*your position is our focus*

# Application Note

| Topic : | **Implementing AssistNow® Online Client for u-blox GPS Receivers** |
|---|---|
| | GPS.G4-SW-05017-C |
| **Author :** | DA |
| **Date :** | 27. Oct 2005 (Rev. C: 20. Dec 2006, GzB) |
| **To :** | |

# 1 Introduction

u-blox maintains a world-wide GPS monitoring network, collects measurements from the base stations, and distributes information derived from these measurements through an Internet-connected Server such as the **AssistNow Online Root Server** or the **AssistNow Online Proxy Server**.

This distribution of information allows ANTARIS and ANTARIS 4 GPS receivers to speed up time to first fix (TTFF) in a number of applications where the receiver has a hard time collecting measurements autonomously, and where an Internet connection is feasible.

u-blox provides access to a demo A-GPS setup which allows authorized users to connect to the u-blox AssistNow Online Root Server. This setup supports customers in evaluating the performance benefits of implementing ANTARIS and ANTARIS 4 based receivers with assistance data support.

This document describes the protocol being used between the server and the clients and gives a reference implementation of a simple client.

# 2 Overview

## 2.1 Setup

The following diagram shows the overall system setup. This document describes the functions which shall be implemented in [4], and the protocol being spoken through [3].

```
  ________________________     ________________________
 /                        \   /                        \
 | GPS data submitter 1 |   | GPS data submitter 2 |   ...  [1]
 \________________________/   \________________________/
             |                            |
             V                            V
         ____|____________________________|______________
        /                                                \
        |            server                              |     [2]
        \________________________________________________/
             |             |                     |
             V             V                     V             [3]
    _________|_       _____|______          ___|_______
   /           \     /            \        /           \
   | client 1 |     | client 2 |   . . .  | client n |        [4]
   \___________/     \____________/        \___________/
    _____V______      _____V______          _____V______
   /            \    /            \        /            \
   |ANTARIS RX|      |ANTARIS RX|          |ANTARIS RX|        [5]
   \____________/    \____________/        \____________/
```

## 2.2 Submitter Function

Submitters [1] are reference GPS receivers located worldwide. These submitters monitor the GPS constellation continuously and submit their measurements to a central server.

At the time of writing, the setup contains two submitter stations, located in US and Europe. These stations allow a full coverage of Europe and the US, and western Asia. A third submitter station is being setup, increasing coverage in the western Pacific region.

## 2.3 Server Function

The server [2] collects measurements from its slave. The server listens on a specific TCP/IP port where assistance clients submit requests. The server answers these requests, returning whatever information the client requested and the server has available.

The server requires that clients authenticate using a username/password combo. It also requires that the client transmits an approximate position of its own location, since the information sent by the server is localized to the clients location, in order to save bandwidth.

## 2.4 Client Function

The Clients' [4] function is to connect to the server, request the information, and evaluate the results. In case of a successful response, the client should then transmit the information received (without the status headers) to an Antaris receiver [5] through its physical connection (USB, UART), possibly relayed through some networks.

## 2.5 Supported receivers

The following ANTARIS receivers are prepared for processing the information delivered by the server

- All ANTARIS 1 receivers, running firmware versions 3.01, 3.04 or 3.10
- All ANTARIS 4 receivers, running firmware versions 4.00 or 4.10

# 3 Detailed Description of the protocol

This section describes the protocol between the u-blox server, and the client requesting information.

The information exchange is loosely built around the HTTP protocol stack. Upon reception of an URL-like request, the server will respond with an HTTP-style header and content. After delivery of all data, the server will terminate the connection.

## 3.1 Request format.

The request is sent from the client to the server, immediately after establishing the socket connection.

A request is sent in plain ASCII, and has the following form

```
key=value;key=value;key=value;…\n
```

The following rules apply:

- The order of keys is not important.
- Keys and values are case sensitive.
- Key/value pairs must be separated by semicolons
- The request must be terminated with a newline character

The following keys are supported

| Key Name | Unit/Range | Mandatory/Optional | Comment |
|---|---|---|---|
| cmd | String | Mandatory | This key determines what kind of information the client requests from the server.<br>- "full" delivers Ephemeris and Almanac data and Approximate Time and Position to the client<br>- "aid" is identical to "full", but does not deliver Almanac<br>- "eph" only delivers Ephemeris which is of use to the client at its current location<br>- "alm" delivers Almanac data for the full GPS constellation |
| user | String | Mandatory | The username, for example "`foo@bar.com`". This must be a valid Email Address, as important server maintenance messages will be sent to this address |
| pwd | String | Mandatory | The password. This field is transmitted in clear text, so it shall not be a safety critical password. Passwords are assigned by u- |

| | | | |
|---|---|---|---|
| | | | blox and are only valid for a given Email address. |
| lat<br>lon | Numeric [degrees] | "Half"-Mandatory<br>Either lat/lon or ex/ey/ez is mandatory | Approximate user position in WGS-84 Latitude / Longitude in units of degrees and fractional degrees.<br>Example:<br>`"lat=47.2;lon=8.55"`. |
| alt | Numeric [meters] | Optional | Approximate user altitude above WGS84 Ellipsoid in units of meters. |
| ex<br>ey<br>ez | Numeric [meters] | "Half"-Mandatory<br>Either lat/lon or ex/ey/ez is mandatory | Approximate user position in ECEF Frame in units of meters<br>Example:<br>`"ex=4286464.77;ey=645609.71;ez=4663731"`. |
| pacc | Numeric [meters] | Optional | Approximate accuracy of submitted position (either lat/lon or ex/ey/ez). If this value is not provided, the server assumes an accuracy of 300km. |
| latency | Numeric [seconds] | Optional | Typical latency between the time the server receives the request, and the time when the assistance data arrives at the GPS receiver.<br>The server uses this value to correct the time being transmitted to the client (if cmd=full or cmd=aid).<br>Example:<br>`"latency=0.27"`. |

Example:

```
cmd=aid;user=foo@bar.com;pwd=whatever;lat=47.28;lon=8.56;pacc=1000\n
```

### 3.1.1 Requirements for the position and time parameters

The position that is being sent to the server is being used for two purposes:

- The server determines the currently visible satellites at the user position, and only sends ephemeris data of those satellites which should be in view at the location of the user. This reduces bandwidth requirements
- The server also feeds back the position and optional accuracy to the user's ANTARIS receiver. Depending on the accuracy of the provided data, the receiver can then choose to select a better startup strategy. For example, if the position is accurate to 100km or better, the ANTARIS receiver will choose to go for a more optimistic startup. This will result in quicker startup time. The receiver will decide which strategy to choose, depending on the 'pacc' parameter. If the submitted user position is less accurate than what is being specified with the 'pacc' parameter, then the user will experience prolonged or even failed startups.

The time that is being sent to the receiver is the time when the request arrived at the server, corrected by the optional 'latency' value. The client needs to ensure, that the nominal latency between receiving the data from the server, and the arrival of the data at the receiver is below 1 second (plus what is being optionally set in the latency value). If the latency time is longer, the receiver may experience prolonged or even failed startups.

If the latency is not deterministic (meaning, that the client can not announce this to the server through the latency parameter), then one may choose to only use the 'eph' command.

## 3.2 Response format.

Upon successful reception of a client's request, the server will return data and terminate the connection.

The data is split in two parts:

- A header containing ASCII data, describing the data that follows
- A data section with 8-bit raw binary data or ASCII data, depending on the request.

In case of a successful, fully authorized request, a typical response from the server looks as follows:

```
u-blox a-gps demo server (c) 1997-2005 u-blox AG\n <1>
Content-Length: 2392\n <2>
Content-Type: application/ubx\n <3>
\n <4>
<<5> 2392 bytes of binary day data>
```

Everything within "<" and ">" is not being sent by the server, but is added here for descriptive purposes. The "\n" indicates the newline character.

1. This is the welcome message from the server
2. The Content-Length line will tell how many bytes of data will be sent in the data section
3. The Content-Type line describes what format the data is in. In case of a successful request, this will be in "application/ubx" format
4. An empty line indicates the end of the header, and the start of the data section
5. In the data section, the indicated number of raw 8-bit data bytes will be sent. The client shall forward this data unmodified to the GPS receiver. Please note that in cases of "full" and "aid" commands, the received data shall be modified immediately, as the data contains approximate knowledge of time. Delays of 1 second or above may result in degraded startup performance of the receiver.

If an error happens, the server will respond slightly different:

```
u-blox a-gps demo server (c) 1997-2005 u-blox AG\n <1>
Content-Length: 38\n <2>
Content-Type: text/plain\n <3>
\n <4>
error: no approximate position given\n <5>
```

Everything within "<" and ">" is not being sent by the server, but is added here for descriptive purposes. The "\n" indicates the newline character.

1. This is the welcome message from the server
2. The Content-Length line will tell how many bytes of data will be sent in the data section

3. The Content-Type line describes what format the data is in. In case of a failure request, this will be in "text/plain" format
4. An empty line indicates the end of the header, and the start of the data section
5. In the data section, the indicated number of ASCII data bytes will be sent. The client shall **not** forward this to the receiver, as it is an error message

The following error messages are currently delivered by the server:

| **Error Message:** | **Reason** |
|---|---|
| `no command given` | There was no command in the request |
| `invalid command` | A command was requested but is not supported |
| `authorization failed` | - Username or password is missing<br>- The username is not valid<br>- The password does not match to the username |
| `no approximate position given` | Neither lat and lon, nor ex, ey and ez values were given in the request |

# A Sample Client implementation

The following code implements a typical client. It connects to the server, send its request, receives the data and, if of application type UBX, sends that data to a GPS receiver via a serial line.

The sample code is written in Perl language and has been tested on a Linux system. Please see your favourite Operating System and Programming Language for information on generic Serial Port- and Socket-Programming.

```
#!/usr/bin/perl -w

# Package for Serial Port handling:
use Device::SerialPort;
# Package for Socket connections:
use IO::Socket;

################################
# INITIALIZATION               #
################################


################################
# RS232 settings               #
################################

$port = '/dev/ttyS0';
$baud = 57600;

################################
# SERVER settings              #
################################
$aiding_server = 'agps.u-blox.com';
$aiding_port    = 46434;
$user = 'foo@bar.com';
$pwd = 'whatever';
$lat = 47.28;
$lon = 8.56;
$acc = 1000;

$request = "cmd=aid;user=$user;pwd=$pwd;lat=$lat;lon=$lon;pacc=$acc\n";

# open the serial port, using the Device::SerialPort methods
my $gps = Device::SerialPort->new ($port) || die "Can't open $port: $!";

# apply settings to the serial port
$gps->baudrate($baud)  || die "fail setting baudrate";
$gps->parity("none")   || die "fail setting parity";
$gps->databits(8)      || die "fail setting databits";
$gps->stopbits(1)      || die "fail setting stopbits";
$gps->handshake("none")|| die "fail setting handshake";
$gps->write_settings   || die "no settings";

################################
# OPEN CONNECTION TO SERVER    #
################################

print "Connecting to server $aiding_server:$aiding_port\n";

$agps = IO::Socket::INET->new("$aiding_server:$aiding_port")
    || die "unable to contact $aiding_server: $!\n";


################################
# SUBMIT REQUEST               #
################################

print $agps $request;
```

```
################################
# RECEIVE RESPONSE HEADER      #
################################

while ($_ = $agps->getline)
{
    # $_ now contains one ASCII line from the server

    if (/^\s*\n$/)  # if we get an empty line
    {
      last; # jump out of loop
    } else {
      # Look whether it has t
      if ( $_ =~ /Content-(\S+):\s*(\S+)/)
      {
          $header{$1} = $2;
      } else {
          # server sends something we don't need.
      }
    }
}


#################################
# LOOK AT HEADER, DECIDE ACTION #
#################################

# If Content Type is text/plain, we dump data to
# the terminal[must be error or something else
# the GPS receiver does not understand]

if (defined $header{'Type'} && ($header{'Type'} eq "text/plain"))
{
    while($_ = $agps->getline)
    {
      print $_;
    }

}

# If the Content-Type is is application/ubx
# we read in that full chunk of data
# and send it to the receiver right away
if (defined $header{'Type'} && ($header{'Type'} eq "application/ubx"))
{
    # read a number of bytest
    if ( read($agps,$ubxdata,$header{'Length'}) != $header{'Length'})
    {
      die "wrong number of bytes received\n";
    }

    # say what we're doing
    printf "Aiding Data: %s bytes \n",$header{'Length'} ;

    # send it to the ANTARIS receiver
    $gps->write($ubxdata);

    # ... and what until the Serial TX buffer is emptied
    $gps->write_drain();
}

# close the GPS receiver port
undef $gps;
# and the socket
undef $agps;
```

# B Sample Server implementation

This sample code basically implements an Aiding server, for deployment in a closed system. An example of such a system would be a fleet management system, where there is a communications link between individual vehicles and the central server.

This server sample code has the following characteristics:

- Client Side (i.e. GPS receivers that need Aiding data)
    - Communication with the clients is done through TCP/IP.
    - Clients shall autonomously connect to Port 46434 on the server, and request aiding data (i.e. Clients need to Pull the data)
        - Adding Push- or Broadcast-like setups can be achieved by modifying the source code accordingly
    - No authentication is being done to the clients.
    - Every connection gets a current set of ephemeris, and AID-INI (if enabled in the server)
    - The AID-INI message provided includes a rough position- and time-estimate. Please read the comments in the source code for function clientdata_prepare() on possible options.
    - The communication to the client is HTTP like, as with the u-blox AGPS Server. See the implementation of function clientdata_transmit().
    - All data transmitted from the clients to the server is ignored.
- Server Side
    - Instead of having a local GPS receiver collecting Ephemeris data, that Ephemeris data is being downloaded from the public u-blox server at regular intervals. Communication with that server is also done through TCP/IP, and requires an Internet connection. For this functionality, a user account on the u-blox server is required. See the implementation of retrieve_ephemeris() for details.
    - The server is multi-client capable, =fork()=ing individual processes. See the main loop in the beginning of the source code.
- Generic
    - The source is written for Perl 5. It has no special requirements on external Packages other than the standard IO::Socket. Using Time::HiRes would be beneficial if the time of the server is synchronized to a global standard, and if the network latency from the server to the clients is low or constant.
    - Operational parameters (username, password, timeout values, appr. position) can be set in the source.
    - All logging is done to STDOUT, errors go to STDERR
    - The source code includes comments which document each step

- The source code has been tested on Linux 2.4 and 2.6, and Windows XP (ActiveState[2] Perl).

```
#!/usr/bin/perl
# ###########################################################################
#
# Copyright (C) u-blox ag
# u-blox ag, Thalwil, Switzerland
#
# All rights reserved.
#
# This  source  file  is  the  sole  property  of  u-blox  AG. Reproduction or
# utilization of this source in whole or part is forbidden without the written
# consent of u-blox AG.
#
# ###########################################################################
#
# Project:
#        ANTARIS AGPS CACHING SERVER
# Description:
#        Multi-client daemon with regular ephemeris cache
#        update from an Internet site
# ###########################################################################
#
# Credits
#
# - socket/daemon code inspired by 'Advanced Perl Programming',
#   Chapter 12.
# - aiding functionality: u-blox internal development
# - aiding protocol: see UBX protocol specification
#
# ###########################################################################

use strict;
use IO::Socket;
#use Time::HiRes qw ( time ); # If available....


# Settings

# Username/Password. Please fill in what agps-account@u-blox.com
# gave you

my $user = 'you@somewhere.com';
my $pass = 'Asdfghjkl';

my $lat = 47.28; # Appr. Position, being used for Ephemeris transfer
my $lon = 8.56;  # from server. [Degrees]
my $acc = 1000;  # [m]

# Which port to listen to for clients to connect
my $LISTENPORT = 46434;

# Parameters to access the Ephemeris Server
my $EPH_SOURCE = 'agps.u-blox.com:46434';
my $EPH_REQ = "cmd=eph;user=$user;pwd=$pass;lat=$lat;lon=$lon;pacc=$acc";


my $sock;
my $client;

my $get_eph_retrieval_time;
my $eph_data;


$sock = new IO::Socket::INET(LocalPort => $LISTENPORT,
                   Reuse     => 1,
                   Listen    => 5)
    or die "can't create local socket: $@\n";

# Timeout the socket, so that the parent process can
# do some other work, and get out of the blocking
# accept() call
$sock->timeout(10);

# Avoid Zombies
$SIG{CHLD} = sub { wait(); };

print STDOUT localtime(time()) . " Accepting connections on Port ", $LISTENPORT, "...\n";

# ########################################################
#
# This is the main loop.
# In here, we
# - accept new client connections, fork, and handle the client
# - periodically check for ephemeris
# - repeat the above forever
#
# ########################################################
while(1) # loop forever
{
    # Check for new ephemeris
    retrieve_ephemeris($EPH_SOURCE,$EPH_REQ);

    # Wait for new connections
    if ((defined $eph_data) &&
   (length($eph_data)>0) &&
   ($client = $sock->accept())
   )
    {
   if (defined $client)
   {

       # new connection established
       print STDOUT localtime(time()) . " " . $client->peerhost() .":".$client->peerport() . "
connect\n",;
       # Fork, to have the server process listen again immediately
       if (fork() == 0)
       {
      my $cli_data;
      $cli_data = clientdata_prepare($client);
      # Child Process - send data
      clientdata_transmit($client,$cli_data . $eph_data); # send concatenated data (client-specific
aid-ini and ephemeris)
      $client->close();
       }
   }
   else
   {
       print "Undefined client\n";
   }
    }
}
1; #never get here


# ########################################################
# The following function prepares a UBX-AID-INI message
#
# The receiver can benefit if we provide it an
# approximate position and time.
# The better the position- and time knowledge, the faster
# it can start up. An approximate position (for example,
# with an uncertainty of 100km) is still better than no
# position information at all.
#
# One possibility is to provide the receiver with his last
# reported position out of a fleet management software,
# and increase the uncertainty of the postion information
# with the age of the information
#
# If this is not possible (for example, because we do not
# have even a rough idea where our client is), we simply
# return an empty string.
#
```

```
# If it is possible to do a position aiding, we put
# together an UBX message. For the format of AID-INI,
# please refer to the protocol specification
#
# If an ANTARIS receiver gets AID-INI data, but already has
# better knowledge of position and/or time (which it finds
# out by comparing the 'accuracy' numbers) than what is
# provided, it will ignore the less accurate data
# automatically
#
# ########################################################

sub clientdata_prepare
{
    my $socket = shift;

    my $posx;   # ECEF X component in meters
    my $posy;   # ECEF Y component in meters
    my $posz;   # ECEF Z component in meters
    my $posacc; # Position accuracy estimate in meters

    my $timenow; # Current time in seconds since GPS 0:0
    my $timeaccuracy; # Etsimated Time accuracy

    my $aidini=""; # Buffer for assembling AID-INI message

    # ------------------------------------------------
    # ESTIMATE POSITION OF THE CLIENT
    # ------------------------------------------------

    # Example 1:
    #
    # lookup last position of this client from database
    # by identifying the client with its IP number
    #
    # my $age
    # ($posx,$posy,$posz,$age) = lookup_last_position($socket->->peerhost())
    #

    # Example 2:
    #
    # Send all clients the same position with an accuracy
    # figure that covers all of our fleet (for example, size
    # of the country they operate in)

    $posx = 4286470.0; # Example position of u-blox HQ
    $posy = 645612.0;
    $posz = 4663733.0;
    $posacc = 1500000; # Estimated accuracy 150km

    # Example 3:
    #
    # Don't do AID-INI - simply return an empty string
    #
    # return "";

    # ------------------------------------------------
    # ESTIMATE TIME in GPS TIME FRAME
    # ------------------------------------------------
    # the following assumes that the Server where this code
    # is run, is somewhat accurately synchronized (sub 200ms
    # to UTC), for example using NTP
    #

    $timenow = time(); # could use Time::HiRes here, to get sub-second resolution (see 'use'
statement at the top)
    $timenow += 15 - 3657*86400 ; # correct offsets of time scales

    $timeaccuracy = 1.0; # estimated time accuracy in [s]

    # $timenow contains seconds since GPS time 0:0
    #
    # at this point, we could also correct $timenow
    # by an average network latency we see
    #
    $timenow +=0.5;
    # to tune, this, one can observe the UBX-INF-DEBUG message
    # "RXM INI dt 725295.363 us". This message is output
    # after the first position fix, and gives the time
    # correction that was applied to the time the
    # receiver got [if a coldstart with aiding was done]


    # ------------------------------------------------
    # ASSEMBLE AID-INI MESSAGE
    # ------------------------------------------------
    #  1. Header
    $aidini  = pack "CCCCv",0xb5,0x62,0x0B,0x01,48;
    #  2. Position Data
    $aidini .= pack "lllVv",
         int(100*$posx),
         int(100*$posy),
         int(100*$posz),
         int(100*$posacc),
         0;
    #  3. Time
    $aidini .= pack "vVl",
         int($timenow / (7 * 86400)), # Week Number
         ($timenow * 1000 )% (7 * 86400 * 1000), # Time of week in [ms]
         0;
    $aidini .= pack "VVlV",
         $timeaccuracy*1000, # Time accuracy
         0,0,0;
    # 4. Flags
    $aidini .= pack "V",0x03; # == Time and Position valid
    # 5. Checksum
    $aidini .= ubx_checksum($aidini);

    return $aidini;
}



# #######################################################
# The following function talks to a client and sends data
#
# We are new in the server role, so talk HTTP-like
# and write the data
#
# #######################################################
sub clientdata_transmit
{
    my $socket = shift;
    my $data = shift;

    printf STDOUT localtime(time()) . " " . $socket->peerhost() .":".$socket->peerport() . " send %i
bytes\n",length($data);
    printf $socket "Content-Length: %i\n",length($data);
    printf $socket "Content-Type: application/ubx\n";
    printf $socket "\n";
    print  $socket $data;
}


# #######################################################
#
# The following connects to a AGPS Server, and retrieves
# Ephemeris data.
#
# The function downloads data every $eph_retrieve_period
# second. It fills the global variable $eph_data
# if successful.
#
# We need to send a request to the server,
# and get back data requested.
#
```

```
#   srv:    "<some welcome message \n"
#   clnt:   "<request string>\n"
#   srv:    "Content-Length: <bytes>\n"
#   srv:    "Content-Type: application/ubx\n"
#   srv:    "\n"    <-- indicates that data is following
#   srv:    "<#bytes of data>"
#
# If an error occurs, it prints to STDERR
#
# ########################################################
sub retrieve_ephemeris
{
    # Function Arguments
    my $srv = shift;
    my $req = shift;

    # Constants
    my $eph_retrieve_period = 600; # [s]

    # Local Variables
    my $bytes = 0;
    my $type = "";
    my $tmpbuf = "";

    # Check whether it is time to download again
    return if (defined($get_eph_retrieval_time) && ( time() - $get_eph_retrieval_time <
$eph_retrieve_period));

    # open a new connection
    my $ephsrv = IO::Socket::INET->new($srv)
   || die "unable to contact $srv: $!\n";

    # send the request
    print $ephsrv $req . "\n";

    # and read the results. First, the header
    while ($_ = $ephsrv->getline)
    {
   # $_ now contains one ASCII line from the server
   if (/^\s*\n$/)  # if we get an empty line
   {
       last; # jump out of while loop - header completed
   }
   else
   {
       # parse Content-Length and Content-Type
       if ( $_ =~ /Content-(\S+):\s*(\S+)/)
       {
      $type  = $2 if ($1 eq "Type");
      $bytes = $2 if ($1 eq "Length");
       }
   }
    }

    # and then, the data
    if ($bytes && ($type eq "application/ubx"))
    {
   # If we receive content-type application/ubx, we will
   # get the data we wanted, so read it
   if ( (read($ephsrv,$tmpbuf,$bytes) == $bytes) && (length($tmpbuf)==$bytes))
   {
       # print out a message if this ephemeris is different to what we had
       if ((!defined $eph_data) || ($eph_data ne $tmpbuf))
       {
      print STDOUT localtime(time()) . " $srv: new ephemeris retrieved\n";
      $eph_data = $tmpbuf;
       }
       # remember time of retrieval
       $get_eph_retrieval_time = time();
   }
   else
   {
       print STDERR localtime(time()) . " $srv: unable to read $bytes\n";
    }

     }
    elsif ($bytes && ($type eq "text/plain"))
     {
   # if we get content-type text/plain, it is
   # some error message -> read it, and dump it
   # to stdout

   if ( (read($ephsrv,$tmpbuf,$bytes) == $bytes) && (length($tmpbuf)==$bytes))
   {
       # Error message from the server - write to STDERR
       print STDERR localtime(time()) . " $srv: $tmpbuf";
   }
   else
   {
       # Unable to read the number of bytes.
       print STDERR localtime(time()) . " $srv: unable to read $bytes\n";
   }
     }


    $ephsrv->close();
    undef $ephsrv;
}

# #######################################################
#
# Helper function to calculate checksum over a given payload
#
# see UBX Protocol specification for more information
#
# #######################################################
sub ubx_checksum
{
    my @vals = unpack "C*", $_[0];

    shift @vals;shift @vals; # ignore header

    my $ck_a = 0;
    my $ck_b = 0;
    foreach (@vals)
    {
        $ck_a  = ($ck_a + $_   ) & 0xff;
        $ck_b =  ($ck_a + $ck_b) & 0xff;
    }
    return (pack "v",($ck_b << 8) + $ck_a);
}
```

# C Server Settings

The following connection settings shall be used

| | |
|---|---|
| Server Name | agps.u-blox.com |
| Server Port | 46434 |
| Protocol | TCP |
| Username / Password | Please contact support@u-blox.com |