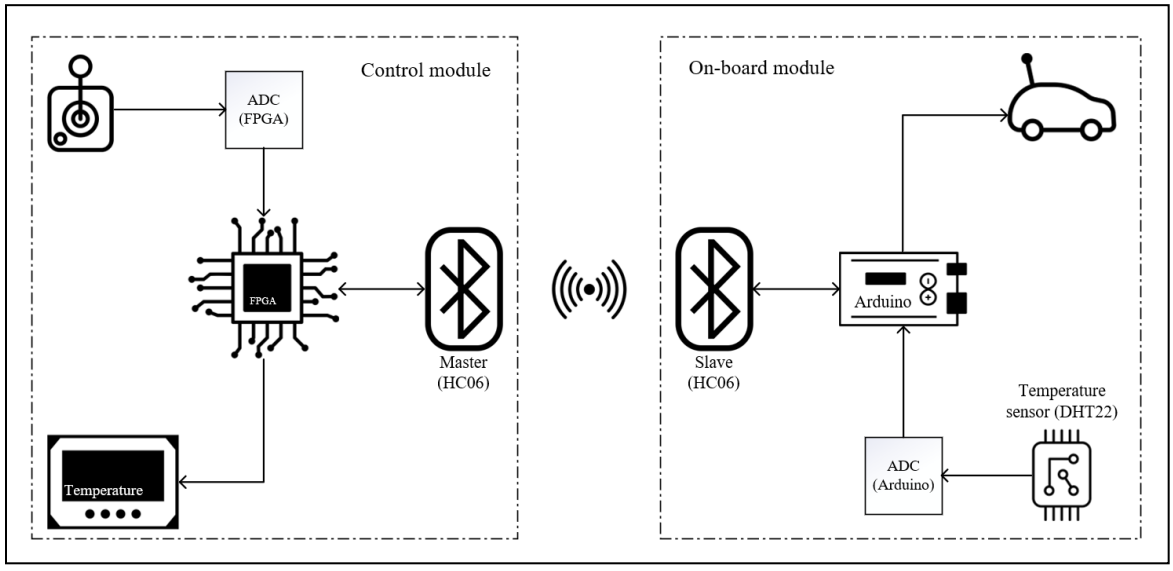


Figure 1 – *System communication block diagram*

2 Design

The two features of the Thermicon are remote-controlled movement and environmental data read-out. Users will be able to navigate the vehicle from a nearby station while sensor readings are relayed from the vehicle. To achieve this, the project will be divided into two corresponding stages.

2.1 UART

Universal asynchronous receiver-transmitters (UARTs) are used extensively in wireless communication. Both transmission and reception utilize serial data communication to respectively deconstruct and reconstruct data between bit-stream and byte form.

In transit, the data is framed as a stream. An example is shown:

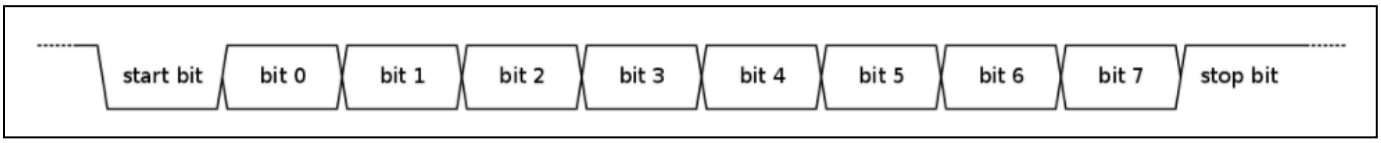


Figure 2 – *A 8-bit UART data segment with no parity*

Typically, the idle-state is held high (voltage at a digital 1), inheriting from legacy telegraphy protocol to ensure line and transmitter functionality. The digital 0 start-bit and digital 1 stop-bit envelops the data bits and any parity bits according to component configuration. The least significant data bit is sent (and received) first, and the receiving UART decodes the piece of data using the start and stop-bits as markers.

The rate at which data is transferred is referred to as the **baud-rate**, and most rate configurations are standardized. At the receiver, a clock runs at a frequency multiple of the baud-rate to periodically sample the data stream. At the detection of a valid start-bit, the subsequent data-bits are clocked into a shift register. The receiving system will access the register after the configured number of bits has been loaded into the register (indicated by the completion of a count of clock ticks corresponding to the UART stream data length).

Wireless transmission using the UART protocol will be facilitated by two HC06 Bluetooth modules. One will be configured as a master and another a slave. Both of which will be connected to the Tx and Rx pins of their respective processors.

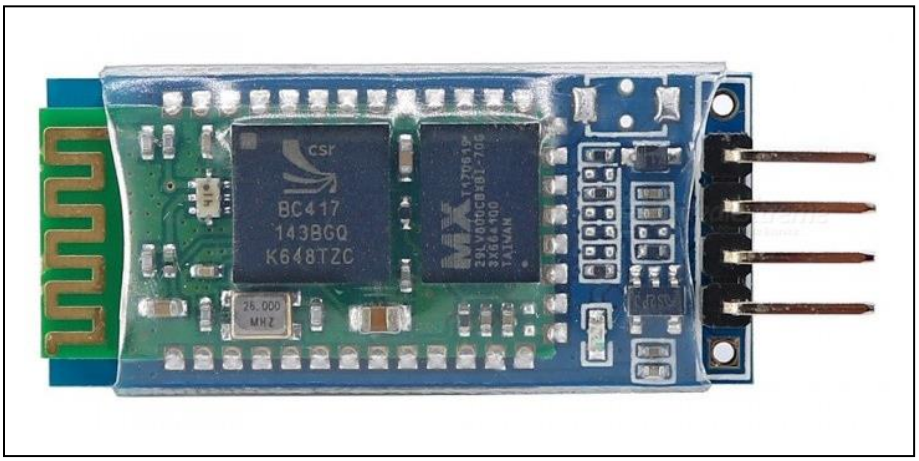
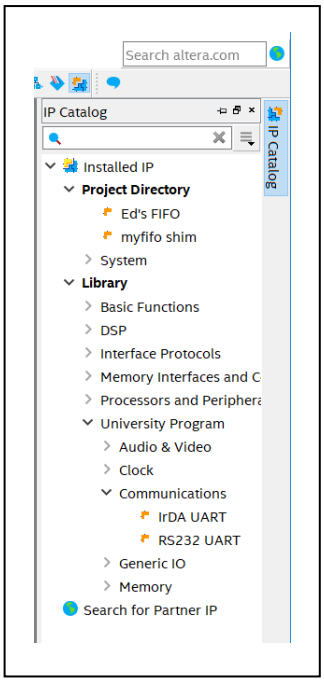


Figure 3 – *A HC06 Bluetooth communicator using UART*

2.1.1 RS232 UART IP Core



Manually implementing UART functionality into a FPGA program involves numerous steps to realize the above-mentioned system, include register initialization, clocking system design and algorithms for data interpretation. Changing between UART timing configurations becomes inconvenient as most changes are made in the UART modules locally.

Fortunately, UART is a widely used protocol and a UART IP Core is included in Quartus's default IP Core libraries. An IP Core is akin to functions found in libraries for higher level languages (such as Boost and Loki for C++). It abstracts the mechanics of the module and allows users to adopt a simple reference-and-use implementation style, saving development time. The RS232 UART IP Core can be accessed in the IP Catalog in the main Quartus window.

The Core allows us to adjust baud-rates, parity bits and data length on the fly. The advantages thereof become apparent in latter sections.

For Thermicon, the Core is used with the following configurations:

- 38400 baud
- No parity
- 1 stop bit

2.2 Control module (FPGA)

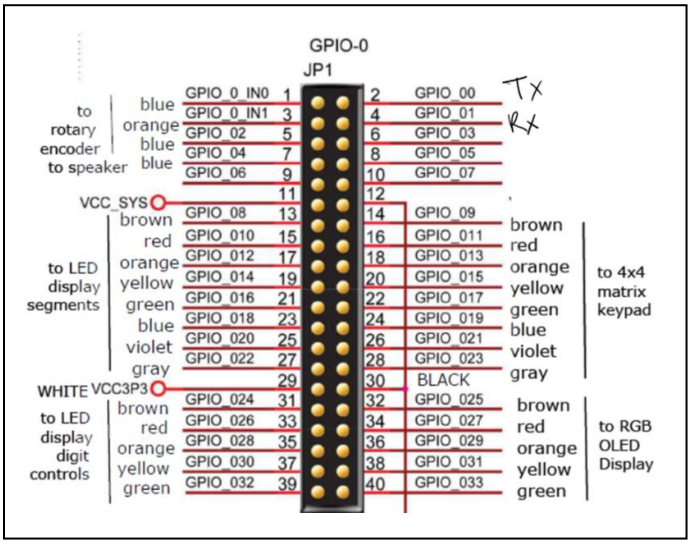


Figure 4 – *Pin-out for Joystick and Tx and Rx pin-assignments*

2.2.1 Directional control

The movement of Thermicon is controlled using a joystick connected to the FPGA control module. The positional data of the joystick is digitized using an ADC on-board the FPGA and its accompanying SystemVerilog module. The data is stored in a register and used as an input to the UART Core's transmission. From there, it is processed by the Core and transmitted to an assigned Tx pin on the development board.

The first 4 bits of each byte of data represents the x position of the joystick, and the last 4 bits the y position. These are categorized into 9 quadrants, with a middle quadrant signifying stationary. When different outer quadrants of positional data are reached, the vehicle will interpret the command as cardinal and intercardinal directions.

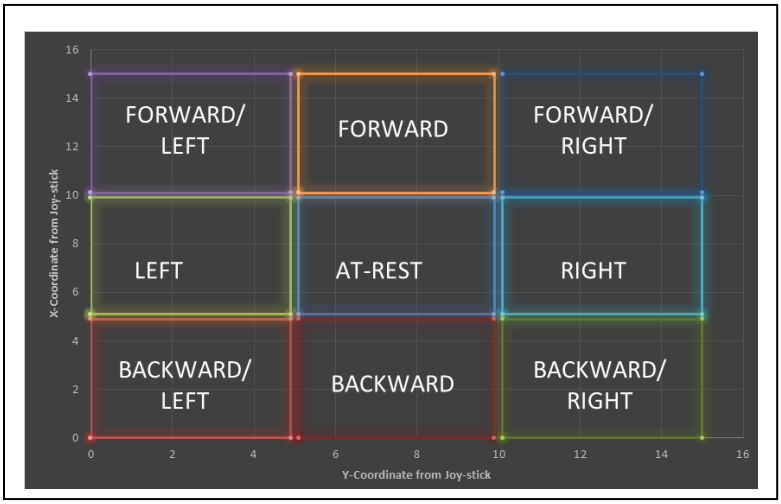


Figure 5 – *Direction/coordination system for Thermicon*

2.2.2 Temperature sensor and display

The DHT22 temperature sensor provides a digital voltage signal corresponding to the temperature in its vicinity. In our design, this signal is digitized through an ADC on-board the Arduino and transmitted through the HC-06 module to the corresponding receiver on-board the FPGA. The FPGA will then process the digital data and provide a temperature value read-out on the connected 7-segment, 4-digit LED.

The LED configuration and usage were explored briefly in Lab 1. It uses a 2-bit decoder cycle through the 4 digits, and an 8-bit decoder to represent the possible numbers and letters. The user-read-out consist of readings in $°C$. To properly display the incoming temperature data, the received data is processed through a binary coded decimal (BCD) converter module such that each digit of the temperature reading is accessible independently.

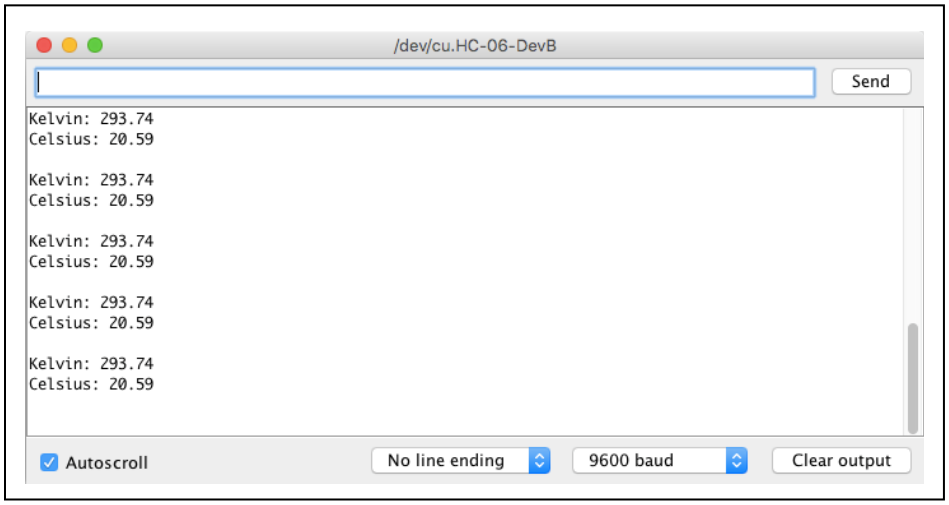


Figure 6 – *Temperature data received via Bluetooth on computer console*

2.3 On-board module (Arduino)

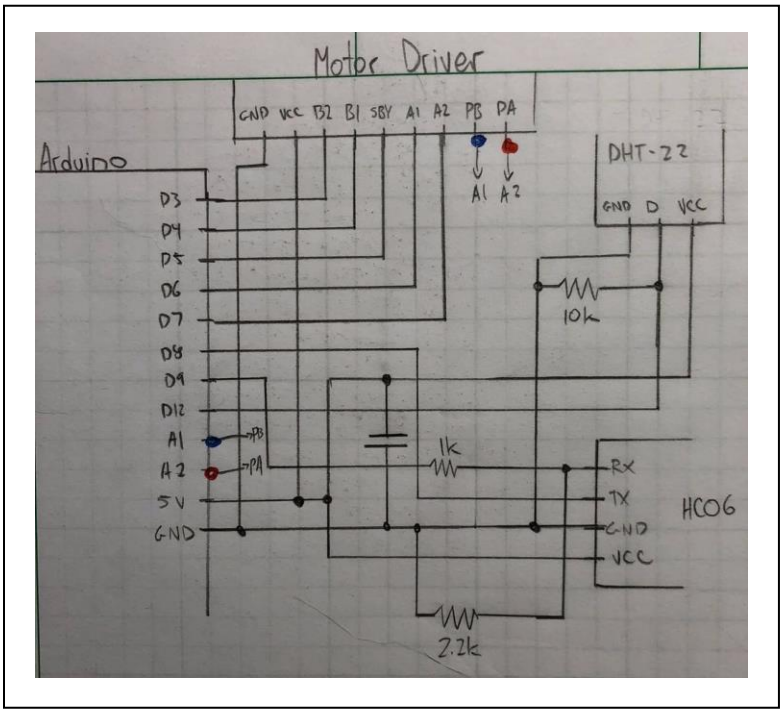


Figure 7 – *Simple Arduino schematic with most pin configurations accomplished in software*

The Arduino provides a very user-friendly programming environment. The initial step involved UART data verification from the FPGA, which turned out to take a lot longer than expected. Using a full-duplex physical wired connection, incoming data was verified using the console available in the Arduino IDE.

Once we guaranteed the correct data could be continuously received, using clever shifting and masking techniques, we were able to parse the data into x and y components so we could relate the joy-stick movements to motor commands. These x and y components of the data represented a coordinate system where both x and y could take on any value from 0 – 15 depending on the current position of the joy-stick. We then broke down the coordinate "map" into different areas which could then be correlated into different motor commands. In Figure 4., you can see we created 9 different rectangles on the map, with an "at-rest" centered joy-stick position relating to the center rectangle.

After driving capabilities had been established, the sensor had to be incorporated so environmental data could be collected. The environmental data was collected at an interval which reflected a combination of battery power consumption as well as the ability to properly represent the environmental condition under experimentation. The data was then converted into a binary value to be send back to the FPGA using the existing UART connection.

Once bi-directional data transfer was perfected, the wired connected was replaced with the wireless HC-06 Bluetooth modules. Timing was further adjusted until data was being reliably send and transmitted continuously on both the FPGA and the Arduino.

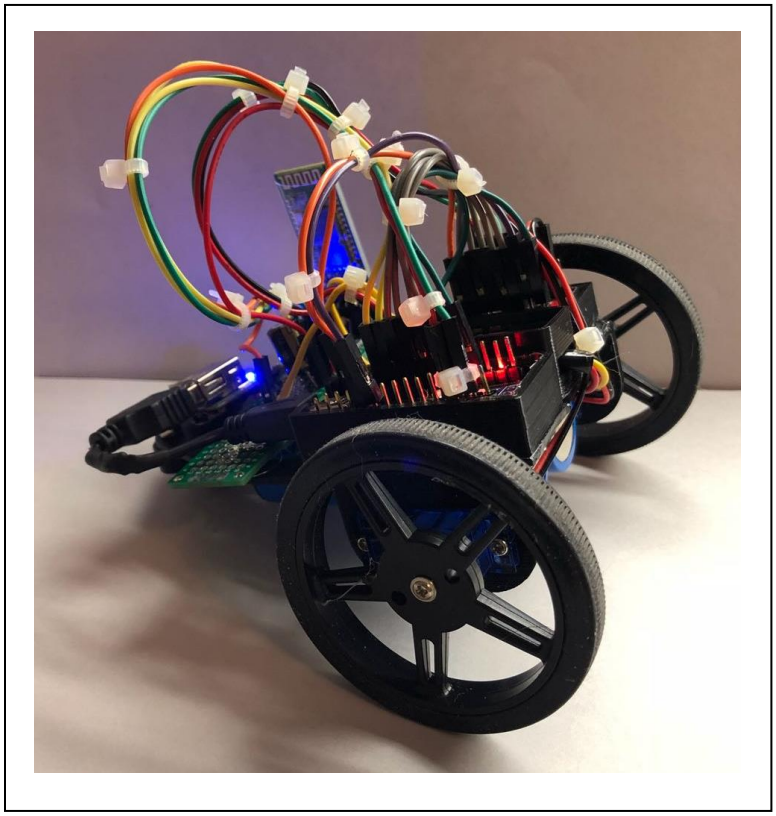


Figure 8 – *Thermicon unit*

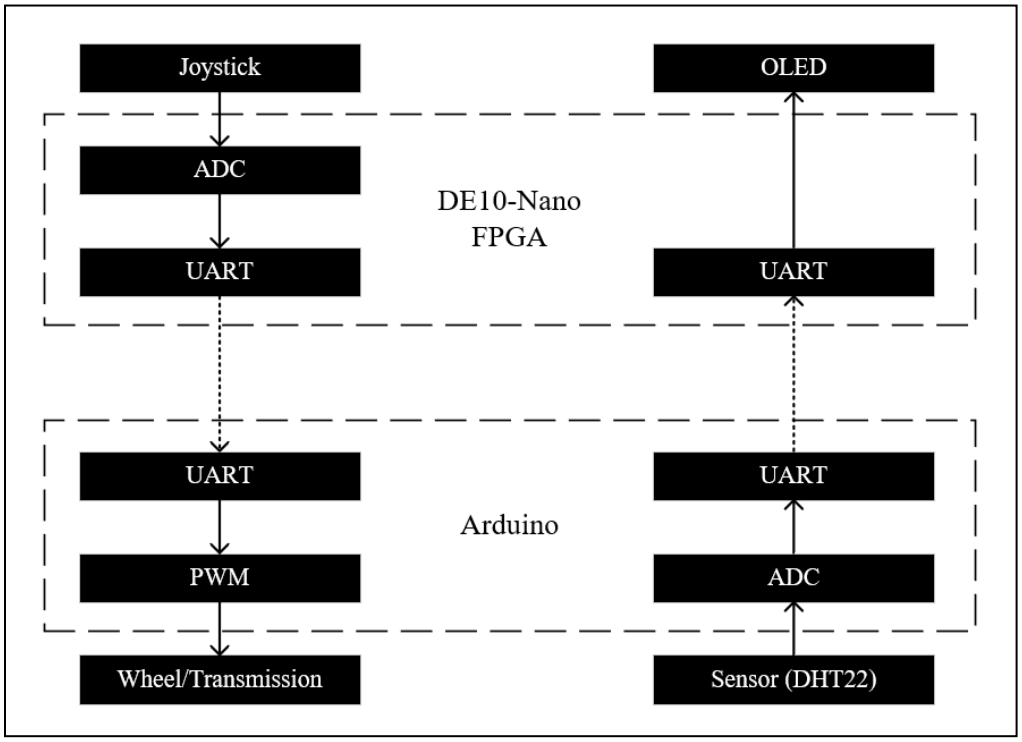


Figure 9 – *Overall data transmission diagram*

3 Potential improvements

A combination of time-constraints and project related problems restricted us from completing all goals originally set out during the project proposal. However, all those goals would definitely be great additional work if this project were to be continued.

Our Thermicon was capable of transmitting either temperature or humidity, but a subsequent version could be capable of simultaneous transfer of both temperature and humidity data between the car and the FPGA controller at fixed time intervals. With more data being transferred, this feature should be accompanied with the FPGA controller being able to save, store, and display at least ten pairs of data points.

The Thermicon was designed for remote mission environmental data collection, thus another important feature would be for the controller to have a "return home" feature, that if pressed the car will trace its steps back to the user. This feature could also be enabled if the connection between the car and controller is lost for some reason.

With any battery-operated device, battery life is crucial, thus using Bluetooth LE (BLE) modules rather than the HC06 modules would definitely improve battery life, but this last addition is by far the most advanced continuation of the project.

4 Conclusion

The remote controlled Thermicon test-unit presented as both an excellent introduction to the world of Bluetooth technology, as well as a fitting educational opportunity to explore the digital design process incorporated in creating a FPGA project from the ground up. Many problems were encountered, especially with the timing aspect for the UART communication protocol over Bluetooth, but we were able to work through and successfully deliver a remote-controlled car that could relay environmental information back to an operator. Project work is always a fun way to learn the material of a class, but with any class project, the educational value delivered is by far the main goal, and working through the encountered problems conjoined with help/suggestions from the instructor turned out to be a very valuable learning experience.

5 References

[1] IVC WIKI. "FPGA remote controlled vehicle." [Online] Available: http://beta.ivc.no/wiki/index.php/FPGA_remote_controlled_vehicle [Feb. 11, 2018].

[2] Pantech Solutions. "Bluetooth based RC car control using Spartan3an FPGA Project Kit." [Online] Available: https://www.pantechsolutions.net/fpga-projects/bluetooth-based-rc-car-control-using-spartan3an-fpga-project-kit [Feb. 11, 2018].

[3] Instructables. (2016) "How to configure HC-06 Bluetooth module as master and slave via command." [Online] Available: http://www.instructables.com/id/How-to-Configure-HC-06-Bluetooth-Module-As-Master-/ [Feb. 12, 2018].

[4] Instructables. (2014) "How to use OLED display Arduino module." [Online] Available: http://www.instructables.com/id/How-to-use-OLED-display-arduino-module/ [Feb. 12, 2018].

[5] N. Heir, A.Ydenberg. (2017). "Game Controller Using an FPGA." BCIT Repository. [Online] Available: https://circuit.bcit.ca/repository/islandora/object/repository%3A350/datastream/PDF/view [Jan. 22, 2018].

[6] NAND LAND. "UART, Serial Port Interface." [Online] Available: https://www.nandland.com/vhdl/modules/module-uart-serial-port-rs232.html [Jan. 22, 2018].

6 SystemVerilog sample code

6.1 thermtop.sv – main module

```
/*

      Module: thermtop

            Main process for Thermicon, including data transceiving, ADC and display
            UART Tx and Rx pins mapped to pin 10 and 11 respectively on GPIO0 bar
            UART data_rx and data_rx exported

      Inputs:

            *

      Outputs:

            *

 */

module thermtop    (     input  logic CLOCK_50,
                         input  logic [1:0] KEY,

      // ADC SPI interface
      output logic ADC_CS_N,        // ssn
      output logic ADC_SADDR,       // mosi
      output logic ADC_SCLK,        // sclk
      input  logic ADC_SDAT,        // miso

      output logic [12:0] DRAM_ADDR,   // dram.addr
      output logic [1:0]  DRAM_BA,     //  .ba
      output logic        DRAM_CAS_N,  //     .cas_n
      output logic        DRAM_CKE,    //     .cke
      output logic        DRAM_CS_N,   //     .cs_n
      inout  logic [15:0] DRAM_DQ,     //     .dq
      output logic [1:0]  DRAM_DQM,    //     .dqm
      output logic        DRAM_RAS_N,  //     .ras_n
      output logic        DRAM_WE_N,   //     .we_n
      output logic DRAM_CLK,

      // UART and external interface
      output logic [7:0]      data_tx,
      output logic            UART_TX,
      input logic             UART_RX,

      // LED display
      output logic [7:0] leds,      // 7-seg LED cathodes
      output logic [3:0] ct         // digit enable
      );

      // registers used in temperature display
      logic [1:0] digit;
      logic [3:0] idnum, tens, ones;
      logic       clkled;
      logic [7:0] data_rx;

      // LED digit decoder
      decode2 decode2_0 (     .digit,
                              .ct
```

```
                        );

// 7-segment output decoder
decode7 decode7_0 (     .num(idnum),
                        .leds
                        );

// temperature displayer
temp temp_0 (       .digit,
                    .tens,
                    .ones,
                    .idnum
                    );

// converts received data to BCD
bcd bcd_0 (         .data_rx,
                    .tens,
                    .ones
                    );

// LED clock
ledclk ledclk_0 ( CLOCK_50,
                    clkled
                    );

// cycles through digits to display on LED
always_ff @(posedge clkled)
      digit <= digit + 1'b1 ;

// assigns data to be transmitted to data registers
assign data_tx = { data[11:8], data[27:24] } ;


// instantiates therm instance and begins data processing
therm u0 (
      .clock_50_clk   (clk),      //  clock_50.clk
      .reset_reset    (reset_n), //  reset.reset

      .sdram_clk_clk (DRAM_CLK),           // sdram_clk.clk
      .sdram_addr     (DRAM_ADDR),         //     sdram.addr
      .sdram_ba       (DRAM_BA),           //           .ba
      .sdram_cas_n    (DRAM_CAS_N),        //           .cas_n
      .sdram_cke      (DRAM_CKE),          //           .cke
      .sdram_cs_n     (DRAM_CS_N),         //           .cs_n
      .sdram_dq       (DRAM_DQ),           //           .dq
      .sdram_dqm      (DRAM_DQM),          //           .dqm
      .sdram_ras_n    (DRAM_RAS_N),        //           .ras_n
      .sdram_we_n     (DRAM_WE_N),         //           .we_n

      // loads Tx and Rx data into their respective registers
      .data_rx_ready (1),
      .data_rx_valid (1),
      .data_rx_data (data_rx),

      .data_tx_ready (1),
      .data_tx_valid (1),
      .data_tx_data (data_tx),

      // directs data to the assigned inputs and outputs
      .data_external_TXD (UART_TX),
      .data_external_RXD (UART_RX)
```

```
	);


	// instantiates SPI master for the ADC, outputs to data

	logic ready ;
	logic valid ;
	logic [31:0] data ;
	logic reset_n, clk ;

	assign clk = CLOCK_50 ;
	assign reset_n = KEY[0] ;

	adcspi a0   (	.sclk(ADC_SCLK),
			.mosi(ADC_SADDR),
			.ssn(ADC_CS_N),
			.miso(ADC_SDAT),

			.ready(ready),
			.valid(valid),
			.data(data),

			.clk(clk),
			.reset(~reset_n)
			) ;
endmodule

/*

	Module: adcspi

		SPI to ADC interface. Used to convert analog data from assigned pins
		to digital data. This is recycled from Lab 5 code.

	Inputs:

		miso  - master in slave out data pin
		ready - data ready
		clk   - 50 MHz clock
		reset - ADC reset

	Outputs:

		mosi  - master in slave out data pin
		valid - data valid
		[31:0] data - output ADC data
		sclk  - clk divided by 16

	Parameters:

		MISO - actual MISO register for input

	Registers:

		[31:0] sr   - shift register for data storage
		rising- rising edge
		falling - falling edge
		done  - flag to allow data shifting

 */
```

```
module adcspi (
	output logic sclk, mosi, ssn,
	input logic miso,

	input logic ready,
	output logic valid,
	output logic [31:0] data,

	input logic clk, reset ) ;

	parameter MISO = {5'b00001,27'b0} ;

	// clock/bit counter
	struct packed {
		logic wordcnt ;
		logic [3:0] bitcnt ;
		logic sclk ;
		logic [3:0] clkcnt ;
		} cnt, cnt_next ;

	logic [31:0] sr ;

	logic rising, falling, done ;

	assign sclk = cnt.sclk ;

	// done all bits
	assign done = cnt ==? '{'1,'1,'1,'1} ;

	// clock/bit counter
	assign cnt_next = ( reset || done ) ? '0 : cnt+1'b1 ;
	always@(posedge clk)
		cnt <= cnt_next ;

	assign rising  =  cnt_next.sclk && ~cnt.sclk ;
	assign falling = ~cnt_next.sclk &&  cnt.sclk ;

	always@(posedge clk) begin
		if ( falling )                  // shift mosi out
			mosi <= sr[31] ;

		if ( rising )                   // shift miso in
			sr <= {sr[30:0],miso} ;

		if ( done ) begin
			data <= sr ;                // copy to parallel out
			sr <= MISO ;                // channel select serial out
			mosi <= MISO[31] ;
			valid <= '1 ;               // data ready
		end

		if ( ready && valid )           // data was read
			valid <= '0 ;
	end

	always@(posedge clk)            // run continously
		ssn <= reset ;
endmodule


// generated LED clock module
```

```
module ledclk (inclk0, c0);

	input	inclk0;
	output	c0;

	wire [0:0] sub_wire2 = 1'h0;
	wire [4:0] sub_wire3;
	wire  sub_wire0 = inclk0;
	wire [1:0] sub_wire1 = {sub_wire2, sub_wire0};
	wire [0:0] sub_wire4 = sub_wire3[0:0];
	wire  c0 = sub_wire4;

	altpll altpll_component ( .inclk (sub_wire1), .clk
	  (sub_wire3), .activeclock (), .areset (1'b0), .clkbad
	  (), .clkena ({6{1'b1}}), .clkloss (), .clkswitch
	  (1'b0), .configupdate (1'b0), .enable0 (), .enable1 (),
	  .extclk (), .extclkena ({4{1'b1}}), .fbin (1'b1),
	  .fbmimicbidir (), .fbout (), .fref (), .icdrclk (),
	  .locked (), .pfdena (1'b1), .phasecounterselect
	  ({4{1'b1}}), .phasedone (), .phasestep (1'b1),
	  .phaseupdown (1'b1), .pllena (1'b1), .scanaclr (1'b0),
	  .scanclk (1'b0), .scanclkena (1'b1), .scandata (1'b0),
	  .scandataout (), .scandone (), .scanread (1'b0),
	  .scanwrite (1'b0), .sclkout0 (), .sclkout1 (),
	  .vcooverrange (), .vcounderrange ());

	defparam
		altpll_component.bandwidth_type = "AUTO",
		altpll_component.clk0_divide_by = 25000,
		altpll_component.clk0_duty_cycle = 50,
		altpll_component.clk0_multiply_by = 1,
		altpll_component.clk0_phase_shift = "0",
		altpll_component.compensate_clock = "CLK0",
		altpll_component.inclk0_input_frequency = 20000,
		altpll_component.intended_device_family = "Cyclone IV E",
		altpll_component.lpm_hint = "CBX_MODULE_PREFIX=lab1clk",
		altpll_component.lpm_type = "altpll",
		altpll_component.operation_mode = "NORMAL",
		altpll_component.pll_type = "AUTO",
		altpll_component.port_activeclock = "PORT_UNUSED",
		altpll_component.port_areset = "PORT_UNUSED",
		altpll_component.port_clkbad0 = "PORT_UNUSED",
		altpll_component.port_clkbad1 = "PORT_UNUSED",
		altpll_component.port_clkloss = "PORT_UNUSED",
		altpll_component.port_clkswitch = "PORT_UNUSED",
		altpll_component.port_configupdate = "PORT_UNUSED",
		altpll_component.port_fbin = "PORT_UNUSED",
		altpll_component.port_inclk0 = "PORT_USED",
		altpll_component.port_inclk1 = "PORT_UNUSED",
		altpll_component.port_locked = "PORT_UNUSED",
		altpll_component.port_pfdena = "PORT_UNUSED",
		altpll_component.port_phasecounterselect = "PORT_UNUSED",
		altpll_component.port_phasedone = "PORT_UNUSED",
		altpll_component.port_phasestep = "PORT_UNUSED",
		altpll_component.port_phaseupdown = "PORT_UNUSED",
		altpll_component.port_pllena = "PORT_UNUSED",
		altpll_component.port_scanaclr = "PORT_UNUSED",
		altpll_component.port_scanclk = "PORT_UNUSED",
		altpll_component.port_scanclkena = "PORT_UNUSED",
		altpll_component.port_scandata = "PORT_UNUSED",
		altpll_component.port_scandataout = "PORT_UNUSED",
```

```
		altpll_component.port_scandone = "PORT_UNUSED",
		altpll_component.port_scanread = "PORT_UNUSED",
		altpll_component.port_scanwrite = "PORT_UNUSED",
		altpll_component.port_clk0 = "PORT_USED",
		altpll_component.port_clk1 = "PORT_UNUSED",
		altpll_component.port_clk2 = "PORT_UNUSED",
		altpll_component.port_clk3 = "PORT_UNUSED",
		altpll_component.port_clk4 = "PORT_UNUSED",
		altpll_component.port_clk5 = "PORT_UNUSED",
		altpll_component.port_clkena0 = "PORT_UNUSED",
		altpll_component.port_clkena1 = "PORT_UNUSED",
		altpll_component.port_clkena2 = "PORT_UNUSED",
		altpll_component.port_clkena3 = "PORT_UNUSED",
		altpll_component.port_clkena4 = "PORT_UNUSED",
		altpll_component.port_clkena5 = "PORT_UNUSED",
		altpll_component.port_extclk0 = "PORT_UNUSED",
		altpll_component.port_extclk1 = "PORT_UNUSED",
		altpll_component.port_extclk2 = "PORT_UNUSED",
		altpll_component.port_extclk3 = "PORT_UNUSED",
		altpll_component.width_clock = 5;

endmodule
```

6.2 decode2.sv – LED digit selector

```
/*

	Module: decode2

		Basic 2 to 4 decoder

	Inputs:

		[1:0] digit - 2-bit input

	Outputs:

		[4:0] ct - 4-bit output

 */
module decode2 (  input logic [1:0] digit,
                  output logic [3:0] ct   );
	always_comb begin
		unique case (digit)
			0: ct = 4'b0001;
			1: ct = 4'b0010;
			2: ct = 4'b0100;
			default: ct = 4'b1000;  // 'default' used to avoid synthesizing a
flipflop
		endcase
	end
endmodule
```

6.3 decode7.sv – LED 7-segment decoder

```
/*

      Module: decode7

            4 to 7 decoder to translate input num into LED display configuration

      Inputs:

            [3:0] num - 4-bit input

      Outputs:

            [7:0] leds - 7-bit output

 */
module decode7 (  input logic [3:0] num,
                              output logic [7:0] leds );
      always_comb begin
            unique case (num)
                  0: leds = 8'b1100_0000;       // 0
                  1: leds = 8'b1111_1001;       // 1
                  2: leds = 8'b1010_0100;       // 2
                  3: leds = 8'b1011_0000;       // 3
                  4: leds = 8'b1001_1001;       // 4
                  5: leds = 8'b1001_0010;       // 5
                  6: leds = 8'b1000_0010;       // 6
                  7: leds = 8'b1111_1000;       // 7
                  8: leds = 8'b1000_0000;       // 8
                  9: leds = 8'b1001_0000;       // 9
                  10: leds = 8'b1111_1111;      // blank
                  default: leds = 8'b1100_0110; // C
            endcase
      end

endmodule
```

6.4 bcd.sv – binary coded decimal converter

```
/*
      Module: bcd

            A simple binary coded decimal converter. Takes an 8-bit
            binary number and converts it to its tens digit and ones digit

      Inputs:

            [7:0] data_rx - 8-bit binary number input

      Outputs:

            [3:0] tens - 4-bit number output of the tens digit
            [3:0] ones - 4-bit number output of the ones digit

      Registers:

            [19:0] shift -    a shift register for storing the original
                              input data
 */
```

```
module bcd  (     input logic [7:0] data_rx,
                  output logic [3:0] tens, ones
                  );

      // variable for storing data_rx and stuff
      logic [19:0] shift;
      int i;

      always_comb begin

            // clear previous data_rx and store new data_rx in shift register
            shift[15:8] = 0;
            shift[7:0] = data_rx;

            // loop over all bits of data_rx
            for (i=0; i<8; i=i+1) begin
                  if (shift[11:8] >= 5)
                        shift[11:8] = shift[11:8] + 3;

                  if (shift[15:12] >= 5)
                        shift[15:12] = shift[15:12] + 3;

                  // shift entire register left once
                  shift = shift << 1;
            end

            // push decimal representation of data_rx to output
            tens = shift[15:12];
            ones = shift[11:8];
      end
endmodule
```

6.5 temp.sv – temperature display module

```
/*

      Module: temp

            Indicates the value of temperature data to be displayed

      Inputs:

            [1:0] digit - digit indicator
            [3:0] tens  - tens digit data
            [3:0] ones  - ones digit data

      Outputs:

            [4:0] idnum - 7-segment output mapped to numbers and data

 */

module temp (     input logic [1:0] digit,
                  input logic [3:0] tens, ones,
                  output logic [3:0] idnum);
      always_comb begin
            unique case (digit)
                  0: idnum = 11;    // "C"
                  1: idnum = ones;  // ones digit
```

```
                2: idnum = tens;  // tens digit
                default: idnum = 10;    // display idnum code for blank
            endcase
        end
endmodule
```

6.6 therm_rs232_0.sv – RS232 UART IP Core[1]

```
// (C) 2001-2017 Intel Corporation. All rights reserved.
// Your use of Intel Corporation's design tools, logic functions and other
// software and tools, and its AMPP partner logic functions, and any output
// files from any of the foregoing (including device programming or simulation
// files), and any associated documentation or information are expressly subject
// to the terms and conditions of the Intel Program License Subscription
// Agreement, Intel FPGA IP License Agreement, or other applicable
// license agreement, including, without limitation, that your use is for the
// sole purpose of programming logic devices manufactured by Intel and sold by
// Intel or its authorized distributors.  Please refer to the applicable
// agreement for further details.


// THIS FILE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
// THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
// FROM, OUT OF OR IN CONNECTION WITH THIS FILE OR THE USE OR OTHER DEALINGS
// IN THIS FILE.

/****************************************************************************
 *                                                                          *
 * This module reads and writes data to the RS232 connector on Altera's     *
 *  DE-series Development and Education Boards.                             *
 *                                                                          *
 ****************************************************************************/

module therm_rs232_0 (
      // Inputs
      clk,
      reset,

      from_uart_ready,

      to_uart_data,
      to_uart_error,
      to_uart_valid,

      UART_RXD,

      // Bidirectionals

      // Outputs
      from_uart_data,
      from_uart_error,
      from_uart_valid,
```

[1] The UART IP Core file is generated. This is only included to show that it was implemented into our main program. The in deserializer and out serializer are not included in this report.

```
        to_uart_ready,

        UART_TXD
);

/*****************************************************************************
 *                            Parameter Declarations                         *
 *****************************************************************************/

parameter CW                    = 11;       // Baud counter width
parameter BAUD_TICK_COUNT       = 1302;
parameter HALF_BAUD_TICK_COUNT= 651;

parameter TDW                   = 10;       // Total data width
parameter DW                    = 8;        // Data width
parameter ODD_PARITY            = 1'b0;

/*****************************************************************************
 *                             Port Declarations                             *
 *****************************************************************************/
// Inputs
input                               clk;
input                               reset;

input                               from_uart_ready;

input       [(DW-1):0]              to_uart_data;
input                               to_uart_error;
input                               to_uart_valid;

input                               UART_RXD;

// Bidirectionals

// Outputs
output[(DW-1):0]                    from_uart_data;
output                              from_uart_error;
output                              from_uart_valid;

output                              to_uart_ready;

output                              UART_TXD;

/*****************************************************************************
 *                           Constant Declarations                           *
 *****************************************************************************/

/*****************************************************************************
 *                 Internal Wires and Registers Declarations                 *
 *****************************************************************************/

// Internal Wires
wire        [(DW-1):0]  read_data;

wire                    write_data_parity;
wire        [ 7: 0]     write_space;

// Internal Registers

// State Machine Registers
```

```
/*****************************************************************************
 *                           Finite State Machine(s)                         *
 *****************************************************************************/


/*****************************************************************************
 *                            Sequential Logic                               *
 *****************************************************************************/

// Output Registers

// Internal Registers

/*****************************************************************************
 *                           Combinational Logic                             *
 *****************************************************************************/

// Output Assignments
assign from_uart_data    = read_data;
assign from_uart_error   = 1'b0;

assign to_uart_ready     = (|(write_space));

// Internal Assignments
assign write_data_parity = (^(to_uart_data)) ^ ODD_PARITY;

/*****************************************************************************
 *                             Internal Modules                              *
 *****************************************************************************/

altera_up_rs232_in_deserializer RS232_In_Deserializer (
      // Inputs
      .clk                    (clk),
      .reset                  (reset),

      .serial_data_in         (UART_RXD),

      .receive_data_en        (from_uart_ready),

      // Bidirectionals

      // Outputs
      .fifo_read_available    (),

      .received_data_valid    (from_uart_valid),
      .received_data          (read_data)
);
defparam
      RS232_In_Deserializer.CW                  = CW,
      RS232_In_Deserializer.BAUD_TICK_COUNT     = BAUD_TICK_COUNT,
      RS232_In_Deserializer.HALF_BAUD_TICK_COUNT= HALF_BAUD_TICK_COUNT,
      RS232_In_Deserializer.TDW                 = TDW,
      RS232_In_Deserializer.DW                  = (DW - 1);

altera_up_rs232_out_serializer RS232_Out_Serializer (
      // Inputs
      .clk                    (clk),
      .reset                  (reset),

      .transmit_data          (to_uart_data),
```

```
        .transmit_data_en       (to_uart_valid & to_uart_ready),

        // Bidirectionals

        // Outputs
        .fifo_write_space       (write_space),

        .serial_data_out        (UART_TXD)
);
defparam
        RS232_Out_Serializer.CW                   = CW,
        RS232_Out_Serializer.BAUD_TICK_COUNT      = BAUD_TICK_COUNT,
        RS232_Out_Serializer.HALF_BAUD_TICK_COUNT = HALF_BAUD_TICK_COUNT,
        RS232_Out_Serializer.TDW                  = TDW,
        RS232_Out_Serializer.DW                   = (DW - 1);
endmodule
```

6.7 thermicon.ino – on-board Arduino program

```
/*************************************************************
 * ELEX 7660 - Thermicon
 *
 * Thermicon on Arduino is a wireless remote controled environmental data
 * colecting device.
 *
 * Developed by Chris Barreiro Stewart and Jing Huang
 *************************************************************/

/* Libraries */
#include <SoftwareSerial.h>
#include "DHT.h"

/* RX = D8, TX = D9 */
SoftwareSerial BTSerial(RX, TX);

/* Pin Definitions */
#define RX 8
#define TX 9
#define AIN1 6
#define BIN1 4
#define AIN2 7
#define BIN2 3
#define PWMA A1
#define PWMB A2
#define STBY 5
#define DHTPIN 12
#define DHTTYPE DHT22 // Model of DHT sensor
#define timeout 10
#define Stop 0
#define Turn 25
#define low 0
#define high 1

/* Variable Definitions */
float _RIGHT, _LEFT;
int speedL = 150;
int speedR = 150;
int negspeedL = -150;
int negspeedR = -150;
char junk;
int humidity = 0;
```

```
int temp = 0;
int x = 0;
int y = 0;
int data;

/* Objects */
DHT dht(DHTPIN, DHTTYPE);

/* Environment Sensor Timer Variables */
unsigned long hpreviousMillis = 0;
const long hinterval = 5000;   // Timer Interval in milliseconds


void setup()
{
	/* Set the baud rate to 38400 for HC-06 */
	BTSerial.begin(38400);
	delay(1000);

	/* Set PIN Directions */
	pinMode(RX, INPUT);
	pinMode(TX, OUTPUT);

	/* Set UART timeout */
	BTSerial.setTimeout(timeout);

	/* Break */
	Drive(Stop, Stop);
	shortBrake(true, true);


	/* Definitions */
	pinMode(AIN1, OUTPUT);
	pinMode(temp, INPUT);
	pinMode(BIN1, OUTPUT);
	pinMode(AIN2, OUTPUT);
	pinMode(BIN2, OUTPUT);
	pinMode(PWMA, OUTPUT);
	pinMode(PWMB, OUTPUT);
	pinMode(STBY, OUTPUT);

	/* Set Motor Driver to Standbye */
	digitalWrite(STBY, HIGH);
}

void loop()
	{

	/* Check for Overflow on UART */
	if(BTSerial.overflow()){
		delay(10);
	}

	/* Check UART for data */
	if(BTSerial.available()){
		data = BTSerial.read();    // Read values from joystick connected to FPGA
		junk = BTSerial.read();

	/* Split Data into x and y coordinates */
		y = data >> 4;
		x = data & 0xf;
```

```
}

        /* Y axis centered */
        if( y < 12 && y > 4 ){
              if( x > 4){
                    /* X axis centered, STOP! */
                    if( x < 12){
                          Drive(Stop, Stop);
                          shortBrake(true, true);
                    }
                    /* Right Turn */
                    else {
                          Drive(speedL,Stop);
                    }
              }
              /* Left Turn */
              else {
                    Drive(Stop, speedR);
              }
        }

        /* Y axis forward */
        if( y >= 12 ){
              /* X axis left */
              if (x <= 4){
                    Drive(speedL - Turn, speedR + Turn);
              }
              /* X axis right */
              else if (x >= 12){
                    Drive(speedL + Turn, speedR - Turn);
              }
              /* X axis centered */
              else{
                    Drive(speedR, speedL);
              }
        }

        /* Y axis backward */
        if( y <= 4 ){
              /* X axis left */
              if (x <= 4){
                    Drive(negspeedR + Turn, negspeedL - Turn);
              }
              /* X axis right */
              else if (x >= 12){
                    Drive(negspeedR - Turn, negspeedL + Turn);
              }
              /* X axis centered */
              else{
                    Drive(negspeedR, negspeedL);
              }
        }

  /* Get Environment Data from senor */
  getENVDATA();


  /* Send Humuduty to FPGA */
  BTSerial.write(humidity);

  /* Send Temperature to FPGA */
```

```
	// BTSerial.write(temp);
}


/* Function Definitions */

/* Move Motor A Forward */
void fwdA()
{
	digitalWrite(AIN1, low); // Set AIN1 LOW
	digitalWrite(AIN2, high); // Set AIN2 HIGH

}

/* Move Motor A Backward */
void revA()
{
	digitalWrite(AIN1, high); // Set AIN1 HIGH
	digitalWrite(AIN2, low); // Set AIN2 LOW
}

/* Move Motor B Forward */
void fwdB()
{
	digitalWrite(BIN1, low); // Set BIN1 LOW
	digitalWrite(BIN2, high); // Set BIN2 HIGH
}

/* Move Motor B Backward */
void revB()
{
	digitalWrite(BIN1, high); // Set BIN1 HIGH
	digitalWrite(BIN2, low); // Set BIN2 LOW
}

/*   General purpose drive function.
 *   Values for RIGHT and LEFT should be between -200 - 200.
 *   Positive numbers for fwd, negative for rev.
 *   200 and -200 are full speed.
 */
void Drive(float RIGHT, float LEFT) {
	// Store Values For Future Use
	_RIGHT=RIGHT;
	_LEFT=LEFT;

	if (RIGHT < 0) {
		revA();
		RIGHT *= -1;
	}

	else {
		fwdA();
	}

	if (LEFT < 0) {
		revB();
		LEFT *= -1;
	}

	else {
		fwdB();
```

```
	}

	analogWrite(PWMA, RIGHT);
	analogWrite(PWMB, LEFT);
}

/*  standby() differs from brake in that the outputs are hi-z instead of
 *  shorted together. After the constructor is called, the motors are in
 *  standby.
 */

void standby(bool disableMotors)
{
	if (disableMotors)
	{
		digitalWrite(STBY, low); // Set STBY LOW
	}
	else
	{
		digitalWrite(STBY, high); // Set BIN1 HIGH
	}
}

/*  shortBrake() shorts the motor leads together to drag the motor to a halt
 *  as quickly as possible.
 */
void shortBrake(bool brakeRight, bool brakeLeft)
{
	if (brakeRight)
	{
		digitalWrite(AIN1, high); // Set AIN1 HIGH
		digitalWrite(AIN1, high); // Set AIN1 HIGH

	}
	if (brakeLeft)
	{
		digitalWrite(BIN1, high); // Set BIN1 HIGH
		digitalWrite(BIN1, high); // Set BIN1 HIGH
	}
}


/*  getENVDATA()
 *  Gets Temperature and Humidy from DHT 22 to Arduino
 */

int getENVDATA() {

	unsigned long hcurrentMillis = millis(); // Set current time

	if(hcurrentMillis - hpreviousMillis >= hinterval) {
		// save the last time you read the sensor
		hpreviousMillis = hcurrentMillis;

		// Reading temperature for humidity takes about 250 milliseconds!
		// Sensor readings may also be up to 2 seconds (it's a very slow sensor)
		humidity = dht.readHumidity();          // Read humidity (percent)
		temp = dht.readTemperature();     // Read temperature as Celcius

		// Check if any reads failed and exit early (to try again).
		if (isnan(humidity) || isnan(temp)) {
```

```
                humidity = 0;
                temp = 0;
            }
        }
}
```