TagMaster

TAGP Protocol Specification v1.1

GEN4
LONG-RANGE RFID

Revision	Date	Comment
04	2007-10-16	Updated for System Software v1.4.0

Copyright

The copyright and ownership of this document belongs to TagMaster AB. The document may be downloaded or copied provided that all copies contain the full information from the complete document. All other copying requires a written approval from TagMaster AB.

Disclaimer

While effort has been taken to ensure the accuracy of the information in this document TagMaster AB assumes no responsibility for any errors or omissions, or for damages resulting from the use of the information contained herein. The information in this document is subject to change without notice.

1   Introduction

The TagMaster GEN4 Readers have an application abstraction layer that software designers may use for application development. The abstraction layer is a high-level daemon process called the TagMaster Daemon (tagd).

tagd listens for incoming connections and serves the connected processes with information about the Reader (ID-tag readings, motion detections, and so forth) and means to control the Reader (output frequency, colour of visual indicator, and so forth).

tagd is accessible over TCP/IP sockets using a software communication protocol called the TagMaster Protocol (TAGP).

This document specifies the TAGP protocol used for communication between tagd and clients.

If considering developing application software using TAGP, note that TagMaster provides a Software Development Kit (SDK), see SDK User's Manual [1]. The SDK includes a software library called taglib, see taglib Software Library Specification [2], that implements the client side of TAGP and can be used to build C/C++ applications.

From a protocol perspective, this document is divided in two parts. Section 2 to section 3 specifies the fundamental message formats and design concept. Section 4 to section 6 specifies Reader parameters (variables, events, devices, and so forth) that can be controlled via the protocol.

It is required to have general knowledge about the TagMaster RFID system to fully comprehend the information in this document. It is recommended to read the GEN4 Reader User's Manual [3] before reading this specification.

The aim of this specification is to present a software designer, who has the necessary education and training, with the information needed to implement TAGP in custom-made application software for the GEN4 Reader.

2   Functional Description

The TAGP protocol is used for communication between a tagd server and clients.

The protocol has been designed with focus on simplicity. Interaction with a server talking TAGP is possible without specialised software, for instance using a standard telnet client (see section 10 Appendix for an example).

The server and the client communicate by exchanging so-called TAGP messages. All messages are based on readable ASCII characters.

The TAGP protocol requires a reliable and connection oriented transportation protocol, for instance TCP/IP.

Using TAGP, multiple clients can be connected and communicate with the server simultaneously. The maximum number of simultaneous client-connections is defined by the server. The TAGP protocol has messages that allow clients to exchange information with each other via the server.

Multiple clients connected to the same tagd server share the same resources; those are the same hardware, functions and so forth. Take measures to ensure that resource conflicts between clients do not arise.
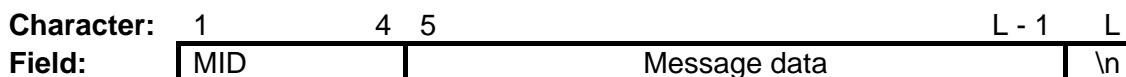
2.1   Message Format

All TAGP protocol messages follow a standard message format.

Each TAGP message begins with a Message Identifier (MID). The MID consists of four uppercase ASCII characters, for example "HELO" and "VARS". However some MIDs consist of three uppercase characters, for example "GET " and "SET ", in which case the fourth character in the MID is a space character (ASCII value 0x20).

The MID is followed by the message data, which is a field of variable length.

The message is terminated by a newline character: "\n" (ASCII 0x0A).

Figure 1   Format of a TAGP message with length L

Character:	1	4	5		L - 1	L
Field:	MID		Message data			\n

The protocol handles the message data field as a string of characters. Consequently each message can have its own structure for the data field, which allows for a flexible protocol format. The maximum length for a message is 1024 characters including the terminating newline character (L_{MAX} = 1024).

Protocol messages are case sensitive, if not explicitly stated otherwise. A protocol message specified with four upper case characters is not recognized by tagd if it sent as four lower case characters, or sent as a mix of upper and lower cases.

A client should not respond to messages sent from the server. Retransmissions are disallowed and not necessary since a reliable transport layer (typically TCP) is required by the protocol.

The client does not have to wait for the reply of a message before sending a new message. The purpose is to have a non-restrictive and simple protocol implementation. Some message types however limit the number of outstanding messages of the same type waiting for response.

2.2 Data Formats and Data Types

This section specifies control characters and various data formats that the TAGP protocol uses.

2.2.1 Escaped Characters

TAGP does not have any restrictions regarding what ASCII characters can be used. However, some characters are part of the protocol and must be escaped when included in the message data. By escaping a character the message receiver is told to interpret the character as text and not as a protocol-specific character.

In order to escape a character, a percentage sign followed by a two-character hexadecimal representation of the character's ASCII code is used. For instance the newline character is written as "%0A".

Table 1 Protocol-specific characters that must be escaped if part of message data

Character	Escaped as	Description
\0	%00	Null
\n	%0A	Newline
%	%25	Percentage sign
,	%2C	Comma
;	%3B	Semi colon
=	%3D	Equal sign

Example: To assign a variable the string value "two plus two=four", the equal sign must be replaced with its ASCII code as in "two plus two%3Dfour" (the underlined part of the string represents the equal sign).

A message can be sent with all data characters escaped, for instance the string "AB" is equal to "%41%42". This style is however not recommended since it creates unnecessary overhead.

Example: Sending a string value containing the percentage character requires that it is escaped. For instance the string "99%" should be escaped with "99%25".

2.2.2 String

A string is a sequence of arbitrary ASCII characters. The maximum length of a string depends on the message format. Note that all control characters must be escaped in a string.

2.2.3 Numerical

A numerical value is represented by a string consisting of numerical characters. For instance the integer value 12345 is written as the string "12345".

2.2.4 Boolean

Boolean variables can hold string values "ON" or "OFF" (equivalent to true or false; and high or low). Boolean string values are not case sensitive, that is, "ON" has the same meaning as "on".

2.2.5 Date and Time Stamp

Date and time stamp is a string formatted according to ISO 8601, which is the international standard for date and time notation as shown below:

`YYYYMMDDhhmmssfff`

- YYYY is the year in the common Gregorian calendar
- MM is the month of the year between 01 (January) and 12 (December)
- DD is the day of the month between 01 and 31
- hh is the number of complete hours that have passed since midnight (00-23)
- mm is the number of complete minutes that have passed since the start of the hour (00-59)
- ss is the number of complete seconds since the start of the minute (00-59)
- fff is the fractions of a second as milliseconds (000-999)

Example: 15th October 2007 at 15:54:10 and 238 ms can be represented with the string "20071015155410238"

2.2.6 List

A list is a group of strings that are separated by a comma or a semicolon, depending on the message specification. For instance the list of three clients named "Axel", "Bob" and "Cesar" would be written as: "Axel,Bob,Cesar".

2.2.7 Binary Data Representation

The protocol is supposed to be readable by humans, but it also supports transmission and reception of binary data. The escape character mechanism is used and a byte of data is represented by a percentage character and its hexadecimal value written as two uppercase ASCII characters. For instance, the decimal value 255 (0xFF) is written as: "%FF".

A larger chunk of binary data is consequently written as a string of ASCII characters representing the hexadecimal value of each byte. For instance the binary value 0x75BCD15 is written as "%07%5B%CD%15". Note that the string occupies exactly 12 characters (three characters per byte) and that all characters are written in uppercase.

3   Protocol Messages

The protocol messages are divided in two sections, messages from client to server and messages from server to client.

3.1   Client Messages

Sections 3.1.1 to section 3.1.9 describe messages sent from client to server.

A server always replies to client messages with a RPLY message, which is described in section 3.2 Server Messages.

3.1.1   HELO: Setup Connection with Server

MID: "HELO"
Message data: <protocol name>/<version>

The HELO message is used to setup a connection and it should be the first message sent from a client to the server after the client has connected to the server. HELO is used to verify that both client and server speak the same protocol.

Until a HELO message has been successfully received, the server does not respond to any other messages, including unknown and corrupt messages.

Table 2   Example of a connection setup with protocol name TAGP and version 1.1

Client message	`HELOTAGP/1.1\n`
Server response	`RPLYHELO00\n`

If a server receives a HELO and implements TAGP with the stated version, it replies with a "00", which means that a connection is set up and that the server is ready to communicate with the client.

Table 3   Example of a connection setup with version mismatch

Client message	`HELOTAGP/1.0\n`
Server response	`RPLYHELO81TAGP/1.1\n`

If the server receives a HELO message with a protocol or version that is does not support, it replies with an error code that explicitly states which version of TAGP it supports.

3.1.2   SET: Set Value to Variable

MID: "SET "
Message data: <variable name>=<value>

Note that the fourth character in this MID is a space character (ASCII value 0x20). SET assigns a value to a server variable, either locally to the client or globally to all connected clients.

Each variable is uniquely identified by its variable name. The variable name is case-sensitive, which means that a server may specify the variables: "foo", "FOO" and "Foo", since they all refer to different variables.

A variable can be set to any ASCII character string as long as protocol-specific characters are escaped.

Note that only one variable at a time can be set. Only one SET message can await a RPLY message at any given time. If a client tries to send a second SET before the first SET has been confirmed, the server will reply with an error code.

Table 4   Example of setting value "100" to a server variable named "FOO".

Client message	SET FOO=100\n
Server response	RPLYSET 00\n

The server confirms with a "00" if the variable was set.

The server replies with an error code if the variable name is not known by the server, the value is out of bounds, the variable is read-only, or the variable is locked by another client.

3.1.3   GET: Get Value from Variable

MID: "GET "
Message data: <variable name>

Note that the fourth character in this MID is a space character. GET is used to request the value of a server variable. The server replies with a RPLY message followed by the name of the requested variable and its value.

Table 5   Example of getting value of variable "FOO"

Client message	GET FOO\n
Server response	RPLYGET 00FOO=100\n

If GET includes a server variable that is readable, the server will reply with its value but if the requested variable is not known or if it is not readable, the server will reply with an error code.

3.1.4   VARS: Get List of Variables

MID: "VARS"
Message data: Empty

The VARS message retrieves a list of the server variables and their attributes. The number of variables can be large and if the reply exceeds the maximum message length, it is divided into several replies. The return code states if more replies follow.

A list of variables and attributes follows the return code of the reply. A semi-colon denotes the end of a variable name with attributes. A variable and its attributes are separated by a comma.

The attributes that a variable is associated with is specified by the server. The protocol defines the following attributes:
- R: Read-only, that is not writable
- W: Writable (and readable)
- G: Global, the variable can be accessed by all clients
- L: Local, only accessed by one client

All variables are read-only and global by default. A local variable, with attribute "L", is a variable that is in a client's private namespace and it cannot be accessed directly by other clients. Some of the local variables can however be indirectly read via the server.

Table 6   Example of getting list of server variables

Client message	`VARS\n`
Server response	`RPLYVARS00FOO,GR;DUMMY,LW\n`

In the example above the variable "FOO" is a global read-only variable while the variable "DUMMY" is local, readable, and writable.

3.1.5   LOCK: Control Exclusive Access

MID: "LOCK"
Message data: "GET "
Message data: "RELS"

LOCK is used to request exclusive access to the server properties and functions (for instance global variables and devices) and limit other clients' access to the server.

Clients can require and release exclusive access by sending message data "GET " (note that the fourth character is a space character) and "RELS" respectively.

The locking client's name is set in a global lock variable at the server. Other clients that also need access can send a message to the locking client. The locking client is therefore required to set its client name prior the LOCK request.

A lock is released automatically if the locking client disconnects from the server.

Table 7   Example of requiring exclusive access

Client message	`LOCKGET \n`
Server response	`RPLYLOCK00\n`

The example above is a successful lock request. If the lock is currently set by another client the server will report this with a return code.

Table 8   Example of releasing a lock.

Client message	`LOCKRELS\n`
Server response	`RPLYLOCK00\n`

3.1.6 PUSH: Write Data to a Device

MID: "PUSH"
Message data: <DID><device data>

The PUSH message is used to write information to a device when a standard SET message is not applicable, typically when more advanced input is required.

A device is uniquely defined by a Device Identifier (DID), which is four uppercase ASCII characters. Clients use the device data field to specify what data to push to a device. The format of the data field depends on the device and is specified in section 5 Devices.

The conceptual difference between a PUSH message and a SET message is that SET stores a value but PUSH sends volatile information required to perform an action. Pushed information is not stored or remembered by the server. PUSH messages also handle more flexible data and are not bound to the pre-defined data types used by standard variables handled by SET.

Table 9   Example of writing data "0xAA22" to a dummy device with address "0x05", identified with DID "DUMY"

Client message	PUSHDUMY05=AA22\n
Server response	RPLYPUSH00DUMY\n

Note that only one PUSH message can await a reply at any given time. If a client tries to send a second PUSH before the first PUSH has been confirmed, the server will reply with an error code.

3.1.7 PULL: Read Data from a Device

MID: "PULL"
Message data: <DID><device data>

The PULL message is used to read information from a device when a standard GET is not applicable.

The device data field specifies what information is requested from a device.

Table 10   Example of reading a dummy device "DUMY" at address 0x05.

Client message	PULLDUMY05\n
Server response	RPLYPULL00DUMYAA22\n

The server replies with data that the device returns, appended to the return code. If the PULL fails, the server returns an error code.

Note that only one PULL message can await a reply at any given time. If a client tries to send a second PULL before the first PULL has been confirmed, the server will reply with an error code.

3.1.8 TALK: Client to Client Data Transfer

MID: "TALK"
Message data: <list of clients>;<talk data>

TALK is used to send data to other clients. Receivers of the message are specified by a list of clients. The talk data field holds the data to send to the clients in the list.

In order to send and receive talk data, the sending and receiving clients must have set client names and enabled talking (refer to variables NAME and TALK in Table 19). A client cannot send a TALK message to itself.

Table 11   Example of sending "Hello all applications!" to clients: "Axel", "Bob" and "Cesar"

Client message	`TALKAxel,Bob,Cesar;Hello all applications!\n`
Server response	`RPLYTALK00\n \n`

As long as at least one of the receiving clients exists and accepts talk data, the server will confirm the TALK message "00". The sending client is responsible to identify clients that did not receive the message. If fine-grained control is required, it is recommended to just send talk data to one client at a time.

3.1.9 PING: Request Echo from Server

MID: "PING"
Message data: Empty

The PING message requests an echo reply from the server. PING is primarily used to keep a connection alive if there is a risk that it will be terminated because it has been idle to long.

Note that keeping a connection alive is not necessary if a client is connected to a server residing on the same host.

Using PING to periodically check if the server is running should not be necessary.

Table 12   Example of requesting an echo reply

Client message	`PING\n`
Server response	`RPLYPING00\n`

3.2 Server Messages

Sections 3.2.1 to section 3.2.4 describe messages sent from server to client.

3.2.1 RPLY: General Message Reply

MID: "RPLY"
Message data: <MID><return code><return data>

Messages sent from clients to a server are replied to by the server.

The RPLY message consists of the MID of the replied client message, a two-character hexadecimal return code that holds information about the client messages

interpretation and execution, and an optional string that may return data to the client or explain the return code.

The return data field is described in section 4 Variables and section 5 Devices.

Table 13 Example of reply to a successful HELO message

Server message	`RPLYHELO00\n`

Table 14 Example of reply when a client tries to get an unknown variable

Server message	`RPLYGET 81Variable not found\n`

Table 15 Server return codes

Code	Description
0x00	OK. Message received and action performed successfully. Also used to state the end of long messages, for instance if prior message had return code 0x01.
0x01	OK continued. Message received and action performed successfully and there will be an additional response. Used when the length of the reply is greater than the maximum TAGP message length.
0x02	Syntax error or bad message format. The received TAGP message is impossible to interpret correctly.
0x03	Variable value out of range, no client received talk message, or client name already in use.
0x04	Too many SET messages. If a client has sent a second SET while it awaits the reply for the first SET, the server replies with this error code.
0x05	Already locked, or not your lock. In response to LOCK when another client already has locked or a client tries to release when not being the locking client.
0x81	Bad TAGP version or variable not found. If the server does not support the version of TAGP that the client speaks it replies with the version it supports.
0x82	Variable error

3.2.2 TALK: Talk Message Forward

MID: "TALK"
Message data: <sender>;<data>

The server TALK message is used to forward client TALK messages.

When the server receives a TALK message from a client, it replies to that client with a RPLY message.

If the TALK message is correct and there are receivers of the message, the server forwards it using the format specified above. The sender field is the name of the sending client and the data field is the data that the sender wishes to send.

Table 16   Example of server message when client "Axel" sends "Hello all applications!"

Server message	`TALKAxel;Hello all applications!\n`

The server will only forward TALK messages to clients that accept TALK messages and have registered a name.

3.2.3 EVNT: Event

MID: "EVNT"
Message data: <EID><time stamp><event data>

EVNT messages are used to inform the client that an event has occurred. All EVNT messages follow the data format specified above.

The Event Identifier (EID) is four uppercase ASCII characters that uniquely specify an event type.

A time stamp with millisecond resolution follows the event identifier, as specified in section 2.2.5 Date and Time Stamp. EVNT messages are sent as soon as they occur and are received chronologically according to the time stamp.

Each event has a data field that holds information correlated to the event, for instance an ID-tag event will hold ID-tag data and a tampering switch event the current state of the switch. The format of the data field depends on the type of event and it is specified in section 6 Events.

Table 17   Example of opening the tampering switch at 2007-01-26 10:03:28.654

Server message	`EVNTTMPR20070126100328654TAMPER=0\n`

3.2.4 DBUG: Debugging Client and Server Messaging

MID: "DBUG"
Message data: <from><protocol message>

The server DBUG message is used when debugging the communication between a client and the server. Also referred to as eavesdropping a client.

The from data field is one character that specifies from who the message was sent. An "S" is used for messages that are sent from the server and a "C" when the message comes from the client that is eavesdropped.

The protocol message field is an exact copy of the message that was sent between client and server. All messages sent from a client are forwarded to an eavesdropping client by the server. All replies from the server are also sent to the eavesdropping client.

Table 18   Example of received message when eavesdropping a client that sends a PING message

Server message	`DBUGCPING\n`
Server message	`DBUGSRPLYPING00OK\n`

To start eavesdropping, a client sets its local DEBUG variable to the name of the client to eavesdrop. For security reasons a client must specifically state that it allows eavesdropping by setting its local EAVESDROP variable to "ON".

4   Variables

Server variables can either be local to a specific client or global to all connected clients. All variables are readable and some variables are also writable.

Default value for each variable is written in parentheses after the variable name. If a default value is not applicable the value is omitted, for instance the current system time variable.

4.1   Local

This section specifies the local variables that can be accessed from a client.

Table 19 Local variables that hold information about client settings.

Name	Type, Attribute	Description
NAME (null)	String, Writable	The name of the client, which must be unique among clients connected to server. NAME is default set to null. A client is not required to specify a name but in order to set a LOCK and to TALK, the client must have a name set. The maximum length of the NAME string is 32 characters.
TALK ("OFF")	Boolean, Writable	To receive and to send TALK messages, a client must explicitly specify that it accepts TALK messages by setting the TALK variable to "ON". To stop receiving TALK messages the client can set the variable to "OFF".
DEBUG ("OFF")	String, Writable	Eavesdrop all TAGP communication between a client and server. Specify the client to eavesdrop by assigning the local DEBUG variable the eavesdropped client's name.
EAVESDROP ("OFF")	Boolean, Writable	A client must explicitly specify that it allows other clients eavesdropping its communication with server by setting the variable to "ON".

4.2 Global

This section specifies the global variables that can be accessed from a client.

4.2.1 System Information

The following variables hold information about the Reader system. Variables marked with an asterisk (*) require that the corresponding Reader option is available.

Table 20 Variables that hold information about the Reader system

Name	Type, Attribute	Description
CLIENTS	List, Readable	Holds a list of connected clients containing their names.
LOCK (NULL)	String, Readable	Holds the name of the client that currently has exclusive access to server.
CB_SERNO	Numerical, Readable	Holds the controller board serial number.
CB_REVISION	Numerical, Readable	Holds the controller board revision.
CB_TYPE	Numerical, Readable	Holds the controller board type number.
RF_SERNO	Numerical, Readable	Holds the RF-unit serial number.
RF_REVISION	Numerical, Readable	Holds the RF-unit revision.
TIME*	Date & time stamp, Writable	Holds the current system time. The time can be adjusted by setting the TIME variable. The setting is also stored to the battery backed up real-time clock.
TEMPERATURE*	Numerical, Readable	Holds the current controller board chip temperature.
TAGD_VERSION	Numerical, Readable	Holds the tagd version.
TAGMOD_VERSION	Numerical, Readable	Holds the tagmod version

4.2.2　Radio Interface

The following variables control the radio interface properties. Variables marked with an asterisk (*) require that the corresponding Reader option is available.

Table 21　Variables that control radio interface properties

Name	Type, Attribute	Description
CARRIER ("ON")	Boolean, Writable	Controls the output carrier wave that can be set to "ON" or "OFF".
FHSS_MODE* ("OFF")	String, Writable	Controls frequency hopping that can be set to three different values: • "off", frequency hopping is switched off and continuous wave (CW) is used. Refer to the FREQUENCY variable below on how to set the carrier frequency in CW mode. • "on", frequency hopping is used. • "adaptive", adaptive frequency hopping is used. The Reader first listens to each frequency and allocates a list of idle frequencies that are not interfered by other radio sources. Note that this is a future option.
FHSS_BANDS* (null)	String, Writable	Specifies the sub-bands that are used during frequency hopping. The band string may contain the capital letters ("A" to "P"). Each letter corresponds to a sub-band to use. For instance "GHK" means that sub-bands "G", "H" and "K" is used when frequency hopping is enabled.
FREQUENCY* (24500)	Numerical, Writable	The output carrier frequency can be set in steps of 100 kHz in the range 2.4360 to 2.4641 GHz. Setting FREQUENCY to "24500" corresponds to the output frequency 2.45 GHz.
READ_LEVEL (100)	Numerical, Writable	Controls the reading range of the Reader. Can be set in the range 0 to 100, of which 100 sets the maximum reading range.
READ_RANGE (4)	Numerical, Writable	Controls the reading range of the Reader. Can be set in the range 1 to 4, of which 4 corresponds to long reading range and 1 to short reading range.

4.2.3　ID-tag

The following variables control the ID-tag handling.

Table 22　Variables that control the ID-tag handling

Name	Type, Attribute	Description

TAG_BITRATE ("ON")	Boolean, Writable	Controls what bit rate the ID-tag decoder uses. If TAG_BITRATE is set to "ON", high data speed ID-tags are expected. If set to "OFF", low data speed ID-tags are expected.
FILTER_TYPE ("off")	String, Writable	Controls ID-tag filter and can be set to four different values: • "off", no filter is used • "once", an ID-tag is reported once and before it is reported again it must be out of the Reader's reading lobe for the time period specified by FILTER_TIMEOUT • "periodic", an ID-tag is reported periodically with a time period specified by FILTER_TIMEOUT • "report", the filter will work in the same manner as if set to "once" but a special ID-tag event is also sent when an ID-tag is leaving the reading lobe for more than a time period specified by FILTER_TIMEOUT. Note that this is a future feature.
FILTER_TIMEOUT (1000)	Numerical, Writable	Controls the ID-tag filter timeout and can be set with a millisecond resolution in the range 0 to 100000. Note that a low FILTER_TIMEOUT value in combination with a "periodic" FILTER_TYPE may result in a large number of ID-tag events. A timeout value greater than 500 ms is recommended for normal operation.
READ_BEEP ("ON")	Boolean, Writable	Controls whether the Reader should beep with the buzzer every time it reads an ID-tag. Set to "ON" to enable read beep and set to "OFF" to switch it off.
TAG_CRC ("ON")	Boolean, Writable	Controls if ID-tags that have been read with a bad CRC should be discarded or not. If set to "ON", ID-tags that have a bad CRC check are not reported. Note that it is strongly recommended to have CRC discarding set to "ON".
TAG_DATACRC ("ON")	Boolean, Writable	Controls if ID-tags that have been read with a bad user data CRC should be discarded or not. If set to "ON", ID-tags that have a bad user data CRC check are not reported. Note that it is strongly recommended to have user data CRC discarding set to "ON".

4.2.4 Peripherals

The following variables control peripheral devices. Variables marked with an asterisk (*) require that the corresponding Reader option is available.

Table 23 Variables that control peripherals

Name	Type, Attribute	Description
BUZZER ("OFF")	Boolean, Writable	Controls state of the buzzer. If BUZZER is set to "ON" the buzzer will sound until BUZZER is set to "OFF". Setting BUZZER overrides any previous push to the BEEP device (see section 5.3.2).
FAN_OUTPUT* ("OFF")	Boolean, Writable	Controls state of the fan output. "ON" correspond to +5 V DC and "OFF" to 0 V DC.
LED ("off")	String, Writable	Controls state of visual indicator. The variable can be set to four different values, each corresponding to a state: • "off", visual indicator is switched off. • "red", visual indicator show red. • "green", visual indicator show green. • "yellow", visual indicator show a mix of both red and green. Setting LED overrides any prior push to the BLNK device (see section 5.3.1).
RELAY ("OFF")	Boolean, Writable	Controls state of the relay. "ON" correspond to relay open and "OFF" to relay closed. Setting RELAY overrides any prior push to the GATE device (see section 5.4.1).
RS485_FULL ("OFF")	Boolean, Writable	Controls the full duplex properties of the RS485 serial interface. RS485_FULL set to "ON" sets the RS485 interface to a 4-wire interface and "OFF" sets the RS485 interface to a half-duplex 2-wire interface.
TAMPER_SWITCH*	Boolean, Readable	Holds the current state of the tampering switch. TAMPER_SWITCH equal to "ON" means that the tamper switch is pressed down and that the lid is closed. "OFF" mean that the tampering switch is open.

4.2.5 Special Features

Note that all these variables require that the corresponding Reader option is available.

Table 24 Variables that control special features of the Reader

Name	Type, Attribute	Description
APOS_MODE ("OFF")	Boolean, Writable	Controls the mode of Accurate Positioning (APOS), which can be either "ON" or "OFF". If set to "ON", an APOS event will be sent as soon as an accurate position was found.
APOS_THRESHOLD (200)	Numerical, Writable	Sets a threshold in the range 1 to 60000 to the accurate positioning algorithm.
APOS_TIMEOUT (500)	Numerical, Writable	Sets the accurate positioning timeout in the range 1 to 655.
APOS_OUTPUT ("OFF")	Boolean, Writable	If set to "ON", the accurate positioning function outputs a pulse on output 1 to indicate that an accurate position was found, exactly APOS_TIMEOUT milliseconds ago.
DOPPLER_FILTER	Boolean, Writable	Note that this is a future feature.
DOPPLER_RADAR	String, Writable	Note that this is a future feature.
DOPPLER_THRESHOLD	Numerical, Writable	Note that this is a future feature.

5   Devices

This section specifies the various devices that the Reader has. A device can be either a peripheral or a function. Devices cannot be controlled simply by GET or SET. Instead PULL messages or PUSH messages are used to control devices.

5.1   ID-tag

There are two devices regarding ID-tags that can be controlled using PUSH messages.

5.1.1   Writing ID-tag

DID: "TAG "
PUSH data: <mark>;<control>;<user data>
PUSH data: "STOP"
PULL data: N/A

The TAG device is used for writing ID-tags and it does not support PULL.

Once the TAG device has been pushed, the Reader will start a write attempt. A started write attempt continues until the write is completed. A WRIT event (see section 6.1.2 Write Complete) is sent when a write attempt is completed successfully.

Pushing the string "STOP" to the device stops a started write attempt; trying to perform a stop when there is not an ongoing write attempt will result in an error. It is necessary to stop an ongoing write attempt before a new write attempt is started.

To write the first writeable ID-tag that appears after this device has been pushed to, the mark field should be set to an asterisk ("*"). If it is necessary to address a specific ID-tag, the mark field should be set to the mark for that particular ID-tag, which is nine bytes of binary data.

The control field contains control characters that specifiy the format of an ID-tag and also controls properties of the write function. Control characters are specified in Table 25 through Table 29. If the control field is set to an asterisk, the format of the ID-tag is not changed, only the data is written to the ID-tag. It is primarily useful when it is necessary to just update the user data of an ID-tag.

The user data field is a string that specifies the user data to write to the ID-tag. If the size (length) of the user data is greater than the specified user data size, the user data field is truncated to fit the corresponding user data size. If the size of the provided user data is less than the specified user data size, the user data is padded with zero-bits.

Table 25   User data size control characters; only use one or none of these characters for a push

Control character	Description
"M"	Mini user data size, 14 bits or 2 bytes (46 bits or 6 bytes if no CRC is used)
"Q"	Quarter user data size, 154 bits or 20 bytes (186 bits or 24 bytes if no CRC is used)
"F"	Full user data size, 574 bits or 72 bytes (606 bits or 76 bytes if no CRC is used)

Table 26   Interval operating mode control characters; only use one or none of these characters for a push

Control character	Description
"R"	Randomly distributed interval lengths
"C"	Constant interval lengths

Table 27   Interval length control characters; only use one or none of these characters for a push

Control character	Description
"0"	Continuous operation, which means no idle period.
"4"	Four intervals
"8"	Eight intervals
"6"	16 intervals

Table 28   Speed control characters; only use one or none of these characters for a push

Control character	Description
"H"	High speed, 16 kbps
"L"	Low speed, 4 kbps

Table 29   Write function control characters

Control character	Description
"D"	Disable automatic user data CRC generation, should be used with caution. It is strongly recommended to use automatic CRC generation for most applications.

Note that if the automatic CRC generation is disabled and an ID-tag is successfully written, that ID-tag cannot be read by a Reader unless the variable TAG_DATACRC is cleared (set to "OFF"). Not using the CRC requires that the client is responsible for performing necessary ID-tag data integrity checks.

Table 30   Example of formatting a tag to quarter memory with constant continuous intervals and high speed (QC0H), and write the string "ABCDEFG" to the user data memory.

Client message	PUSHTAG *;QC0H;ABCDEFG\n
Server response	RPLYPUSH00\n

Note that the string in the example above is not long enough to fill the user data memory and consequently it will be padded with zero-bits. If there is an ongoing write attempt, the server response will contain an error message.

Table 31   Example of stopping an ongoing write attempt.

Client message	PUSHTAG STOP\n
Server response	RPLYPUSH00\n

Note that if there is no ongoing write attempt while stopping, the server response contains an error message.

5.1.2 Flush ID-tag Filter

DID: "FLSH"
PUSH data: Empty
PULL data: N/A

Pushing to the device FLSH will flush the ID-tag filter, resulting in that the ID-tag filter will be emptied on all ID-tags residing in filter. This device doesn't require any additional data.

Table 32   Example of flushing the ID-tag filter.

Client message	PUSHFLSH\n
Server response	RPLYFLSH00\n

5.2 Wiegand and Mag-stripe

The Reader supports Wiegand and Mag-stripe communication over one common physical interface and these are accessed using the WIEG and MAGS devices. These two devices do not support PULL, due to the fact that these interfaces are unidirectional.

5.2.1 Wiegand

DID: "WIEG"
PUSH data: <skip bits>;<data>
PULL data: N/A

Pushing to the WIEG device sends raw data to the Wiegand interface. Note that this device gives raw access to the Wiegand interface. Parity calculation and zero padding must be handled by the client.

The skip bits field specifies how many of the least significant bits in the last byte that is not used. Sending just four bits to the Wiegand interface should for instance require one byte of data and that the skip bits field is set to four (see Table 34).

The data field is the information to send. The most significant bit in the first data character is sent first.

Table 33   Example of sending the string "ABCDE" to the Wiegand interface

Client message	PUSHWIEG0;ABCDE\n
Server response	RPLYPUSH00\n

Setting the first field to zero, as in the example above, means that all information from the last byte is sent.

Table 34   Example of sending four bits "1000" (0x8) to the Wiegand interface.

Client message	PUSHWIEG4;%80\n
Server response	RPLYPUSH00\n

5.2.2 Mag-stripe

DID: "MAGS"
PUSH data: <skip bits>;<data>
PULL data: N/A

Pushing to the MAGS device sends raw data to the Mag-stripe interface. Note that this device gives raw access to the Mag-stripe interface. Parity calculation and zero padding must be handled by the client.

The skip bits field specifies how many of the least significant bits in the last byte that is not used. Sending just four bits to the Mag-stripe interface should for instance require one byte of data and that the skip bits field is set to four.

The data field is the information to send. The most significant bit in the first data character is sent first.

Table 35   Example of sending the string "ABCDE" to the Mag-stripe interface

Client message	`PUSHMAGS0;ABCDE\n`
Server response	`RPLYPUSH00\n`

5.3   Peripherals

Peripheral devices can be controlled by PUSH messages.

5.3.1   Blink with Visual Indicators

DID: "BLNK"
PUSH data: <colour1>;<duration in hexadecimal>;<colour2>
PULL data: N/A

Pushing to the device BLNK will set visual indicators to the colour specified by the colour1 field, keep that colour for a time period specified by the duration field in milliseconds, and then set the visual indicators to the colour specified by the colour2 field.

The fields specifying colours can hold the same values as variable LED described in Table 23.

Table 36   Example of setting the visual indicators to red for 400 milliseconds (190 hexadecimal) and then turning off the visual indicators

Client message	`PUSHBLNKred;190;off\n`
Server response	`RPLYPUSH00\n`

Note that if the device BLNK has been successfully pushed according to the example above and the variable LED is set to green during the 400 milliseconds timeout, the BLNK is automatically aborted.

5.3.2   Beep with Buzzer

DID: "BEEP"
PUSH data: <duration in hexadecimal>
PULL data: N/A

Pushing to the device BEEP results in beeping the buzzer. The length of the beep is specified by the duration field in milliseconds.

If the state of the buzzer is on before this PUSH message is sent, the only resulting action is that the buzzer is turned off after the given duration.

Table 37   Example of beeping the buzzer for 2.5 seconds (2500 ms, 9C4 hexadecimal)

Client message	`PUSHBEEP9C4\n`
Server response	`RPLYPUSH00\n`

Note that if the device BEEP has been successfully pushed according to the example above and the variable BUZZER is set to off during the 2500 ms timeout, the BEEP is automatically aborted.

5.4   Inputs and Outputs

Inputs and outputs are controlled by PUSH and PULL messages.

5.4.1   Open Relay

DID: "GATE"
PUSH data: <duration in hexadecimal>
PULL data: N/A

Pushing to the device GATE will open the relay for the period of time specified by the duration field in milliseconds. If the state of the relay is open before this PUSH message is sent, the only resulting action is that the relay is closed after the given duration.

Table 38   Example of opening the relay for 5 seconds (5000 ms, 1388 hexadecimal)

Client message	`PUSHGATE1388\n`
Server response	`RPLYPUSH00\n`

Note that if the device GATE has been successfully pushed according to the example above and the variable RELAY is set to off during the 5000 ms timeout, the GATE is automatically aborted.

5.4.2   Outputs

DID: "OUTP"
PUSH data: <output>=<value>
PUSH data: "ALL"=<mask>
PULL data: <mask>

Push the OUTP device to control the state of the outputs.

It is possible to address and set a specific output using the corresponding output string as specified in the table below. The value field can hold the value "0" or "1", where "1" is equivalent to on or high, and is the value to set to an addressed output.

It is also possible to address all outputs at the same time by setting the output field to "ALL" and the value field to a bit mask. The mask is a two-character hexadecimal bit mask. Each bit in the mask corresponds to the state of a certain pin as specified in the table below. If a bit is set in the mask the corresponding output is set.

Pulling the OUTP device returns a mask field, in which each bit corresponds to the state of a specific output, as specified in the table below.

Note that some pins are not included when addressing "ALL". Bit numbers marked with an asterisk (*) cannot be set using push to all outputs, they can only be pulled.

Table 39   Outputs, pin-names, and bit number definitions

Output	<output>	Bit number in <mask>
Output 1	"OUT1"	0, least significant bit in the output mask field
Output 2	"OUT2"	1
Expansion interface output 1	"EXPOUT1"	2
Expansion interface output 2	"EXPOUT2"	3
Wiegand output clock	"WICLK"	4*
Wiegand output data	"WIDATA"	5*
Wiegand output load	"WILOAD"	6*

Table 40   Example of pulling output for current state and all outputs are set to 0 (mask is "00")

Client message	PULLOUTP\n
Server response	RPLYPULL00OUTP00\n

Table 41   Example of setting expansion interface output 2 to value 1

Client message	PUSHOUTPEXPOUT2=1\n
Server response	RPLYPUSH00\n

Table 42   Example of setting outputs 1 and 2 to value 1 (mask is "03", which corresponds to the hexadecimal value 0x03 and is equivalent to the bit sequence 11)

Client message	PUSHOUTPALL=03\n
Server response	RPLYPUSH00\n

5.4.3   Inputs

DID: "INPT"
PUSH data: N/A
PULL data: <mask>

Pull the INPT device to retrieve the current state of the inputs.

The PULL reply from the server contains the mask field, which is a two-character hexadecimal bit mask. Each bit corresponds to the state of a specific input, as specified in the table below.

Table 43 Inputs and bit number definitions

Input	<input>	Bit number
Input 1	"INPUT1"	0*, least significant bit in the mask field
Input 2	"INPUT2"	1*
Input 3	"INPUT3"	2*
Expansion interface input 1	-	3
Expansion interface input 1	-	4
Expansion interface interrupt input	"EXP"	5*

Bit numbers marked with an asterisk (*) cannot be configured to send an input change event, see section 5.4.4 Input Event Mask and 6.2.2 Input Change.

Table 44 Example of retrieving the current state of the inputs, the mask value "00" states that no inputs are driven high

Client message	PULLINPT\n
Server response	RPLYPULL00INPT00\n

5.4.4 Input Event Mask

DID: "INPM"
PUSH data: <mask>
PULL data: <mask>

It is possible to specify the inputs that should send an input change event, see section 6.2.2 Input Change. The inputs that can send an event are specified in Table 43.

If a bit in the mask is set to 1, the corresponding input will send a software event when changing state.

Table 45 Example of pushing an input event mask stating that only input 3 will send an event, which corresponds to setting the bit number 2 to 1

Client message	PUSHINPM04\n
Server response	RPLYPUSH00\n

Table 46 Example of retrieving the current input event mask after the previous example

Client message	PULLINPM\n
Server response	RPLYPULL00INPM04\n

6   Events

Events are sent when there is something to report. All events follow the same message format as specified in section 3.2.3 EVNT: Event.

6.1   ID-tag

Events can be sent when an ID-tag is read and when a write attempt is completed.

6.1.1   ID-tag Read

EID: "TAG "
Event data R/O: \<mark>\<status>
Event data R/W: \<mark>\<control>\<user data>\<status>

Note that the fourth character in this EID is a space character.

Read-only ID-tags, such as a MarkTag, don't have user data. Read and writable ID-tags, such as a ScriptTag, have user data and a control field that specify the user memory size and the mode of the ID-tag. Both ID-tag types have a status field that provides information about the battery level.

Table 47   Example of reading MarkTag with ID 11478318.

Server message	`EVNTTAG 20070118143420957%04%02%BC%94%BA%15%E3`
	`%AA%08%00%00\n`

Table 48   Example of reading ScriptTag with ID 01150794, quarter size user data memory set to "abcdefghijklmnop"; note that the user data is padded with zero-bits

Server message	`EVNTTAG 20070129143053615%00%00F%3D+%B5%A3%98`
	`%AE@abcdefghijklmnop%00%00%00+%E5%1F%0E%CF%9F`
	`%0F`

Note that the event data must be un-escaped before it is used and before any sizes and lengths specified in this section (or referred to from this section) are correct.

The ID-tag event data originate from a MarkTag if the length of the event data is less than 12 bytes, otherwise it should be considered a readable and writable ScriptTag.

Converting the event data to an 8-digit decimal number (removing embedded CRC information and so forth) is done with the following formula (written in ANSI C syntax), which apply to both MarkTag and ScriptTag:

```
id = ((data[1] & 0x3f) << 22) | (data[2] << 14) |
     (data[3] << 6) | ((data[4] & 0xfc) >> 2);
```

Where data[0] correspond to the first event data byte, data[1] to the second event data byte, data[2] to the third, and so forth.

For a MarkTag, the status field indicates the current battery level. If status is all ones, the battery level is low. Converting the event data to a status value is done with the following formula:

```
status = ((data[8] << 6) | (data[9] >> 2)) & 0xfe;
```

For ScriptTag, the control field state the current mode of the ID-tag as specified in the table below. Converting the event data to a control value is done with the following formula:

```
control = ((data[8] << 6) | (data[9] >> 2)) & 0xfe;
```

Table 49   Bit-definitions for control field (where bit 0 is the least-significant bit)

Bit	Description
7	ID-tag data speed (0 = low data speed, 1 = high data speed)
6	Interval mode (0 = fixed intervals, 1 = randomly distributed intervals)
5	Operation (0 = continuous operation, 1 = intermittent operation)
4:3	The combination of bits 4 and 3 define the user data memory size (1:1 = N/A, 0:1 = full memory, 1:0 = quarter memory, 0:0 = mini memory)
2:1	The combination of bits 2 and 1 define the interval length (0:1 = 1:1 = 16 intervals, 1:0 = eight intervals, 0:0 = four intervals)
0	Don't care

The ID-tag user data starts at the eleventh byte of the ID-tag event data, or data[10] using the same nomenclature as earlier. The length of the event data for a ScriptTag depends on the user data size (see Table 25).

Note that the user data contain a 32 bit user data CRC if the ScriptTag was written with automatic CRC generation enabled (see section 5.1.1 Writing ID-tag). ID-tags with bad CRC are automatically discarded (if TAG_DATACRC is "ON") so there is no need to verify the CRC.

Removing the CRC from the user data involve masking the last user data byte. For mini, quarter and full size respectively:

```
last_m = data[11] & 0xfc; /* beginning data[10] */
last_q = data[29] & 0xc0; /* beginning data[10] to data[28] */
last_f = data[81] & 0xfe; /* beginning data[10] to data[80] */
```

Since status information for a ScriptTag is appended to the user data, its position also depends on the user data size. Extracting the status value for a ScriptTag is done with the following formula, for mini, quarter and full size respectively:

```
status_m = ((data[15] << 6) | (data[16] >> 2)) & 0xfe;
status_q = ((data[33] << 2) | (data[34] >> 6)) & 0xfe;
status_f = ((data[85] << 6) | (data[86] >> 2)) & 0xfe;
```

The most significant bit in the status information (bit number 7) states the current battery level. If this bit is 1 the battery level is low. Bits 6 to 0 are don't care.

There is an example in the Appendix that shows how to decode and extract information from an ID-tag read event, see section 10.3.

6.1.2 Write Complete

EID: "WRIT"
Event data: Empty

The server sends a write complete event after a write attempt is completed. Note that there is no time limit regarding how long the write attempt can continue.

Table 50   Example of a successful write attempt completed at 2007-01-26 10:03:28.654

Server message	`EVNTWRIT20070126100328654\n`

6.2 Inputs

Events can be sent when tampering switch and inputs change state.

6.2.1 Tamper Switch Change

EID: "TMPR"
Event data: TAMPER=<value>

A tampering switch event is sent when the tampering switch changes state. That is when the tampering switch goes from closed to open, or from open to closed.

The value field specifies the new state of the tampering switch. "0" corresponds to open and "1" to closed.

Table 51   Example of opening the tamper switch at time: 2007-01-26 10:03:28.654.

Server message	`EVNTTMPR20070126100328654TAMPER=0\n`

6.2.2 Input Change

EID: "INPT"
Event data: <input>=<value>

The input field specifies the input that has changed state. Possible inputs are specified in Table 43.

The value field contains the new value of the input, either "0" or "1".

Table 52   Example of input 2 going from low to high at 2007-01-27 23:05:12.200

Server message	`EVNTINPT20070127230512200INPUT2=1\n`

6.3 Special Features

This section describes special features.

6.3.1 Accurate Position

EID: "APOS"
Event data: Empty

When an accurate position has been determined the APOS event is immediately sent. It states that the Reader passed on top of an ID-tag (or the opposite, ID-tag passed on top of a Reader) exactly APOS_TIMEOUT milliseconds ago.

This event does not contain any event data.

Table 53   Example of accurate position found at 2007-01-26 10:03:28.654

Server message	`EVNTAPOS20070126100328654\n`

7　Contact

For any further inquiries, please contact TagMaster AB.

7.1　Technical Support

Phone: + 46 8 632 19 50
Fax: +46 8 750 53 62
E-mail: support@tagmaster.com

7.2　Office

TagMaster AB
Kronborgsgränd 1
S-164 87 KISTA, Sweden

Phone: +46 8 632 19 50
Fax: +46 8 750 53 62
E-mail: sales@tagmaster.com
Web: www.tagmaster.com

8   Glossary

\n	Newline character
APOS	Accurate positioning
client	An application connected to a tagd server, such as a Reader application or a netcat connection. A client can access and control the resources of the tagd server.
CRC	Cyclic Redundancy Check, a type of hash function used to produce a checksum, in order to detect errors in transmission or storage
Cygwin	Cygwin is a Linux-like environment for Windows
DID	Device ID
EID	Event ID
ID-tag	ID-carrier in the TagMaster system, which is readable and writable via the radio interface.
ISO 8601	The International Standard for the representation of dates and times
Mag-stripe	Card reading protocol used for reading magnetic stripe cards
MarkTag	Read-only TagMaster ID-tag
MID	Message ID
N/A	No applicable
netcat	Network utility for reading from and writing to network connections
Reader	TagMaster GEN4 Reader
RFID	Radio Frequency Identification
ScriptTag	Readable and writable TagMaster ID-tag
server	*see* tagd
tagd	The TagMaster daemon, a server residing in each TagMaster Reader that implements the TAGP protocol. tagd allows clients to connect and access the Reader resources.
taglib	The TagMaster software Library
TAGP	The TagMaster Protocol
Wiegand	Trade name for a technology used in card readers and sensors, particularly for access control applications

9   References

[1]   *SDK User's Manual*
Doc no. 06-195

[2]   *taglib Software Library Specification*
Doc no. 05-249

[3]   *GEN4 Reader User's Manual*
Doc no. 06-118

10   Appendix

This section provides some practical information regarding the TAGP protocol.

10.1 Manual Interaction with tagd

The simplicity and the readability of the TAGP protocol makes it possible to manually communicate with a tagd server using TAGP.

10.1.1 Network Utility Software

Manual interaction requires some sort of network utility that allows a user to connect to a tagd server with a TCP/IP socket, send information on the socket and receive information from the socket. There are several standard utilities for Windows, Linux, and UNIX. A network utility called netcat, is recommended.

Table 54 Network utilities and clients

Name	Description
netcat	Network utility for reading from and writing to network connections on either TCP or UDP. netcat is usually installed by default on most Unix and Linux systems. On Windows systems netcat may require installation. It is also possible to use netcat within the Cygwin environment <http://www.cygwin.com>. For more information concerning netcat, see the official project homepage <http://netcat.sourceforge.net>.
telnet	Standard network protocol for Internet and LAN connections. Telnet is not ideal for manual TAGP communication because it performs some visual formatting of the TAGP protocol that makes it more difficult to use. The obvious advantage is that the telnet client is usually installed by default on all systems, including Windows, Unix and Linux.

10.1.2 Connect to tagd

In order to connect to tagd, the IP address and the TCP port at which the tagd server is listening must be known. The default port is 9999 but tagd can be configured to listen to another port.

The following example shows how to use netcat to connect to a tagd server with IP address 193.15.235.111 listening at the default port.

Table 55 Example session

Row	Terminal in- and output (user input in bold)
0	joro@BlackBox:~$ **nc 193.15.235.111 9999**
1	**HELOTAGP/1.1**
2	RPLYHELO00
3	**SET NAME=terminator**
4	RPLYSET 00OK
5	**GET NAME**
6	RPLYGET 00NAME=terminator
7	**GET LED**
8	RPLYGET 00LED=green
9	**SET LED=off**
10	RPLYSET 00OK
11	**PUSHBLNKred;1000;off**
12	RPLYPUSH00
13	**GET FILTER_TYPE**
14	RPLYGET 00FILTER_TYPE=off
15	**SET FREQUENCY=24510**
16	EVNTTAG 20070129111953473%00%F5%9C%F8%A3%8D'P%00%00
17	RPLYSET 00
18	EVNTTAG 20070129111953483%00%F5%9C%F8%A3%8D'P%00%00
19	EVNTTAG 20070129111953493%00%F5%9C%F8%A3%8D'P%00%00
20	EVNTTAG 20070129111953503%00%F5%9C%F8%A3%8D'P%00%00
21	EVNTTAG 20070129111953513%00%F5%9C%F8%A3%8D'P%00%00
22	**SET FILTER_TYPE=periodic**
23	RPLYSET 00OK
24	**SET FILTER_TIMEOUT=500**
25	RPLYSET 00OK
26	EVNTTAG 20070129112114933%00%F5%9C%F8%A3%8D'P%00%00
27	EVNTTAG 20070129112115434%00%F5%9C%F8%A3%8D'P%00%00
28	EVNTTAG 20070129112115934%00%F5%9C%F8%A3%8D'P%00%00
29	EVNTTAG 20070129112116435%00%F5%9C%F8%A3%8D'P%00%00
30	EVNTTAG 20070129112116935%00%F5%9C%F8%A3%8D'P%00%00
31	EVNTTAG 20070129112117435%00%F5%9C%F8%A3%8D'P%00%00
32	EVNTTMPR20070129112146144TAMPER=1
33	EVNTTMPR20070129112146344TAMPER=0

At row 0 netcat is started from a Linux prompt with the IP address and port as input parameters (the netcat utility is most likely just called "nc").

If netcat is able to establish a connection to the addressed tagd server, a raw TCP/IP socket is opened.

At row 1, the communication is initialised by sending the HELO message and the protocol version. The server response at row 2 states that the initialisation was successful.

At row 3 to 14 some variables are accessed using SET and GET messages. Note that a GET or a SET is always followed by a RPLY from the server before the next GET or SET message is sent.

Events are sent immediately as they occur. At row 15 the variable FREQUENCY is set. Before the server responds to that SET message (row 17) an ID-tag event was received (row 16). It implies that before setting the new frequency was completed the Reader read an ID-tag.

Because the ID-tag filter is off (as stated at row 14) ID-tag events are not filtered, which can be seen on the time stamps in the ID-tag events at row 18 to row 21.

When enabling the tag filter (see row 22 to row 25) the ID-tag events are sent periodically as seen on the time stamps at row 26 to row 31 (each time stamp is separated with at least 500 ms).

Note that netcat is available on the Reader and can be used to interact with tagd when logged on over the service interface. The following command can be used:

```
tag:/$ nc localhost 9999
```

10.2 Considerations using TAGP

A Reader application is a software application that runs inside the Reader. Reader applications can be based on taglib or use TAGP directly.

From a tagd perspective, Reader applications are also clients. tagd makes no difference on whether a client is local (for instance a Reader application) or external (for instance a netcat client).

Multiple clients connected to the same tagd server share the same resources; those are the same hardware, functions and so forth.

Variables in the global namespace and devices are accessible from all clients (if they are not locked) and can be set without the other client knowing it. For instance if client A sets the frequency to 2.454 GHz, client B could set the frequency to 2.452 GHz without client A noticing that change.

10.3 Decoding ID-tag Events

This section shows how to decode ID-tag events into useful ID-tag information. The examples are illustrated using ANSI C like pseudo code.

The ID-tag event data field is escaped and it must be un-escaped before the event data can be used.

For example the event data "%00%00F%3D+%B5" corresponds to the byte array {0x00, 0x00, 0x46, 0x3d, 0x2b, 0xb5}. Some characters are escaped and some are not. For instance null is escaped and sent as "%00" and character "F" is not required to be escaped and is sent as clear text (see section 2.2.1 Escaped Characters).

10.3.1 MarkTag

The following ID-tag read event will be decoded in this section:

```
EVNTTAG 20070118143420957%04%02%BC%94%BA%15%E3%AA%08%00%00\n
```

The first four bytes in the message is the MID, which states that this is an event message "EVNT". The next four bytes is the EID, which explicitly state that this is an ID-tag read event "TAG ". The following 17 bytes is the time stamp at which this ID-tag was read, it reads: 2007-01-18 14:34:20.957.

The remaining bytes are the ID-tag event data:

```
%04%02%BC%94%BA%15%E3%AA%08%00%00
```

After un-escaping, the event data correspond to the following data array (where the first index 0 holds value 4):

```
data = {0x4, 0x2, 0xBC, 0x94, 0xBA, 0x15, 0xE3, 0xAA, 0x8,
        0x0, 0x0}
```

The size of the data is 11 bytes, which is less than 12 bytes and hence this ID-tag is a MarkTag. Using the formula specified in section 6.1.1, the ID for the ID-tag is calculated as:

```
id = ((0x2 & 0x3F) << 22) | (0xBC << 14) | (0x94 << 6) |
     ((0xBA & 0xFC) >> 2) =
   = 0x00AF252E =
   = 11478318
```

Using the formula for status information, the status is calculated as:

```
status = ((0x8 << 6) | (0x0 >> 2)) & 0xfe = 0x0
```

From the information above the ID-tag from which the event originates has ID 11478318 and a high battery level.

10.3.2 ScriptTag

The following ID-tag read event will be decoded in this section:

```
EVNTTAG 20070129143053615%00%00F%3D+%B5%A3%98%AE@abcdefghijklmno
p%00%00%00+%E5%1F%0E%CF%9F%0F\n
```

As in the previous section the first 25 bytes specify the message, the event type and the time at which the event occurred. The remaining event data bytes are:

```
%00%00F%3D+%B5%A3%98%AE@abcdefghijklmnop%00%00%00+%E5%1F%0E%CF%9
F%0F
```

After un-escaping, the event data correspond to the following data array (where 'a' is equal to 0x61 and so forth):

```
data = {0x0, 0x0, 'F', 0x3D, '+', 0xB5, 0xA3, 0x98, 0xAE, '@',
        'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k',
```

```
               'l', 'm', 'n', 'o', 'p', 0x0, 0x0, 0x0, '+', 0xE5, 0x1F,
               0x0E, 0xCF, 0x9F, 0x0F}
```

The size of the data is greater than 12 bytes and hence this ID-tag is a ScriptTag. The ID and the control field for the ID-tag are calculated as:

```
id = ((0x0 & 0x3F) << 22) | ('F' << 14) | (0x3D << 6) |
     (('+' & 0xFC) >> 2) =
   = ((0x0 & 0x3F) << 22) | (0x46 << 14) | (0x3D << 6) |
     ((0x2B & 0xFC) >> 2) =
   = 0x00118F4A =
   = 01150794


control = ((0xAE << 6) | ('@' >> 2)) & 0xfE =
        = ((0xAE << 6) | (0x40 >> 2)) & 0xfE =
        = 0x90 =
        = 0b'10010000' /* binary representation */
```

Bit 7 and bit 4 are set. According to Table 49 this means that the ID-tag: uses high data speed, fixed intervals, continuous operation, quarter memory and four intervals. The mode can be written with the corresponding code: "QC4H".

Knowing that the ID-tag is in quarter memory, the status can be calculated as:

```
status = ((0xCF << 2) | (0x9F >> 6)) & 0xfe = 0x3E
```

The useful user data (excluding CRC) is index 10 to 29 in the data array, where the last index should be masked to remove CRC:

```
data[29] = data[29] & 0xC0 =
         = '+' & 0xC0 =
         = 0x2B & 0xC0 =
         = 0x0
```

From the information above the ID-tag from which the event originates has ID 01150794, high battery level, set to mode "QC4H", and has the following user data:

```
user_data = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j',
             'k', 'l', 'm', 'n', 'o', 'p', 0x0, 0x0, 0x0, 0x0}
```