Win500 Remote Control & Monitoring Protocol

1. **Data types**
   a. **char**: 8-bit unsigned value, 0x00-0xFF
   b. **short**: 16-bit unsigned value, 0x0000-0xFFFF
   c. **long:** 32-bit unsigned value, 0x00000000-0xFFFFFFFF
   d. All multi-byte values are sent "little-endian" (least-significant byte sent first)

2. **Connection to server**
   a. TCP connection to IP : Port specified in Win500 server
   b. Data sent as "stream of bytes" (packetization is above the transport layer)

3. **Packet structure**
   a. char: **Command:** value from list of commands below
   b. short: **Length:** *total* size of packet, in bytes
   c. long: **Timestamp:** sender's "millisecond timer", for syncing received packets to real time at receiver
   d. char[]: **Payload:** <Command>-dependent bytes
   e. short: **Checksum:** sum of all previous bytes (starting with <Command>)

4. **Commands – Win500 to Remote Client**
   a. **'S' (0x53) : Status**
   Payload length: 16
   Payload[0] = current scanner mode:
      0: Tune
      1: Scan
      2: Manual
      3: Program
      4: Search
      5: Weather
      9: Sweeper/Stalker
   Payload[1] = Flags (bitmap)
      bit 0: squelch open
      bit 1: audio unmuted
      bit 2: XF flag
      bit 3: HD2 flag
      bit 4: HD5 flag
      bit 5: set if a "mobile/base" unit
      bit 6: set if backlight is in "dim" mode
   Payload[2,3] = **short** indicating battery level A/D value. ORed with 0x8000 if on external power
   Payload[4,5] = **short** indicating RSSI value
   Payload[6,7] = **short** indicating "zeromatic" value
   Payload[8] = RED LED value
   Payload[9] = GRN LED value

Payload[10] = BLU LED value
Payload[11,14] = currently-tuned frequency, in Hz
Payload[15] = current rx mode: AM=0, FM=1, NFM=2

b. ‘L’ (0x4C) : LCD Data – full

Note: Win500 also supports connections to the PSR-310/410 scanners, which have a 5th line of LCD text. If Win500 is connected to such a scanner, then the payload will be 82 bytes long: 80 bytes of LCD data, with the icons at offsets 80 and 81. It is up to the client app to look at the length field, verify that it indicates a payload of 66 or 82, then decide how much LCD data is present and how to display it. The information below is for a non-310/410 scanner.

Payload length : 66
Payload[0,63] = LCD text, top-left to bottom-right
Payload[64] = LCD Icons (bitmap)
- bits 0-2: signal bars, 0-5
- bit 3: ‘S’
- bits 4-5: battery

Payload[65] = LCD Icons (bitmap)
- bit 0: ‘F’
- bit 1: ‘G’
- bit 2: ‘A’
- bit 3: ‘T’
- bit 4: <up arrow>
- bit 5: <down arrow>
- bit 6: backlight on

c. ‘A’ (0x41) : Audio samples, compressed

Payload length : variable, must be deduced from packet’s **Length** value. Should be verified against ‘dwBufferLength’ field in the WAVEHDR structure.
Payload[] : audio sample data

d. ‘a’ (0x61) : Audio samples, uncompressed

Payload length : variable, must be deduced from packet’s **Length** value. Should be verified against ‘dwBufferLength’ field in the WAVEHDR structure.
Payload[] : audio sample data

e. ‘C’ (0x43) : Server information

Payload length: 5
Payload[0] = Flags (bitmap)
- bit 0: Win500 server allows control of scanner

Payload[1] = Win500’s remote protocol version
Payload[2] – Payload[3] = <reserved>

Payload[4] = 'D' if server instructing client to close the connection

f. **'l' (0x6C) : Log Message**
Payload length: variable
Payload[] = string data, or special flags

This represents text strings or control messages for the log message window as seen in the Win500 Remote Client Windows application. On a small, remote device, it may not be desired to handle these messages.

If the payload length is 1, then the payload consists of a single byte indicating a control flag. At present, the only valid control flag value is 0xFF. This tells the client to erase the last line of the log message window (generally, because a new string will be coming to replace it, e.g. when a new CTCSS value has been detected on a "searching" channel).

If payload length is greater than 1, then the payload is a <nul>-terminated ASCII string of text for the log message window. The client should verify that the last byte of the payload, as determined by the payload length, is really 0x00.

g. **'t' (0x74) : "Chat" message**
Payload length: variable
Payload[] = string data

This represents a text string sent via Win500's "Chat Send" function. It's a <nul>-terminated string and should be validated in the same way as the 'l' command above. On a small, remote device, it may not be desired to handle these messages.

5. Commands – Remote Client to Win500

a. **'t' (0x74) : "Chat" message**
Payload length: variable
Payload[] = string data

This represents a <nul>-terminated ASCII text string sent to Win500, for display in its log message window. A small, remote device may not wish to generate this message.

b. **'c' (0x63) : Info request**
Payload length: 0

This asks Win500 to send the "Server Information" message above (command 'C', 0x43). The client should not sit and wait for a response; it should continue normal operation and handle the response if/when it is received.

c. **‘k’ (0x6B) : Key press data**
Payload length: 1
Payload[0] = key code

This is a key press that Win500 will forward to the connected scanner. The key values can be found in the PSR-500 manual, in Appendix A “Remote Control Protocol”.

6. **Audio sample format**
All audio samples correspond to Microsoft’s WAV format, 8000 samples per second, 8 bits per sample, monaural. Uncompressed data is plain PCM, while compressed data is GSM6.10. Each Payload[] starts with a Microsoft **WAVEHDR** structure; the remainder of the payload is the sample data. The “pointer” members of the **WAVEHDR** structure are not valid.

It is presumed that the reader is familiar with whatever coding is required to “play audio” on his target platform / environment, so details are not specified here. However, on the Windows Mobile (“Pocket PC”) platform, I used the various *waveOut* APIs, such as *waveOutOpen, waveOutSetVolume, waveOutPrepareHeader, waveOutWrite, waveOutUnprepareHeader, waveOutReset,* and *waveOutClose*.

7. **Recommended processing/parsing of received data**
Since Win500 sends its data over the stream-oriented TCP transport, the client will not receive “packets” from the transport. Instead, TCP delivers a stream of bytes. The client must parse the received stream, looking for command bytes, extracting (and validating) the “length” field, then validating the checksum. Only after completely validating a received packet should the client attempt to act on that data. This is especially true if there is some active TCP agent (like my TCPMux program) between Win500 and the client app.

Here is some pseudo-code describing the recommended method:

```
// Some function that tells us the socket is still connected to the host
int SocketConnected( void );

// Some function that reads data from a socket, returning the number
// of bytes read. Places the data at <buffer>, only reading up to
// <availableSpace>
int SocketReceive( unsigned char *buffer, int availableSpace );

// Some function that processes valid received packets.
void ProcessPacket( unsigned char Cmd, unsigned char *Payload, int PayloadLen );

// the standard C "memmove()" function
void *memmove( void *dest, const void *src, size_t count );


const int RxBufferSize = 20000;
unsigned char RxBuffer[ RxBufferSize ];
```

```
int RxLength;

RxLength = 0;

while ( SocketConnected() )
    {
    // try to get some data from the socket
    int bytes = SocketReceive( RxBuffer + RxLength, RxBufferSize - RxLength );

    if ( bytes )    // got some data
        {
        RxLength += bytes;

        int done = 0;
        int needBufferShift = 0;

        do
            {
            // if some 'valid' check below failed, throw away the first received
            // byte
            if ( needBufferShift )
                {
                memmove( RxBuffer + 0, RxBuffer + 1, RxLength - 1 );
                RxLength--;
                needBufferShift = 0;
                }

            // make sure we have the minimum byte count for a packet
            if ( RxLength < 9 )
                {
                done = 1;
                continue;
                }

            // get the "command" byte
            unsigned char cmd = RxBuffer[ 0 ];

            // Make sure command byte is valid. Also take this opportunity to get
            // an "expected packet length"
            int expectedSize = 0, expectedSize310 = 0;
            switch ( cmd )
                {
                // invalid command byte. Toss it and keep trying
                default:
                    needBufferShift = 1;
                    continue;

                // fixed-size packets
                case 'S':
                    expectedSize = 25;
                    break;
                case 'L':
                    expectedSize = 75;
                    expectedSize310 = 91;
                    break;
                case 'C':
                    expectedSize = 14;
                    break;

                // variable-length packets... just leave the expected sizes == 0
                case 'A':
                case 'a':
                case 'l':
                case 't':
                    break;
                }
            // Note that in the above, we handled ALL possible commands, even those
            // we may not be interested in (e.g. 't' and 'l'). We do this so that
            // we can more easily sync to the actual packet boundaries



            // Now extract the total packet length
            int packetLen = (int)RxBuffer[1] + (int)RxBuffer[2] * 256;

            // If the indicated length is something ridiculous, then the packet
            // can't possibly be valid (or we're not yet synced)
            if ( packetLen < 9 || packetLen > 20000 )
                {
                needBufferShift = 1;
                continue;
                }

            // For the fixed-size packets, make sure it matches what we expect. If
            // it doesn't, toss the first byte as we did above and continue
            if ( expectedSize && packetLen != expectedSize &&
                 expectedSize310 && packetLen != expectedSize310 )
                {
                needBufferShift = 1;
                continue;
                }

            // look at the variable-length packets, and make sure <packetLen> looks
            // good...

            // Ah - but before we look at those packets, we have to make sure we
            // have enough data (we're going to be looking into the packet!).
            // If we don't have enough bytes, wait for more
            if ( RxLength < packetLen )
                {
                done = 1;
                continue;
                }

            // OK, now we'll go look at the variable-length packets

            // first, the audio packets
            if ( cmd == 'A' || cmd == 'a' )
                {
                LPWAVEHDR pWaveHdr = (LPWAVEHDR)(RxBuffer + 7);
                expectedSize = 9 + sizeof(WAVEHDR) + pWaveHdr->dwBufferLength;
                if ( packetLen != expectedSize )
                    {
                    needBufferShift = 1;
```

```
          continue;
          }
      }

  // now the log message packet
  if ( cmd == 'l' )
      {
      // make sure it's either a single 0xFF byte or a <nul>-terminated
      // string
      if ( (packetLen == 10 && RxBuffer[7] != 0xFF) ||
           (packetLen != 10 && RxBuffer[packetLen-3] != 0x00) )
          {
          needBufferShift = 1;
          continue;
          }
      }

  // check for chat text. Make sure it's null-terminated
  if ( cmd == 't' )
      {
      if ( RxBuffer[packetLen-3] != 0x00 )
          {
          needBufferShift = 1;
          continue;
          }
      }

  // calculate checksum on rx data
  unsigned short rs;
  int i;
  for ( i=0, rs=0; i<packetLen-2; rs += RxBuffer[i++] );
  // extract received checksum
  unsigned short s = (unsigned short)RxBuffer[packetLen-2] +
                     (unsigned short)RxBuffer[packetLen-1] * 256;

  // if calculated checksum doesn't match the received, toss first byte
  // and keep trying
  if ( s != rs )
      {
      needBufferShift = 1;
      continue;
      }



  // Wheeee! We have a complete, valid packet. Go do something (or
  // nothing) with it...
  ProcessPacket( cmd, RxBuffer + 7, packetLen - 9 );

  // And remove the packet from the receive buffer
  if ( RxLength == packetLen )
      // shortcut... buffer had exactly one packet, so just zero the
      // length
      RxLength = 0;
  else
      {
      memmove( RxBuffer, RxBuffer + packetLen, RxLength - packetLen );
```

```
            RxLength -= packetLen;
            }


        // keep looping as long as we have a possible candidate packet
        } while ( ! done );
    }
}
```