AN-3: Ant Server Interface

1. Summary

This application note describes and defines the RockyLogic *antserve.exe* server program. The server accepts text commands over a TCP link, controls one or more Ant8 or Ant16 modules attached to local USB ports, and sends responses back over the TCP connection.

2. Introduction

The initial Ant8 and Ant16 support software was a monolithic application, making it difficult or impossible to run the program across a network. It was also difficult to open up the software to allow user programmability. The new generation of software is implemented as three layers:

a. the top (GUI) layer, which interfaces to

b. the Ant interface DLL. This deals with the low-level *register bashing* control of the Ant8 and Ant16 and interfaces across a TCP link to

c. the *antserve* server program, the subject of this application note.

Each server thread listens for a command, performs the requested task, and sends back a response. All commands, responses, and parameters are plain text.

3. Commands and Responses

Commands are single lower-case letters and responses start with the same letter in upper case. The command format depends on whether the command has no parameters, a fixed-length parameter, or a variable length parameter:

```
x          // command with no parameters
xpar       // fixed-length parameter encoded in par
xpar;      // variable-length parameter encoded in par and terminated by a ';'
```

The response format depends on whether or not the response has return data, and whether or not the response has an error code.

```
X;         // no return data, no error code
X$err;     // no return data, error encoded in err
Xret;      // return data encoded in ret, no error code
Xret$err;  // return data encoded in ret, error encoded in err
```

3.1 Encoding

Numeric values, are encoded byte by byte as hexadecimal text strings. Numeric values larger than a single byte are sent least significant byte first. The meaning of a byte is determined by context. For example:

```
10          // one byte, 10h (decimal 16)

0102        // as single bytes: 01h followed by 02h
            // as a 16-bit value: 0201h (decimal 513)

01020304    // as single bytes: 01h followed by 02h, 03h, 04h
            // as 16-bit values: 0201h (decimal 513) followed by 0403h (decimal 1027)
            // as a 32-bit value: 04030201h (decimal 67305985)
```

Character strings are sent with no encoding. As with numeric values, the meaning of a byte is determined by context. The character values ';' and '$' are escaped with a '\'.

3.2 Errors and Exceptions

The response to each command is terminated with a semi-colon. Successful execution of a command does not return an error code. An error code in a response is flagged with a leading $. Error codes are grouped as follows:

D – USB driver error

Given that USB devices can be unplugged at any time, applications should always guard against receipt of a driver error. The error response is of the form *Dab* where

```
D   // indicates a driver class error

a   // one byte indicating the driver operation which caused the error
    //      1       ListDevices
    //      2       OpenEx
    //      3       Close
    //      4       Purge
    //      5       SetTimeouts
    //      6       Read
    //      7       Write

b   // one byte error code returned by the driver
```

A – Ant access protocol error

This error code covers such events as closing a device that is not open. The error response is of the form *Ae* where

```
A   // indicates an access protocol class error

e   // one byte error code
    //      1       no module open
    //      2       module open
    //      3       open failed - module name unknown
    //      4       open failed - module now unplugged
    //      5       open failed - module in use
    //      6       FPGA load failed - not a known Ant type
    //      7       FPGA load failed - bad configuration index
    //      8       FPGA load failed - other reasons
```

P – parameter error

This error code covers such events as receiving an Open command with no device name specified or the device name too short. The error response is of the form *Pe* where

P // indicates a parameter class error

e // one byte indicating the operation which caused the error
// 1 module name length too short. Should be 8 chars.

M – memory allocation exception

Occurrence of this error code reflects a serious problem in the heart of the server. The error response is *M*.

X – program error

This is a catch-all code reflecting an unclassified serious problem in the heart of the server. The error response is *X*.

3.2 List of Commands

command: q – who are you

response: Qstr;

comments: str contains the program name, the creation date, and the date/time at which this instance of the server started running in the following format:

antserve,creation_date,run_start_date,run_start_time;

example: q
Qantserve,Nov 12 2005,Nov 22 2005,12:48:12;

.....

command: a – maximize window

response: A;

comments: this is a debugging command which gives a visual feedback that connection has been made to the server program. Inside the server it sets a flag which is inspected and cleared by a timer every second. Thus there is a maximum of one maximize or minimize per second.

example: a
A;

.....

command: b – minimize window

response: B;

comments: this is a debugging command – see the description of *a*. Maximize has priority over minimize.

example: b
B;

.....

command: l – list modules

response: Lstr;

comments: str contains a list of the ant modules attached in the following format:

RLccccccctxy...RLccccccctxy

RLcccccc is the eight character module ID, which always starts with RL
t is the Ant model indicator (r=Ant8, o=Ant16)
x is the in-use indicator (b=busy, f=free)
y is the present indicator (p=present, m=was present, now missing)

example:
```
l
LRL3ID7A3ofp;                    // one free Ant16

l
LRL3ID7A3ofpRL2NRS51rbp;         // one free Ant16, plus one busy Ant8
```

……

command: o – open module

format: ostr

response: O;

comments: str is the Ant module to be opened in the following format:

RLcccccc

where RLcccccc is the eight character module ID, which always starts with RL

example:
```
oRL3ID7A3
O;                               // success
```

……

command: c – close module

response: C;

comments:

example:
```
c
C;                               // success
```

……

command: p – purge the USB buffers

format: p

response: P;

comments: removes any pending characters from the USB input and output buffers

example:
```
p
P;                               // success
```

……

command: w – write bytes

format: wbytestr;

response: W;

comments: bytestr contains a sequence of byte values to be sent to the Ant module, terminated by a ';' character. Each byte is a two-character hexadecimal value. For instance 0a3e2d8f is a sequence of four values: 0a, 3e, 2d, 8f. The bytes are written to the Ant exactly as received.

.

example:
```
w01020304;                    // write the four bytes 01h, 02h, 03h, 04h
W;
```

……

command: r – read bytes

format: rcount

response: Rbytestr;

comments: *count* is the 4-byte encoded count of bytes to be read.

bytestr is the stream of bytes read from the Ant.

example:
```
r03000000                     // read three bytes
R606162;                      // byte values are 60h, 61h, 62h
```

……

command: f – load a configuration

format: fix

response: F;

comments: ix is the 1-byte index of the configuration to be loaded into the Ant:

1	monitor the probe wires
2	X4 – 4 samples per clock
3	X2 – 2 samples per clock
4	X1 – 1 samples per clock
5	SLOW – sample at xx MHz or less
6	SYNC – synchronous sampling (not the Ant8)

example:
```
f01                           // load the monitor configuration
F;                            // succcess
```

……

command: t – flush the TCP buffer

format: t

response: T;

comments: Error responses from the server force an output buffer flush. All other messages are subject to standard TCP output buffering. This command forces a flush of the output buffer.

example:
```
t
T;                            // response buffer in the server is now flushed
```

4. Running antserve.exe

The command line for the program is

```
antserve port=portnum
```

portnum: TCP port number. The default is 8279.

The program starts, installs itself minimized on the taskbar, scans for Ants on the USB ports, and listens for incoming connections on the designated TCP port. It rescans the USB ports for Ants every second when maximized.

Only one copy of antserve.exe can be running on each machine – the program silently terminates itself if a running server is detected.

Double clicking the icon on the taskbar will bring up a window which displays the currently installed Ants and their status.

You can verify the TCP connection to the server by starting up telnet (from a command prompt), connecting to the server and executing simple commands such as q or l.

5. Ant Registers

The Ant8 and the Ant16 are implemented with a combination of fixed logic and configurable logic. This section describes the registers in the *fixed* logic.

The register interface is a stream of 8-bit bytes over a USB link. A *read* operation returns an 8-bit data value from a register. A *write* operation sends a 6-bit data value to a 2-bit address, with the following encoding:

bits 7..6	***bits 5..0***
00	Register address. There are 64 possible addresses, of which addresses 0 to 15 are in the fixed logic and addresses 16 to 63 are available for the configurable logic.
01	Register data. Write this value to the currently addressed data register.
10	Read count. Read this number of bytes from the currently addressed register and send them over the USB link. The count can range from 1 to 64 - a value of 0 is interpreted as 64.
11	reserved

To write N bytes to register R

1. write R to the address register
2. write N bytes to the data register

To read N bytes from register R

1. write R to the address register

2. write N to the read count register, where N is 64 or less (see ablove)
3. read N bytes from the device

The write FIFO is 128 bytes deep, so for maximum throughput several read counts can be written, followed by a single large read. For instance send 16 read counts of 64, then do a single read of 1024 bytes.

5.1 Fixed Registers

These are the fixed registers in the Ant8 and the Ant16. When writing to registers, the value written to an unused register bit should be 0. When reading registers, and the value of an unassigned bit is unpredictable.

.....

register 0 – ID register

write	bits 5 to 0	unused
read	bits 7 to 0	A single read from this register will return the ID code of the Ant unit. 72h: this is an Ant8 6Fh: this is an Ant16 A multiple read returns a reverse alphabetic character string, terminating in the ID code of the Ant unit. For instance, *read 5* from an Ant8 returns this hexadecimal string: 64h 63h 62h 61h 72h // *DCBAr* in ASCII and *read 4* from an Ant16 returns this hexadecimal string: 63h 62h 61h 6Fh // *CBAo* in ASCII

.....

register 1 – configurable logic core-voltage control

write	bit 0	0: FPGA core voltage off 1: FPGA core voltage on
	bits 5 to 1	unused
read	bit 0	FPGA core voltage
	bits 7 to 1	unassigned

.....

register 2 – FPGA Control and Status

write	bit 0	0: drive FPGA PROGn signal low 1: drive FPGA PROGn signal high
	bits 5 to 1	unused
read	bit 0	FPGA PROGn signal
	bit 1	FPGA INITn signal
	bit 2	FPGA DONE signal
	bits 7 to 3	unassigned

……

register 3 – FPGA configuration data

write	bits 5 to 0	FPGA configuration data, shifted into the FPGA MSB first.
read	bits 7 to 0	unassigned