BG SYSTEMS, INC.

JFx Joystick

TCP Software Interface

Revision 1.00

June 21, 2011

Unrestricted Distribution

Table of Contents

1 Document Purpose

This document describes in detail the software interfaces for the BG Systems, Inc. JFx Josytick with TCP option.

1.1 Conventions

Throughout this ICD the following fonts are used to differentiate software code written on the host, button identification on the HCU, the state of the HCU, and any actions.

`Software code`	Example software for an XP host application appear in the monotype font.
BUTTON	Buttons are identified in the body of the text as all caps in this font.

2 Software

The JFx-tcp joysticks have an OB1K-ARM-12 microcontroller inside the joystick enclosure. The MCU has firmware which communnicates data to computers on a local area network by the TCP socket protocol.

This section contains code fragments to illustrate the communication protocol and certain commands that can be sent to the MCU.

```
// Example of a comment
```

Source code is provided and can be downloaded from `www.bgsystems.com`

2.1 Startup

On startup the initial IP address will be set to 192.168.1.10. The TCP packets are 64 bytes long.

The following sections shows startup from an XP application from `tcp_jf_main.cpp`

2.1.1 Initialization

The example code below is not complete, and there are fragments from header files. The intent is to demonstrate how to communicate with the MCU from a host software application.

For convenience a data structure is defined to contain pertinent information. The data structure is defined in `bg_tcp.h`

```
typedef struct BG_IP {
   char ip_name[24];                  // default host name
   SOCKET ip_socket;                  // regular socket
   struct sockaddr_in ip_addr;        // socket IP address
   char multicast_name[24];           // multicast group name
   SOCKET multicast_socket;           // multicast socket
   struct sockaddr_in multicast_addr; // multicast IP address
   struct ip_mreq bg_mreq;            // multicast data structure
   int multicast_ttl;                 // multicast time to live
   int multicast_port;                // multicast port
   int rate;                          // requested update rate
} bg_ip;
```

The fragment below is from the main application `tcp_jf_main.cpp`

```
WSADATA wsaData;  // Winsock2
bg_ip bgipdata;   // BG data structure for convenience

   int iResult = WSAStartup( MAKEWORD(2,2), &wsaData );

   if ( iResult != NO_ERROR )
         printf("Error at WSAStartup()\n");printf(".... WSAStartup() OK\n");
```

The next call is to convenience function `setup_arm_tcp()` which is in the `bg_basic.cpp` file. This sets up the BG data structure so that the incoming data packet is converted correctly. This data structure is common to various BG Systems firmware configurations and allows a standard conversion routine to be used. Details about the data structure are in `bg_arm2.h`

2.1.2 Open Sockets

The next code fragment calls utility functions to open sockets.

```
// Define ip name from header for two way socket communication
// pHostName = "192.168.1.10"
   sprintf(bgipdata.ip_name, "%s", pHostName);
   set_socket_data(&bgipdata.ip_addr, bgipdata.ip_name, DEFAULT_SOCKET_PORT);
// create socket for communication to MCU
   ret_val = create_socket();
// connect socket
   ret_val = connect_socket(bgipdata.ip_addr);
```

2.1.2.1 set_socket_data()

Convenience function to set the socket data structure with the host IP address and Port number. Note that this is for the two way communication socket and not for the Multicast socket.

```
void set_socket_data(sockaddr_in *pSockAddr,
         const char *pHostName, int portNumber)
{
   pSockAddr->sin_family = AF_INET;
   pSockAddr->sin_port = htons(portNumber);
   pSockAddr->sin_addr.s_addr = inet_addr( pHostName );
}
```

2.1.2.2 create_socket()

Convenience function to create a standard IP socket to communicate with the MCU. This socket allows data to be written to the MCU and information to be read from the MCU.

```
int create_socket()
{
   printf("Call socket\t");
   bgipdata.ip_socket = socket( AF_INET, SOCK_STREAM, IPPROTO_TCP );

   if ( bgipdata.ip_socket == INVALID_SOCKET )
   {
         printf( "Error at socket(): %ld\n", WSAGetLastError() );
         WSACleanup();
         return 0;
   }
   printf(".... socket() OK\n");
   return(1);
}
```

2.1.2.3 create_multicast_socket()

Convenience function to create a multicast socket for the host to listen to. This socket will receive data broadcast from the MCU. Only called when the MCU is set to multicast data, which is not the default.

```
int create_multicast_socket()
{
   printf("Call socket for multicast\t");

   bgipdata.multicast_socket = socket( AF_INET, SOCK_DGRAM, 0 );
   if ( bgipdata.multicast_socket == INVALID_SOCKET )
   {
         printf( "Error at socket(): %ld\n", WSAGetLastError() );
         WSACleanup();
         return(-1);
   }

   memset(&bgipdata.multicast_addr, 0, sizeof(bgipdata.multicast_addr) );
   bgipdata.multicast_addr.sin_family = AF_INET;
   bgipdata.multicast_addr.sin_addr.s_addr = htonl(INADDR_ANY);

   bgipdata.multicast_addr.sin_port = htons(bgipdata.multicast_port);
   printf("port %d  %d\n", bgipdata.multicast_addr.sin_port,
               bgipdata.multicast_port);
   if ( bind(bgipdata.multicast_socket,
               (struct sockaddr*)&bgipdata.multicast_addr,
                     sizeof(bgipdata.multicast_addr) ) < 0)
   {
         perror("bind failed");
         return(-1);
   }
   bgipdata.bg_mreq.imr_multiaddr.s_addr = inet_addr(MULTICAST_GROUP);
   bgipdata.bg_mreq.imr_interface.s_addr = htonl(INADDR_ANY);
   if ( setsockopt(bgipdata.multicast_socket, IPPROTO_IP, IP_ADD_MEMBERSHIP,
         (const char*)&bgipdata.bg_mreq, sizeof(bgipdata.bg_mreq) ) < 0)
   {
         perror("setsockopt failed");
         return(-1);
   }
   printf("Open multicast socket %d OK\n", bgipdata.multicast_socket);
   return(1);
}
```

2.3 Setup

The MCU can be configured in one of two ways:

1. Point a web browser to the IP address of 192.168.1.10 and set parameters using the form in the browser. Note that multicast socket options are not typically used, and the update rate only applies to the multicast option. Contact `john@bgsystems.com` if you have questions about this.

2. From a host application, parameters can be set. See section 3.7 for details.

2.4 Data Packet

The data packet from the multicast is retrieved as follows:

```
sprintf_s(out_buffer, 12, "e");
st = send_socket();

Sleep(1);
counter++;

if (st == 1 )
{
     st = read_socket();
     if ( st <= 0 )
     {
           printf("read_socket st %d\n", st);
           bgd.run = 2;
     }
     st = convert_tcp_arm(&bgdata, in_buffer);
}
```

The code above makes a call to `send_socket()` to request a packet of data. Then there is a `Sleep(1)` to allow data to be received from the MCU. A call to `read_socket()` actually retrieves the buffer with the data.

This data is then passed to a convenience function `convert_tcp_arm()` to decode the character string into the data structure containing analog values and switch values.

The data packet is 64 bytes for multicast, but data is only packed as follows:

```
str[0]      0x24      '$'   Start of data
str[1]      0xff      GP0 16-23
str[2]      0xff      GP0 8-15
str[3]      0xff      GP0 0-7
str[4]      0x0f00    A0 MSB
str[5]      0x00ff    A0 LSB
str[6]      0x0f00    A1 MSB
str[7]      0x00ff    A1 LSB
str[8]      0x0f00    A2 MSB
str[9]      0x00ff    A2 LSB
str[10]     0x0f00    A3 MSB
str[11]     0x00ff    A3 LSB
str[12]     0x0f00    A4 MSB
str[13]     0x00ff    A4 LSB
```

```
str[14]     0x0f00     A5 MSB
str[15]     0x00ff     A5 LSB
str[16]     0x0f00     A6 MSB
str[17]     0x00ff     A6 LSB
str[18]     0x0f00     A7 MSB
str[19]     0x00ff     A7 LSB
str[20]     0xff       Counter
str[21]     0xff       CRC
str[22]     0xff       CRC
str[23]     0x0a       '\n'  End of packet
```

2.5 Data Conversion

Data from the MCU is sent as a 64 byte multicast packet. The packet is read by `read_socket()` and returns a character buffer. This character buffer is passed to the `convert_tcp_arm()` function which scans the characters and puts them into the BG data structure. The conversion function is the generic BG Systems function. The example below shows a part of conversion routine.:

```
int convert_tcp_arm(bgarm2 *bgp, char *str)
{
   int i, digp, j;
   int k = 0;
   int st;
   int len = 0;

   digp = 0;
   i = 1;

   /*
    *  Load the GPIO 0 digital input values into din0
    */

   if (bgp->dig_in & GP0 && ( (bgp->dig_conf & GP0) != GP0))
   {
         for ( j = 2; j >= 0; j-- )
               bgp->gp0[j] = 0xff & str[i++];
   }

   /*
    *  Load the 8 12-bit analog values into inbuf
    */

   for (k = 0; k < 8; k++)
   {
         if ( bgp->analog_in12 & (0x1 << k ))
         {
               digp  = ((0x0f & str[i++]) << 8 );
               digp |= ( 0xff & str[i++]);
               bgp->ain12[k] = (100.0 * ((float)digp/4095.0));
               bgp->raw_ain12[k] = digp;
         }
   }
```

The example above would be the part of the routine used by a joystick to load buttons and analog values.

2.6 CRC Checking

In the convert_tcp_arm() function there is a CRC check to ensure that the data received is valid. The last two characters of the data string contain the CRC value which is also computed

```
// Perform CRC check

bgp->crc_error = 0;
len = bgp->str_len;

crc1 = ( ( (str[len-3]&0xff)
               | ((str[len-2]&0xff)<< 8)  )
               & 0xffff);

crc = 0xffff;

for ( i = 1; i <= (len-4); i++)
      crc = (crc >> 8) ^ crc_table[(crc ^ (str[i])) & 0xff];
crc = (~crc) & 0xffff;

if ( crc - crc1 != 0 )
{
      bgp->crc_error = 1;
      printf("crc error\n");
}
```

2.7 Configuration API

There are five states for the MCU that can be configured and there are two ways to configure these states. From a host computer a web browser can connect to the MCU which brings up a configuration page with text fields showing the current state of the configurable parameters: Update Rate (Rate), IP Address (IP), Multicast Address (MA), Multicast Port (MP), and Time to Live (TTL). To change a parameter, just enter a new value and then click on the button labeled "Update Settings".

Note that in general the multicast option is not used, and the key item that you may want to change is the IP address. If you have more than one joystick on a network they should have unique IP addresses. Of course once you have changed the IP address, you must make a note of the new IP address in order to communicate with the joystick. If you are using the API to set the IP address from within a program, then you will have to close the default socket and open a new socket with the new IP address.

The parameters can also be set from host software using an open socket (see section 2.1.2). Once the socket is open it can be used to configure the parameters as shown below. It is of course crucial that the socket is open to the correct IP address, so if you change the IP address of the MCU the socket will have to be closed and re-opened with the new IP address.

The following sections indicate how the parameters can be retrieved from the MCU and set to the MCU. In the sections below, the following global variables are used:

```
extern bg_ip bgipdata; // data structure defined in bg_tcp.h
extern char in_buffer[128];  // buffer for reading data from MCU
extern char out_buffer[12];  // buffer for sending data to MCU
```

Section 2.7.1 will explain in some detail how to handle socket communication with the MCU and the following sections will be limited to key details.

2.7.1 Update Rate

To retrieve the update rate, we simply make a call to get_rate() which contains the following code:

```
void get_rate()
{
     int st;

     sprintf_s(out_buffer, 12, "%c%c%c", COMMAND, READ, RATE);
     st = send_socket();

     if ( st < 0 )
           printf("error sending socket\n");

     st = read_socket();
     if ( st < 0 )
           printf("error reading socket\n");

     bgipdata.rate = (int)in_buffer[0];
}
```

In the code above, the out_buffer is set with three characters:

```
COMMAND
READ
RATE
```

These are defined in bg_tcp.h. COMMAND indicates that the MCU is receiving a command from the host, READ means that the host wants to read data from the MCU, and RATE is the parameter to be read. This information is sent to the MCU with the send_socket() convenience function. If there is no error, then we can read_socket() which returns data back in the in_buffer[]. In this case the rate is an integer returned in the first character of the buffer.

To set the update rate:

```
void set_rate()
{
     int st;
     char txt[4];
     int i = 0;

     for ( i = 0; i < 4; i++ )
           txt[i] = 0x0;

     txt[0] = bgipdata.rate;  // Must be 10 or 100

     sprintf_s(out_buffer, 12, "%c%c%c%c", COMMAND, WRITE, RATE, txt[0]);
     st = send_socket();

     if ( st < 0 )
           printf("error sending socket\n");
}
```

In the code above, a character buffer is set to null, and then the first character is set with the bgipdata.rate data member. This should be set by the host software to either 10 or 100 for the update rate in

Hz. Then the out_buffer is set with four characters:

```
COMMAND
WRITE
RATE
txt[0]
```

As before, the COMMAND tells the MCU that it is receiving a command from the host, WRITE indicates that the host is sending information, RATE indicates the parameter being set, and txt[0] is the value being sent.

The convenience function send_socket() is then called to actually send the data.

2.7.2 IP Address

To retrieve the current IP address of the MCU we call

```
void get_ip()
{
     int st;
     struct in_addr in;
     char *ip_buf;

     sprintf_s(out_buffer, 12, "%c%c%c", COMMAND, READ, HOSTIP);
     st = send_socket();
     if ( st < 0 )
          printf("error sending socket\n");

     st = read_socket();
     if ( st < 0 )
          printf("error reading socket\n");
     in.S_un.S_un_b.s_b1 = in_buffer[3] & 0xff;
     in.S_un.S_un_b.s_b2 = in_buffer[2] & 0xff;
     in.S_un.S_un_b.s_b3 = in_buffer[1] & 0xff;
     in.S_un.S_un_b.s_b4 = in_buffer[0] & 0xff;
     ip_buf = inet_ntoa(in);
     bgipdata.ip_addr.sin_addr.s_addr = inet_addr(ip_buf);
}
```

This function is similar to get_rate() but the data retrieved needs manipulation to get it into a useable format.

The command itself is straightforward with HOSTIP being the parameter to be READ. The data returned is in four bytes and they need to be put into the in_addr data structure which is then passed to the system call inet_ntoa to convert to a string which can be passed to inet_addr() which can be used to fill the bgipdata member.

To set the IP address:

```
void set_ip()
{
     int st;
     char txt[4];
     int i = 0;

     for ( i = 0; i < 4; i++ )
          txt[i] = 0x0;

     txt[0] = bgipdata.ip_addr.sin_addr.S_un.S_un_b.s_b4;
     txt[1] = bgipdata.ip_addr.sin_addr.S_un.S_un_b.s_b3;
     txt[2] = bgipdata.ip_addr.sin_addr.S_un.S_un_b.s_b2;
```

```
        txt[3] = bgipdata.ip_addr.sin_addr.S_un.S_un_b.s_b1;

        sprintf_s(out_buffer, 12, "%c%c%c%c%c%c%c", COMMAND, WRITE, HOSTIP,
              txt[0], txt[1], txt[2], txt[3]);
        st = send_socket();

        if ( st < 0 )
              printf("error sending socket\n");
}
```

In this case the data to be sent is put into four bytes of the txt[] array as shown above.

2.7.3 Multicast Address

Reading and setting the multicast address is done in the same manner as the IP address. The code fragments below are not complete, but contain the important information:

```
void get_multicast_ip()
{
...
     sprintf_s(out_buffer, 12, "%c%c%c", COMMAND, READ, MULTIP);
     st = send_socket();
...
     st = read_socket();
...
     in.S_un.S_un_b.s_b1 = in_buffer[3] & 0xff;
     in.S_un.S_un_b.s_b2 = in_buffer[2] & 0xff;
     in.S_un.S_un_b.s_b3 = in_buffer[1] & 0xff;
     in.S_un.S_un_b.s_b4 = in_buffer[0] & 0xff;

     ip_buf = inet_ntoa(in);
     bgipdata.multicast_addr.sin_addr.s_addr = inet_addr(ip_buf);
}

void set_multicast_ip()
{
....
     txt[0] = bgipdata.multicast_addr.sin_addr.S_un.S_un_b.s_b4;
     txt[1] = bgipdata.multicast_addr.sin_addr.S_un.S_un_b.s_b3;
     txt[2] = bgipdata.multicast_addr.sin_addr.S_un.S_un_b.s_b2;
     txt[3] = bgipdata.multicast_addr.sin_addr.S_un.S_un_b.s_b1;

     sprintf_s(out_buffer, 12, "%c%c%c%c%c%c%c", COMMAND, WRITE, MULTIP,
           txt[0], txt[1], txt[2], txt[3]);

     st = send_socket();
....
}
```

2.7.4 Multicast Port

To read the multicast port:

```
void get_multicast_port()
{
...
     sprintf_s(out_buffer, 12, "%c%c%c", COMMAND, READ, MULTPORT);
     st = send_socket();
...
     st = read_socket();
...
     bgipdata.multicast_port = (in_buffer[1] & 0xff) << 8
                                     | (in_buffer[0] & 0xff);
}
```

To set the multicast port:

```
void set_multicast_port()
{
...
     txt[0] = (bgipdata.multicast_port & 0xff);
     txt[1] = (bgipdata.multicast_port & 0xff00) >> 8;

     sprintf_s(out_buffer, 12, "%c%c%c%c%c", COMMAND, WRITE, MULTPORT,
                                              txt[0], txt[1]);

     st = send_socket();
...
}
```

2.7.5 TTL

To read TTL:

```
void get_multicast_ttl()
{
...
     sprintf_s(out_buffer, 12, "%c%c%c", COMMAND, READ, TTL);
     st = send_socket();
...
     st = read_socket();
...
     bgipdata.multicast_ttl = in_buffer[0];
}
```

To set TTL:

```
void set_multicast_ttl()
{
...
     txt[0] = bgipdata.multicast_ttl;
     sprintf_s(out_buffer, 12, "%c%c%c%c", COMMAND, WRITE, TTL, txt[0]);
     st = send_socket();
...
}
```