Quadrocopter Flight Control

Department of Electrical and Computer Engineering

Bradley University

Eric Backman

Advisor: Dr. Malinowski

May 13, 2012

Abstract

The goal of Quadrocopter Flight Control is to implement digital remote control for a quadrocopter with an aerial collision avoidance system. The quadrocopter is controlled semi-autonomously using a Linux microcontroller (LUC) running collision avoidance algorithm, and interfaced to wireless communication system. The LUC utilizes an Atmel microprocessor to interface to the array of infrared distance sensors and a flight stabilizing controller that is built in into the quadrocopter. A computer controlled by a remote operator sends commands wirelessly to the on-board LUC. This controller alters these commands as necessary to avoid colliding with nearby objects using the data from the infrared sensors. The distances from the infrared sensors is adjusted to account for the tilt of the quadrocopter. To prevent accidents if the wireless connection is lost, the quadrocopter is capable of stabilizing the flight and an emergency landing. A successful system has been implemented but could not do any flight testing in limited time.

Table of Contents

Introduction

Quadrocopters are the small aircraft of choice for many people due to their ability to fly through the air much like a plane while maintaining the helicopter's ability to hover and move at low airspeeds. By equipping such a copter with distance sensors a simple object avoidance scheme can be added to provide safety through human error. These sensors could eventually be used to map the surrounding terrain and navigate through tight areas.

This project uses microcontroller Linux to control a quadrocopter. Most of the work was integrating the BeagleBoard with the sensors and joystick in order to create simple flight. The BeagleBoard sends these commands to a slave microcontroller that simulates the RC standard signals required by the quadrocopter flight controller. The quadrocopter takes wireless signals from a joystick initially to allow for manual control. Simple object avoidance was implemented using infrared sensors.

System Description

Quadrocopters are ideal for small scale flying robots due to their ability to hover like a helicopter without the need to change the rotor blade pitch angle. This simplifies their design and control. In the prior year, Brad Bergerhouse, Nelson Gaske, and Austin Wenzel worked on an autonomous quadrocopter. They constructed the quadrocopter platform, installed a real time operating system on BeagleBoard, and started sensor implementation.

The primary goal for this project was to get the quadrocopter flying and responding to inputs from sensors using microcontroller Linux. The platform was already built but it needed controllers and sensors to function. A joystick was connected to another BeagleBoard or PC that connects wirelessly to the onboard controller. This allows a user to control the quadrocopter. Simple object avoidance was programmed to prevent the quadrocopter from crashing into walls, as well as simple flight patterns to allow for semi-autonomous flight and navigation.

The project was to replace the remote control portion of the remote control quadrocopter platform with my semi-autonomous control system. The quadrocopter already has a built in flight controller with accelerometers to maintain stability without user input. Controlling this requires me to simulate the RC signals to the flight controller. This was done by using the Atmega's timers to create 50 Hz pulse width modulation (PWM) signals with different duty cycles corresponding to different commands.

The Atmega controller uses the built in analog to digital converters to take the inputs from the sensors and send them to the BeagleBoard through a serial connection. The BeagleBoard takes these inputs and uses them to influence the command coming either from a joystick. It generates a signal for yaw, pitch, roll and elevation and it sends these signals to the Atmega controller that generates the required PWMs and give them to the built-in flight controller. The flight controller takes the commands and generates the PWMs for the four motors.

This system is shown in Figure 1 below.

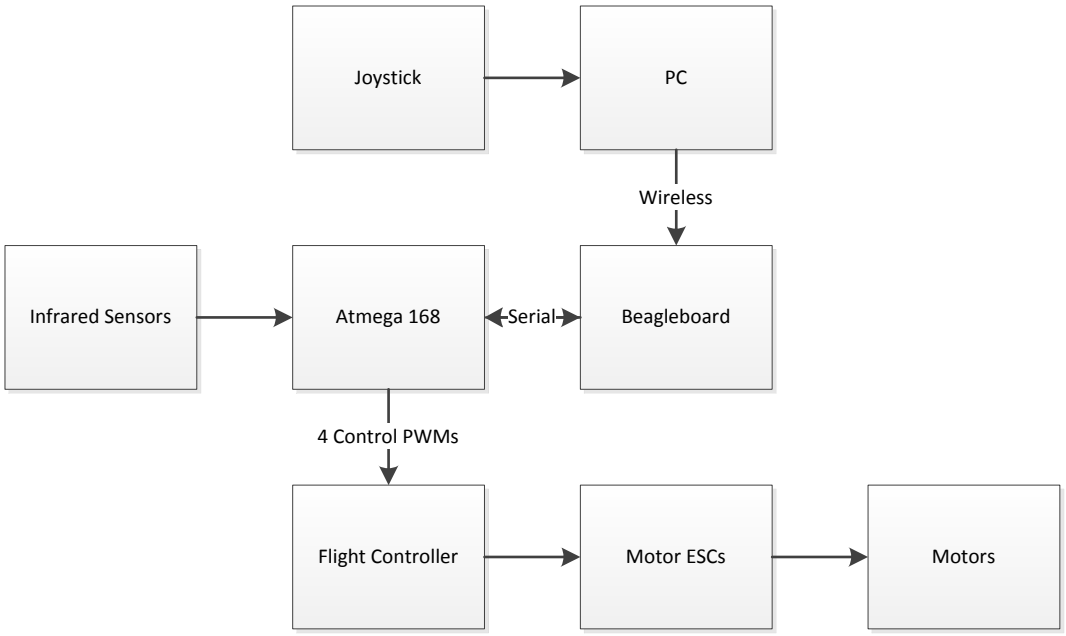


Figure 1. System Block Diagram

The following hardware was used in the design

- BeagleBoard xM
- Xaircraft X650 Quadrocopter
- Atmega 168
- Max 232
- 6 IR distance sensors (Sharp GP2Y0A02YK0F)
- Battery (11.1V 3S LiPo)
- Belkin Wireless USB network card
- 5V Switching Power Regulator

The Beagleboard xM acts as the main controller, deciding the eventual PWMs of the system based on joystick and sensor inputs. The Quadrocopter includes a flight controller, Motor Electronic Speed Controllers(ESCs), Motors, and the body that everything is mounted on. This is controlled using RC standard PWMs generated by the Atmega controller. The Atmega controller reads the analog voltages sent out by the infrared sensors and generates the PWMs to interact with the flight controller. The MAX232 chip corrects the voltage levels for the serial communication between the BeagleBoard and Atmega controller as shown in Figure 1. The battery and the switching power regulator provide the

power for the system. The wireless network card allows the BeagleBoard to wirelessly connect to the computer running the joystick. The specifics of each subsystem are explained below.

Power Distribution

BeagleBoard, Atmega 168, and infrared sensors run on 5V. A switching voltage regulator was needed to run these this system using the 11.1 V battery. The electric circuit for the power converted is included in appendix A. This resulted in a power distribution as shown below in figure 2.

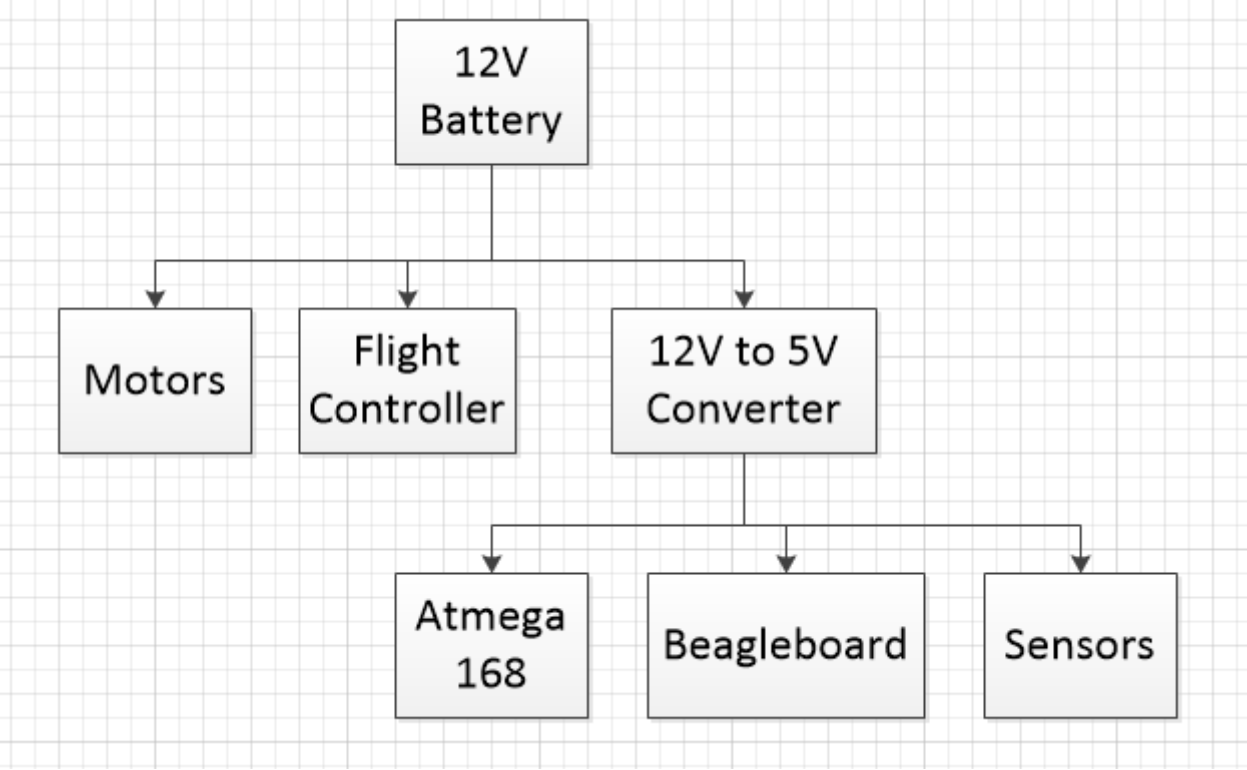


Figure 2. Power Distribution

Infrared Sensors

The infrared sensors generate a voltage from about 0-3V as shown in Figure 3 below. The outputs of 6 sensors are continually read in a round robin fashion using an analog multiplexer connected to the analog-to-digital converters located on the Atmega controller. The system uses a 5V reference voltage and has 8 bits of resolution. This yields an error of +/- 20 mV which is acceptable for this project. Two more bits of resolution could be utilized if needed.

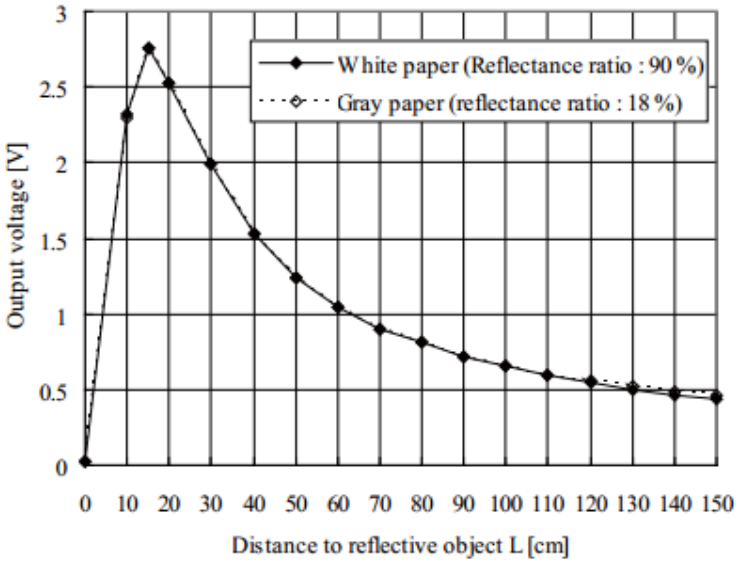


Figure 3. Infrared Sensor Characteristics

The sensor array is made of 6 SHARP GP2Y0A02YK0F sensors with one pointing in each direction (front, back, up, down, left, and right). This allows for simple object avoidance requiring minimal sensors and complexity of controls. This is the simplest arrangement that allows full coverage and gives me the time to finish the whole project.

This 8 bit value is converted back to an approximate voltage on the BeagleBoard using the equation:

$$Voltage(\frac{V}{100}) = ADC_{output} * \frac{100 * Reference\ Voltage}{256}$$

This allows us to compare it to Figure 3 to determine the approximate distance. This could be done using a look-up table or a mathematical approximation. The reference voltage was obtained experimentally by inputting a voltage and comparing it to the result. This assumes the object is greater than 20 cm away which is a valid assumption due to the sensor arrangement.

Serial Port Communication

The serial port between the Atmega 168 and the BeagleBoard contains 2 pieces of information using a 9600 baud rate. One set of all infrared sensor readings is sent every 70 ms from the Atmega controller to the BeagleBoard. This is slower than was originally planned but due to the set-up of the BeagleBoard

command loop the data lags behind. This could be solved by running multiple tasks at once using threads and running at a faster baud rate.

The other information being sent is the PWM commands from the BeagleBoard to the Atmega controller. The 5 byte packet is sent once every command loop cycle which takes about 10 ms. The first byte lets the microcontroller know the message is starting and the next 4 are pitch, roll, throttle and yaw respectively. These signals are put into an incoming data buffer which is used to update the PWM array every 20 milliseconds when the PWM signals restart. These values are then compared with a counter every microsecond to determine when the PWMs turn off.

The TTL logic voltage levels of the Atmega serial port are not compatible with the standard RS232 port which is used on the BeagleBoard so a voltage conversion circuit is needed. A MAX 232 chip is used as shown in figure 4 below to allow for serial communication.

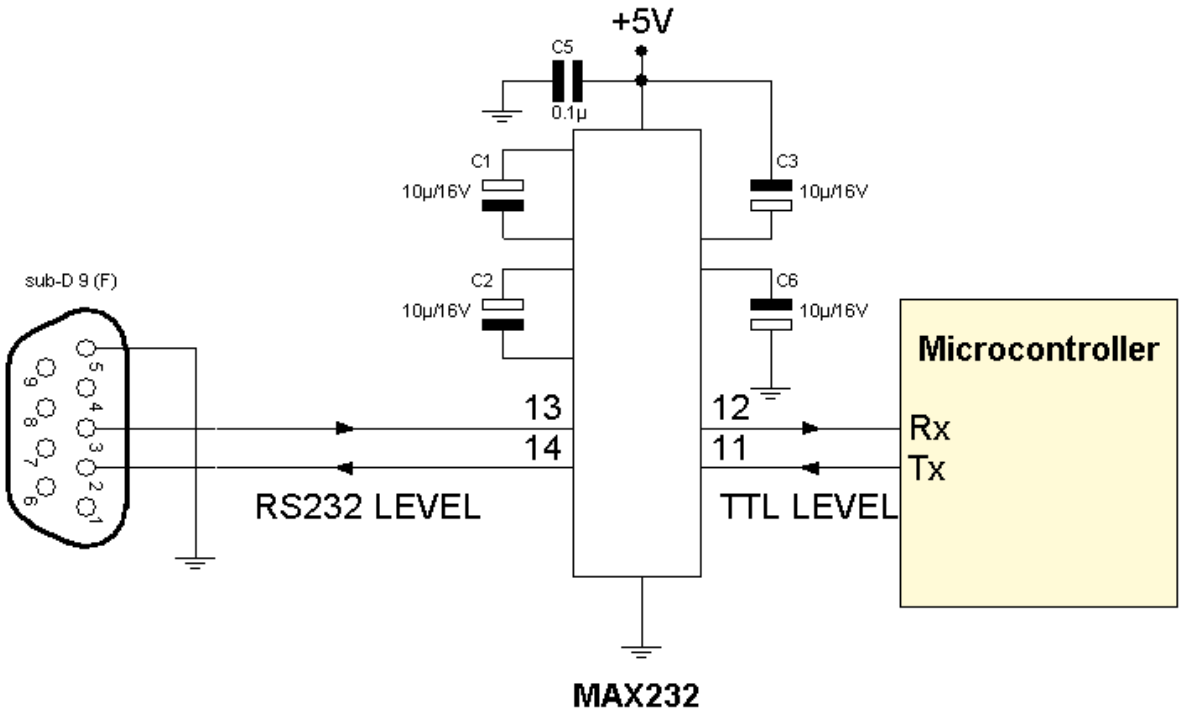


Figure 4. MAX232 Circuit

RC Standard PWMs

The industry standard RC PWM consists of a number of pulses occurring with a frequency of 50Hz. The pulses vary from -100%PWM to +100%PWM indicating a high voltage time of 1ms to 2ms respectively. The remainder of the 20ms period is low voltage. The 0% PWM then becomes a pulse with a 1.5ms high time followed by 18.5ms of low voltage. Figure 5 below demonstrates the shape of the RC pulses

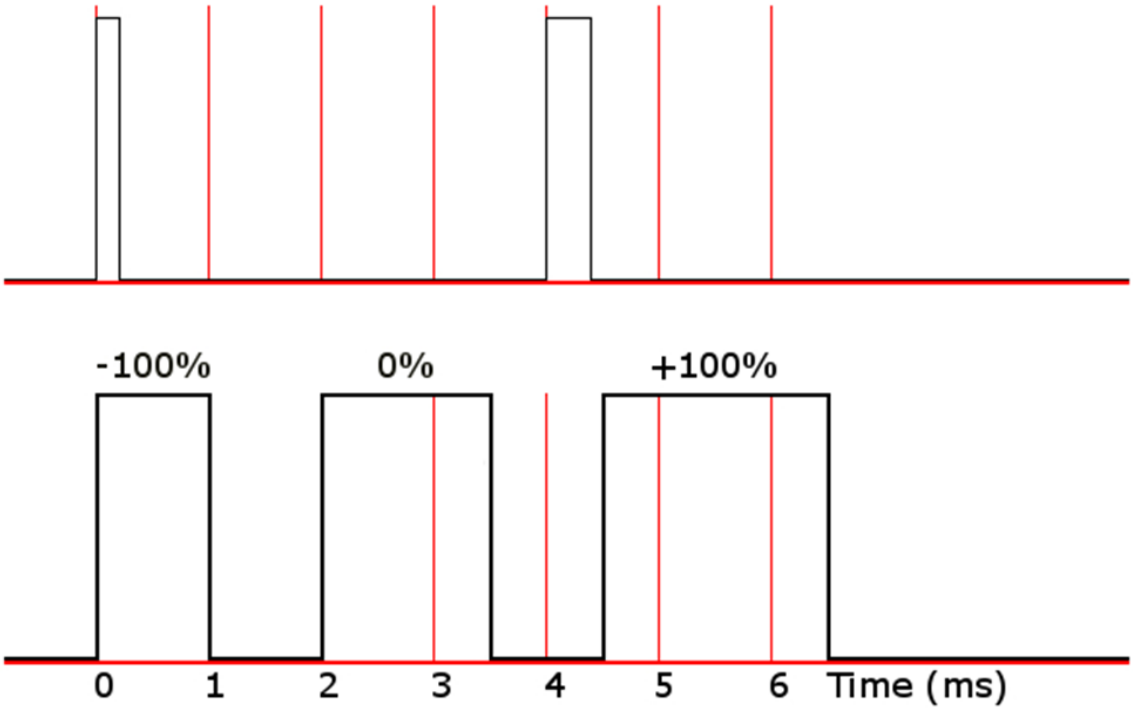


Figure 5. RC Standard PWM

The FC1212 flight controller on the quadrocopter receives 4 PWM signals corresponding to yaw, pitch, roll and throttle. The Atmega 168 receives 5 bytes of information from the serial port. The first byte signals the start of the message while the next 4 contain the data which is thrown into an array. This array is checked every 10 microseconds to turn off the digital outputs running the PWMs at the correct times.

Joystick

A simple joystick reading application using the windows joystick libraries was used to read the joystick inputs. The most significant byte for the x,y,z and r values as well as the first 8 buttons are sent to the BeagleBoard to be read. They are sent over a wireless network with the BeagleBoard as a server and the computer as a client. These signals are read on the BeagleBoard and analyzed to determine the PWM signals to be sent to the Flight Controller.

Results

The system is completed and mounted on the quadrocopter. All subsystems were tested individually and together as shown in Figure 6. However, there was not sufficient time to do any flight testing. This means none of the object avoidance or stability was tested. The distance measurements and throttle are the two things that can be checked and they work exactly as designed. The system quickly responds to the joystick inputs but the infrared sensor data seems slower than expected.

Figure 6.Quadrocopter and its Complete Control System

Future Work

The first thing to do with the quadrocopter would be to perform basic flight testing to make sure the system is stable. The object avoidance could be tested and tuned to desired performance. That would finish up this project and allow expansion into stronger aerial navigation techniques.

The main command loop on the BeagleBoard seems slow and could harm high speed performance. This could be remedied by adding threads to run communication. The processor is being wasted while waiting to read the serial port and network packets. This would boost the performance greatly. One of the things that were going to be added was some code that would compensate for the tilt of the

quadrocopter. This would be done by estimating the tilt using the command angle and using it to calculate the distance if it were perpendicular to that surface. This would be effective in environments with square walls and would allow more effective measurements.

A camera could be attached that would create opportunities for aerial image processing and allow the quadrocopter to more accurately detect and avoid obstacles. The BeagleBoard has a built-in DSP and the processing power to do many more interesting things.

Following the integration described above, the platform will provide the department with an excellent source of future project topics including: aerial navigation and mapping, 3-dimensional localization, and aerial drones. These topics will be filled with new and exciting projects for future teams and provide further opportunity to expand the project possibilities at Bradley University.

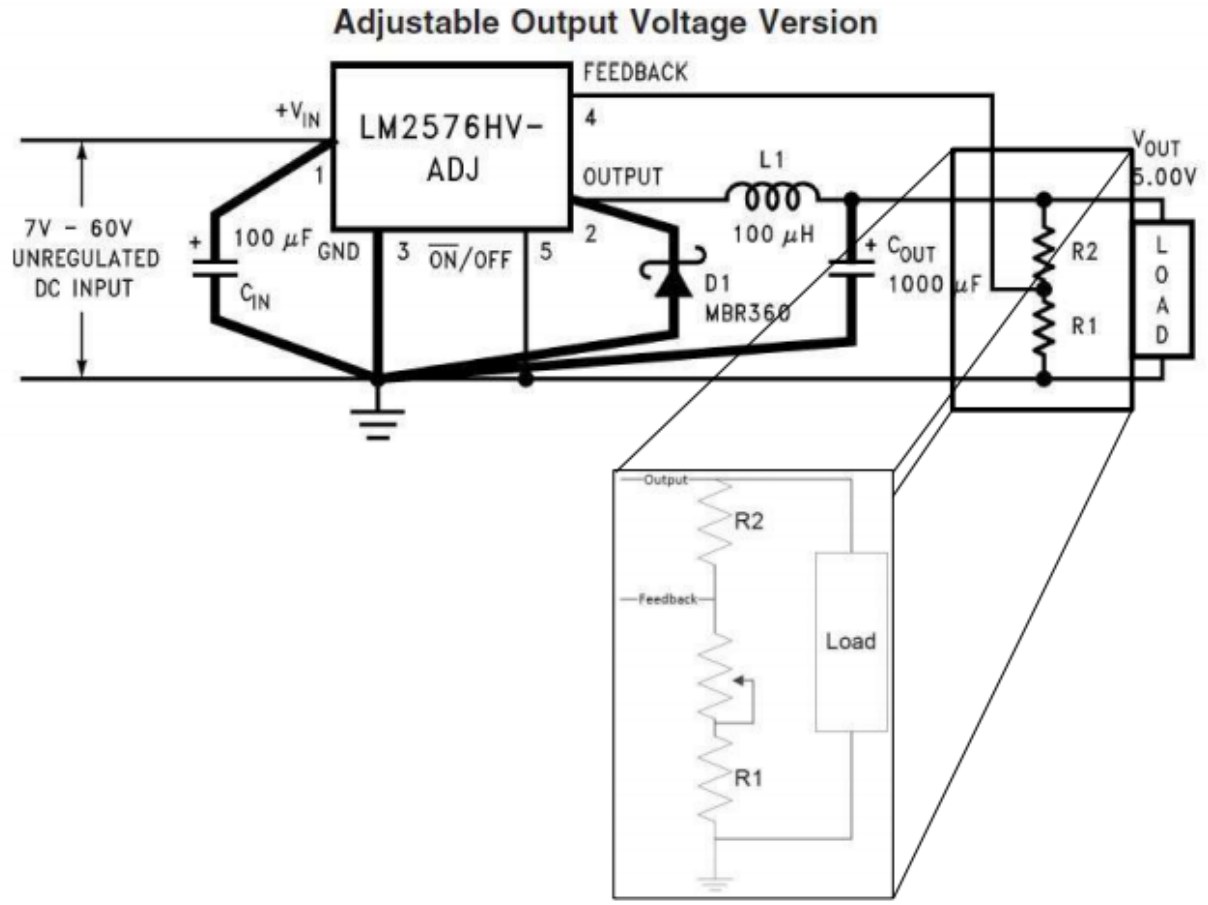
References

[1] Brad Bergerhouse, Nelson Gaske, Austin Wenzel. Aerial Collision Avoidance System. Senior Project, Electrical and Computer Engineering Department, Bradley University, May 2012, http://cegt201.bradley.edu/projects/proj2012/quadcptr/

[2] Introduction to Autonomous Mobile Robots, 2/ed., by R. Siegwart, I. R. Nourbakhsh, D. Scaramuzza, MIT Press, 2011, ISBN: 978-0262015356

[3] SHARP Electronics TDS for GP2Y0A02YK0F. Dec. 01,2006 SHARP Electronics. <http://www.sharpsma.com/webfm_send/1487>

Appendix A: 11.1V to 5V Switching Voltage Regulator



Appendix B: BeagleBoard Code

Main.c

```
#define closesocket close
#define SOCKET int
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>
#include <stdio.h>              /* necessary to get sprintf    */
#include <string.h>

#define PROTOPORT 1200

#include <time.h>
#include "serial.h"

int main(){
//UDP socket init
   struct  protoent *ptrp;  /* pointer to a protocol table entry   */
   struct  sockaddr_in sad; /* structure to hold server's address  */
   struct  sockaddr_in cad; /* structure to hold client's address  */
   SOCKET  sd;             /* socket descriptor - integer        */
   int     alen;           /* length of address                  */
   int     port;           /* protocol port number               */
           char buf[1000];     /* buffer for string the server sends  */
   int  visits = 0;     /* counts client connections          */
   int  n;             /* number of characters received      */
   int  m;              /* number of characters sent back      */
           int joycount = 0;
           port = PROTOPORT;
   if (port <= 0)                /* test for illegal value      */
   {                            /* print error message and exit */
      fprintf(stderr,"bad port number");
      exit(1);
   }
   memset((char *)&sad,0,sizeof(sad)); /* clear sockaddr structure */
   sad.sin_port = htons((u_short)port); /* set server port number  */
   sad.sin_family = AF_INET;          /* set family to Internet   */
   sad.sin_addr.s_addr = INADDR_ANY;  /* set the local IP address */
   /* Map UDP transport protocol name to protocol number */
```

```
   ptrp = getprotobyname("udp");
   if ( ptrp == 0) {
      fprintf(stderr, "cannot map \"udp\" to protocol number");
      exit(1);
   }
   /* Create a socket */
sd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
   //sd = socket(PF_INET, SOCK_DGRAM, ptrp->p_proto);
   if (sd < 0) {
      fprintf(stderr, "socket creation failed");
      exit(1);
   }
   /* Bind a local address to the socket */
   if (bind(sd, (struct sockaddr *)&sad, sizeof(sad)) < 0) {
      fprintf(stderr, "bind failed");
      exit(1);
   }
struct timeval to;
to.tv_sec =0;
to.tv_usec =1000;
if(setsockopt(sd, SOL_SOCKET, SO_RCVTIMEO, &to, sizeof(to)) <0 ){
            printf("Error in socket\n");
            exit(0);
}

//Serial init
  int fd;
  int incc=0, incc2=0;                  /* input character count (from fread) */
  int outcc=0;                /* output character count (from write) */
  const int BUFFER_SIZE = 256;
  unsigned char buffer[1];//BUFFER_SIZE];
fd= serial_init();
int y = 0;
//int x = clock();
char *serial_device = "/dev/ttyS2";
unsigned char output[5];
output[0] = 0xFF;//Message start
output[1] = 0;//PWM 1
output[2] = 0;//PWM 2
output[3] = 0;//PWM 3
output[4] = 0;//PWM 4
int distance[6];
```

```
int count = 7;
char joystick[5];
char land=0, takeoff = 0, off = 1;
char landcount;

  while ( 1) {

        //Read 1 infrared signal from serial port and put in it distance array
        incc2 = read( fd, buffer, 1 );
        if ( incc2 > 0 ) {
                if(buffer[0] == 0x01){
                        count = 0;
                        fprintf( stderr, "New Line\n");
           int i = 0;
                while( i<6){
                                fprintf( stderr, "%d ", distance[i]);
                                i++;
                }
                }
                else if(count <7){
                        distance[count] = (1.647*buffer[0]);  //Convert from character to Volts*100
                        count++;
                }
        } else if(incc2<0){
                  perror( serial_device );
                  break;
        }

        //Receive joystick packet
        joycount = 0;
   alen = sizeof(cad);
   n = recvfrom(sd,buf,sizeof(buf),0,(struct sockaddr*)&cad,&alen);
   if (n<0)
   {
                if(landcount <25)          //If haven't received packet in 25 loops, go to landing mode
                        landcount++;
                else{
                        land = 1;
                        takeoff= 0;
                        fprintf( stderr, "Land\n");
                }
   }
```

```
else if(n>=0) /* We could receive a useful empty packet */
{
		landcount = 0;
		joystick[0] = buf[0];
		joystick[1] = buf[1];
		joystick[2] = buf[2];
		joystick[3] = buf[3];
		joystick[4] = buf[4];
}


	//control algorithms
	if(((joystick[4])&0x01) == 0x01){
		takeoff = 1;
		land = 0;
		off = 0;
		fprintf( stderr, "Takeoff\n");
	}
	if(((joystick[4])&0x02) == 0x02){
		land = 1;
		takeoff= 0;
		off = 0;
		fprintf( stderr, "Land\n");
	}
	if(((joystick[4])&0x04) == 0x04){
		takeoff = 0;
		land = 0;
		off = 0;
		fprintf( stderr, "Normal Flight\n");
	}
	if(((joystick[4])&0x08) == 0x08){
		off = 1;
		takeoff = 0;
		land = 0;
		fprintf( stderr, "Kill Switch\n");
	}
	if(off == 1){
		output[1] = 150;
		output[2] = 150;
		output[3] = 100;
		output[4] = 150;
	}
```

```
        else if(land == 1){
                output[1] = 150;
                output[2] = 150;
                output[3] = 140;
                output[4] = 150;
        }
        else if(takeoff == 1){
                output[1] = 150;
                output[2] = 150;
                output[3] = 160;
                output[4] = 150;
                if(distance[4]>100)
                        output[3] = 150;
        }
        else{
                output[1] = joystick[0]*100/254+100;
                output[2] = joystick[1]*100/254+100;
                output[3] = joystick[2]*100/254+100;
                output[4] = joystick[3]*100/254+100;
                if(distance[0]>100 && joystick[0]<127)
                        output[1] = 150;
                if(distance[1]>100 && joystick[0]>127)
                        output[1] = 150;
                if(distance[2]>100 && joystick[1]<127)
                        output[2] = 150;
                if(distance[3]>100 && joystick[1]>127)
                        output[2] = 150;
                /*if(distance[4]>100 && joystick[2]<127)
                        output[3] = 150;
                if(distance[5]>100 && joystick[2]>127)
                        output[3] = 150;*/
        }

                //Send data to serial port
                outcc = write(fd, output,5);
                 if ( outcc < 0 ) {
                                perror( serial_device );
                                break;
                 }
 }
return 1;
}
```

Serial.h

```
#include <termios.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>


int serial_init();
```

Serial.c

```
//* program to echo characters from stdin to the serial port *

#include <termios.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

volatile int echo_serial_go;

/* Echo characters to the serial port from stdin, until an error occurs,
   or until some other thread sets echo_serial_go to zero. */
int
serial_init() {
char *serial_device = "/dev/ttyS2";
speed_t baud_rate = B9600;
  int fd;                   /* file descriptor for serial I/O */
  struct termios prev_tio;
  struct termios tio;         /* port i/o settings */


  fd = open( serial_device,    /* open serial device (duh...) */
       O_RDWR | O_NOCTTY);    /* read/write/serial not controlling terminal*/
  if (fd < 0) { perror( serial_device ); return(-1); }
  tcgetattr( fd, &prev_tio);    /* preserve the current port settings */
```

```
                    /* set up port for reading or writing */
  memset( &tio, '\0', sizeof( struct termios ) );
  tio.c_iflag = IGNPAR;       /* input mode: ignore framing/parity errors */
  tio.c_oflag = 0;            /* output mode */
  tio.c_cflag = (             /* control mode: */
       baud_rate              /*  must use predefined codes (see termios) */
       | CS8                  /*  8 bits per character */
                              /*  no parity (PARENB enables parity) */
       | CSTOPB               /*  one stop bit (CSTOPB sets two) */
       | CREAD                /*  enable receiver */
       | CLOCAL               /*  ignore modem control lines */
       | CRTSCTS );           /*  use flow control lines */
  tio.c_lflag = 0;            /* local mode */

                    /* adjust behavior for read/write functions */
  tio.c_cc[VTIME]   = 1;      /* never timeout; if non-zero, timeout occurs*/
                    /*  if no character after c_cc[VTIME] * 0.1s */
  tio.c_cc[VMIN]    = 0;      /* a minimum of 8 char's retrieved each read */

  tcflush( fd, TCIFLUSH);     /* flush input buffer */
  tcsetattr( fd, TCSANOW, &tio);/* install new port settings (immediately) */
return fd;
}
```

Appendix C: Joystick Code

```
//* Example illustrating Joystick access in Windows *

#include <limits.h>
// We need winmm.h but it needs some more of windows.h
// Add to the linked libraries list: winmm.lib/* * code for example client program that uses UDP * */

#ifndef unix
#include <winsock2.h>
/* also include Ws2_32.lib library in linking options */
#else
#define closesocket close
#define SOCKET int
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>
/* also include xnet library for linking; on command line add: -lxnet */
#endif

#include <stdio.h>
#include <string.h>

#define PORT 1200
                        /* default protocol port number */

char    localhost[] = "136.176.16.17";   /* default host name          */
/*------------------------------------------------------------
 * Program:   client
 *
 * Purpose:   allocate a socket, connect to a server, and print all output
 *
 *------------------------------------------------------------
 */
int main()
{
   struct  hostent  *ptrh;  /* pointer to a host table entry      */
   struct  protoent *ptrp;  /* pointer to a protocol table entry  */
   struct  sockaddr_in sad; /* structure to hold an IP address    */
   int     alen;          /* length of address                  */
   SOCKET  sd;            /* socket descriptor - integer        */
```

```
   char    host[256];      /* pointer to host name             */

#ifdef WIN32
   WSADATA wsaData;
   if(WSAStartup(0x0101, &wsaData)!=0)
   {
      fprintf(stderr, "Windows Socket Init failed: %d\n", GetLastError());
      exit(1);
   }
#endif

   memset((char *)&sad,0,sizeof(sad)); /* clear sockaddr structure */
   sad.sin_family=AF_INET;        /* set family to Internet     */

   strcpy(host,localhost); // host=localhost

   /* Convert host name to equivalent IP address and copy to sad. */
   ptrh = gethostbyname(host);
   if ( ((char *)ptrh) == NULL ) {
      fprintf(stderr, "invalid host: %s\n", host);
      exit(1);
   }
   memcpy(&sad.sin_addr, ptrh->h_addr, ptrh->h_length);

   sad.sin_port = htons((u_short)PORT); /* use default port number */
   alen = sizeof(sad);

   /* Map TCP transport protocol name to protocol number. */
   ptrp = getprotobyname("udp");
   if ( ptrp == 0) {
      fprintf(stderr, "cannot map \"udp\" to protocol number\n");
      exit(1);
   }

   /* Create a socket. */
   sd = socket(PF_INET, SOCK_DGRAM, ptrp->p_proto);
   if (sd < 0) {
      fprintf(stderr, "soclet creation failed\n");
      exit(1);
   }

            short keeprunning = 1;
```

```
        while (keeprunning)
  {
                int  n;          /* number of characters received     */
                int  m;          /* number of characters sent back    */
                char buf[1000];     /* buffer for data from the server    */
    JOYCAPS    jc;          // to keep features of the joystick
    JOYINFOEX  lastJoyState;  // to keep joystick state

    if (joyGetNumDevs()==0)
    {
      fprintf(stderr, "Please connect a joystick\n");
      Sleep(5000);
      continue;
    }

    if (joyGetDevCaps(JOYSTICKID1, &jc, sizeof(jc))!=JOYERR_NOERROR)
    {
      fprintf(stderr, "Please connect a compatible joystick\n");
      Sleep(1000);
      continue;
    }

    lastJoyState.dwSize=sizeof(lastJoyState);
    lastJoyState.dwFlags=JOY_RETURNALL | JOY_RETURNPOVCTS | JOY_RETURNCENTERED |
JOY_USEDEADZONE;

    {
      const unsigned int naxes = jc.wNumAxes;
      const unsigned int nbuts = jc.wNumButtons;
      const int MID_VALUE =(int)(USHRT_MAX/2);

      int cur_b = 0;
      int cur_x = 0;
      int cur_y = 0;
      int cur_z = 0;
      int cur_r = 0;
      int cur_u = 0;
      int cur_v = 0;
      int cur_h = 0;

      int old_b = 0;
```

```
int old_x = 0;
int old_y = 0;
int old_z = 0;
int old_r = 0;
int old_u = 0;
int old_v = 0;
int old_h = 0;

int cnt = 0;

while(keeprunning)
{
                                Sleep(200);
  if (joyGetPosEx(JOYSTICKID1, &lastJoyState) != JOYERR_NOERROR)
  {
    fprintf(stderr, "Please reconnect the joystick\n");
    Sleep(1000);
    break; // exit to the outer loop and check for a new joystick connection
  }

  old_b = cur_b;
  old_x = cur_x;
  old_y = cur_y;
  old_z = cur_z;
  old_r = cur_r;
  old_u = cur_u;
  old_v = cur_v;
  old_h = cur_h;

  cur_b = lastJoyState.dwButtons;
  cur_x = (int)(lastJoyState.dwXpos)-MID_VALUE;
  cur_y = (int)(lastJoyState.dwYpos)-MID_VALUE;
  cur_z = (int)(lastJoyState.dwZpos)-MID_VALUE;
  cur_r = (int)(lastJoyState.dwRpos)-MID_VALUE;
  cur_u = (int)(lastJoyState.dwUpos)-MID_VALUE;
  cur_v = (int)(lastJoyState.dwVpos)-MID_VALUE;
  cur_h = lastJoyState.dwPOV;

  cnt++; // to force display/send every so many samples even if no change
```

```
        //if
(cnt>45||cur_b!=old_b||cur_x!=old_x||cur_y!=old_y||cur_z!=old_z||cur_r!=old_r||cur_u!=old_u||cu
r_v!=old_v)
        {
          fprintf(stdout, "B: ");
          fprintf(stdout, "%1d", !!(cur_b&0x200)); // or check if ==0 or !=0 inside an IF statement...
          fprintf(stdout, "%1d", !!(cur_b&0x100));
          fprintf(stdout, "%1d", !!(cur_b&0x080));
          fprintf(stdout, "%1d", !!(cur_b&0x040));
          fprintf(stdout, "%1d", !!(cur_b&0x020));
          fprintf(stdout, "%1d", !!(cur_b&0x010));
          fprintf(stdout, "%1d", !!(cur_b&0x008));
          fprintf(stdout, "%1d", !!(cur_b&0x004));
          fprintf(stdout, "%1d", !!(cur_b&0x002));
          fprintf(stdout, "%1d", !!(cur_b&0x001));
          if (naxes>0) fprintf(stdout, " X: %7d", cur_x);
          if (naxes>1) fprintf(stdout, " Y: %7d", cur_y);
          if (naxes>2) fprintf(stdout, " Z: %7d", cur_z);
          if (naxes>3) fprintf(stdout, " R: %7d", cur_r);
          if (naxes>4) fprintf(stdout, " U: %7d", cur_u);
          if (naxes>5) fprintf(stdout, " V: %7d", cur_v);
          if (naxes>6) fprintf(stdout, " more skipped");
          if (cur_h<=36000)  fprintf(stdout, " POW:  %7d" , cur_h);
          fprintf(stdout, "\n");
          cnt=0;


                                    /* Send data to socket in order to request reply. */
                                    buf[0] = (cur_x+32767)>>8;
                                    buf[1] = (cur_y+32767)>>8;
                                    buf[2] = (cur_z+32767)>>8;;
                                    buf[3] = (cur_r+32767)>>8;;
                                    buf[4] = cur_b;
                                    m = 5;            /* send 5 bytes */
                                    n = sendto(sd,buf,m,0,(struct sockaddr*)&sad,alen);
                                    if(n<0)
                                    {
                                              fprintf(stderr,"Error in sending");
                                    }
                                    else
                                    {
                                              /* Read data from socket and write to user's screen.
```

```
                                                //n = recvfrom(sd,buf,sizeof(buf),0,(struct
sockaddr*)&sad,&alen);
                                                if (n > 0) {
                                                        buf[n]='\0';   /* just in case place the
termination at the end of the string*/
                                                //      printf("%s", buf);
                                                //}
                                        }

          }

          if (cur_b==15)
          {
            fprintf(stdout, "End program button sequence initiated\n");
            keeprunning=0;
          }

        }


      }
   }


   /* Close the socket. */
   closesocket(sd);

#ifdef WIN32
   WSACleanup();                /* release use of winsock.dll */
#endif

   /* Terminate the client program gracefully. */
   return(0);
}
```

Appendix D: Atmega 168 Code

Main.c

```
/*
 * Quadrocopter.c
 *
 * Created: 10/11/2012 8:17:23 AM
 *  Author: ebackman
 */

#include <avr/io.h>
#include <avr/interrupt.h>
#include "Serial.h"

char message_count;
uint16_t ReadADC(uint8_t channel)
{
	ADMUX |= channel;		// Channel selection
	ADCSRA |= _BV(ADSC);		// Start conversion
	while(!bit_is_set(ADCSRA,ADIF));  // Loop until conversion is complete
	ADCSRA |= _BV(ADIF);		// Clear ADIF by writing a 1 (this sets the value to 0)

	return(ADC);
}


void adc_init()
{
	ADCSRA = 0b01000111;  //Enable ADC and set 128 prescale
}

int count = 0;
int PWM[4];
int PWMTemp[4];
void MyCharReceivedFN (char c)
{
	// This function is called from within an interrupt
	// It must execute quickly - no loops or delays
	if(c == '/n'){
		message_count = 0;
	}
```

```
        if(message_count < 4)
                PWMTemp[message_count] = c;
        message_count++;
}



ISR(TIMER0_OVF_vect){
        count ++;
        if(count >= 1250){
                for(int i = 0; i<4; i++)
                        PWM[i] = PWMtemp[i];
                count = 0;
                PORTB |= 0x0F;
        }
        if(count == PWM[0]){
                PORTB &= 0xF7;
        }
        if(count == PWM[1]){
                PORTB &= 0xFA;
        }
        if(count == PWM[2]){
                PORTB &= 0xFC;
        }
        if(count == PWM[3]){
                PORTB &= 0xFE;
        }
}

int main(void)
{
        for(int i = 0; i<4; i++)
                PWM[i] = 0;
        char data[6];
        char channel = 0;
        adc_init();
        uart1_initialize(uart_bps_9600, MyCharReceivedFN);
        DDRB  = 0b11111111;   //DDRA for 128
        DDRD  = 0b00001100;
        TCCR0A |= 0; //Normal Mode
        TCCR0B = _BV(CS00); //No Prescaling
        //TCCR0 = 0x01; //for ATMEGA128
```

```
        TIMSK0 = 0x01;   //Enable timer overflow interrupt
        //                OCR0A = (unsigned char)(200);
        //sei(); //Global interrupt enable
  while(1)
  {
                //data[channel] = ReadADC(channel);
                //channel += channel;
                //if( channel > 5){
                //        channel = 0;
                        //uart1_putc(data[0]);
                //}
                uart1_putc(0x55);
    //TODO:: Please write your application code
  }
}
```

Serial.c

```
//* USART1 interrupt-based library - implementation file *

#include "Serial.h"
#include <avr/io.h>
#include <avr/interrupt.h>

#define xtal 20000000L

static void (*UART1_RX_ISR_function)(char) = 0;

void uart1_initialize( uint16_t baud, void (*handle_rx)(char) )
{
  uint32_t temp = xtal/16/baud-1;
  UBRR0H = (temp >> 8) & 0x0F;
  UBRR0L = (temp & 0xFF);

  UART1_RX_ISR_function = handle_rx;

  // Set frame format: 8data, 2stop bit
  UCSR0C = /*(1<<USBS0)|*/(3<<UCSZ00);

  UCSR0B = (1<<RXEN0)|(1<<TXEN0)|(1<<RXCIE0);
                //  ^^^^^^^^^^  data received interrupt enabled (when global interrupts are enabled)
}
```

```
void uart1_shutdown ()
{
   UCSR0B = 0;
}


uint8_t uart1_ready_TX ()
{
   return ( 0 != (UCSR0A & 1<<UDRE0)  );
}

void uart1_putc (char c)
{
   while( 0 == (UCSR0A & 1<<UDRE0)  ) ;
   UDR0 = c;
}

void uart1_puts (const char* const s)
{
   for ( const char* p = s; *p!='\0'; ++p )
   uart1_putc(*p);
}

ISR(USART0_RX_vect)
{
   char value = UDR0;                  // read UART register into value
   UART1_RX_ISR_function(value);     // call the _SHORT_ user defined function to handle it
}
```

Serial.h

```
//* USART1 interrupt-based library - header file *

#ifndef BIOS_UART1_INT_H_
#define BIOS_UART1_INT_H_
#include <stdint.h>


void    uart1_initialize   ( uint16_t baud, void (*handle_rx)(char) );
void    uart1_shutdown     ();
```

```
// This example shows only data item received interrupt
// Your function must be as short as possible, preferably without any loops

// This example will send data the classic way for simplicity
uint8_t uart1_ready_TX     ();
void    uart1_putc         (char c);
void    uart1_puts         (const char* const s);


#ifndef _SER_BOUD
#define _SER_BOUD
   #define uart_bps_50        50
   #define uart_bps_75        75
   #define uart_bps_110       110
   #define uart_bps_134       134
   #define uart_bps_150       150
   #define uart_bps_200       200
   #define uart_bps_300       300
   #define uart_bps_600       600
   #define uart_bps_1200      1200
   #define uart_bps_1800      1800
   #define uart_bps_2400      2400
   #define uart_bps_4800      4800
   #define uart_bps_9600      9600
   #define uart_bps_19200     19200
   #define uart_bps_38400     38400
   #define uart_bps_57600     57600
   #define uart_bps_115200    115200
#endif


#endif /* BIOS_UART_INT_H_ */
```