

***taglib Software Library Specification***  
**v1.4.0**

<b>Revision</b>	<b>Date</b>	<b>Comment</b>
02	2007-10-16	Updated for System Software 1.4.0

**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.

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Design Concepts</b>	<b>5</b>
2.1	TAG Handle.....	5
2.2	TAGHUB .....	6
2.3	Event .....	8
2.4	Event Callbacks.....	9
2.5	Talk Messages .....	10
<b>3</b>	<b>Functions Overview</b>	<b>11</b>
3.1	tagerr Return Codes.....	11
3.2	Library Functions.....	12
3.3	TAGHUB Functions.....	12
3.4	TAG Handle Functions .....	17
3.5	Client Functions.....	18
3.6	Radio-related Functions .....	20
3.7	ID-tag Functions .....	23
3.8	Reader Indicator Functions .....	27
3.9	Reader I/O Functions .....	28
3.10	Reader System Functions .....	31
3.11	Communication Interfaces Functions .....	32
3.12	Special Functions .....	33
<b>4</b>	<b>Examples</b>	<b>35</b>
4.1	First Application.....	35
4.2	Using Callbacks.....	36
4.3	Adding an RS232 File Descriptor .....	38
4.4	Talk Messages .....	40
4.5	Multiple Readers .....	42
4.6	Using Inputs and Outputs.....	43
<b>5</b>	<b>Contact</b>	<b>47</b>
5.1	Technical Support .....	47
5.2	Office .....	47
<b>6</b>	<b>Glossary</b>	<b>48</b>
<b>7</b>	<b>References</b>	<b>49</b>

# 1 Introduction

taglib is a software library in C, which presents an interface for interacting with TagMaster's GEN4 Readers. For a software architecture overview, see GEN4 Reader User's Manual [1]

taglib provides easy to use function calls that perform tasks such as reading information from ID-tags, turning the indicators on or off, and configuring Reader settings.

taglib is more than just an interface for controlling properties of a Reader; it provides a framework to manage several Readers simultaneously from one application process. taglib implements a communication protocol called TAGP. For information about TAGP, see TAGP Protocol Specification [2].

taglib supports integration of file descriptors, which allows for versatile event-driven software development.

taglib is flexible and can be used together with other existing libraries, for example Linux APIs used for accessing serial interfaces, USB interfaces and so forth.

## 2 Design Concepts

taglib is a network oriented software library and supports development of distributed applications using TAGP. It has been designed with the aim to support fast, easy, and reliable development of both Reader application software and host application software.

Reader application software controls the local Reader. Reader application software can also control one or several remotely connected Readers. Host application software resides in an external system, for instance a workstation or a server, which acts as master of a Reader or a set of Readers.

Writing code with taglib is preferably done in event-driven fashion, as opposed to using polling. Polling is resource-demanding and therefore considered bad design practice when writing code with taglib.

The Reader has, like other embedded systems, limited resources, which must be taken into account when developing Reader application software. The Reader has two types of memory:

- 16 MB Flash memory accessed via the Journaling Flash File System Version 2 (JFFS2)
- 32 MB volatile SDRAM

The Reader is built around an AT91RM9200 MCU with an ARM9 CPU core running at 180 MHz (200 MIPS).

### 2.1 TAG Handle

A TAG handle is simply a reference to a Reader. Readers have an application abstraction layer that software designers utilise for application development. The abstraction layer is a high-level daemon process called TagMaster Daemon (tagd). The daemon process is used to control Reader properties such as frequency channel, buzzer, and relay state.

From an application designer's point of view, tagd is a server handling the resources of the Reader. The application is a client connected to the tagd server requesting access to the resources of the Reader. Several clients, or peers, can be connected to the same tagd server. Only one instance of tagd can run on each physical Reader.

A TAG handle specifies the tagd server and which socket to use for communication. Note that TAG handles cannot report events; they are used only to address Readers in function calls. A concept called TAGHUB is used to report events from Readers to the application.

The creation of handles is done with the `tag_reader_init()` function call (see section 3.4.1). The IP address and the port number of a tagd server are required to create a TAG handle to that specific tagd server. A TCP/IP socket to the tagd server is created and the function returns an initialised handle, which can be used to set Reader properties, send messages, and so forth.

The use of TAG handles is flexible. An application can have several TAG handles to different Readers and if required, more than one handle to the same Reader.

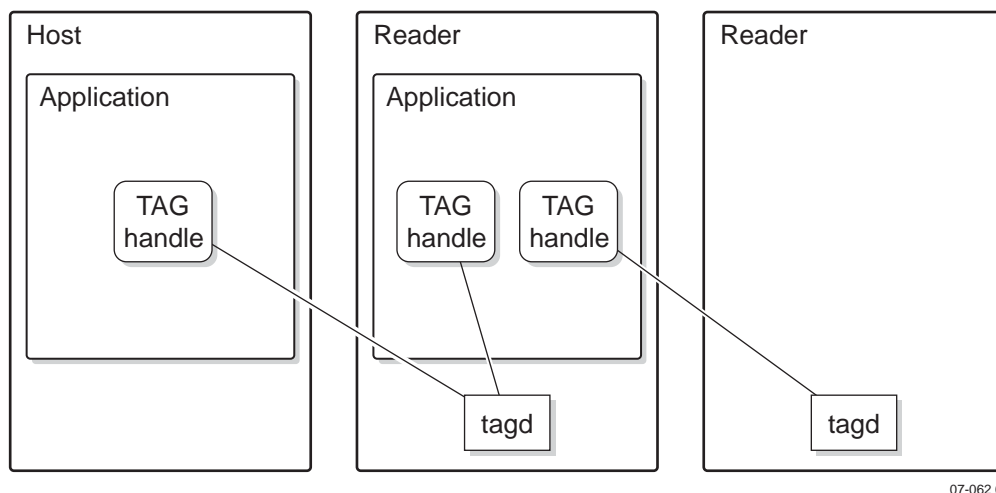


Figure 1 TAG handle overview

The tagd server can be configured to accept a maximum number of connections. However, taglib has no built-in restriction regarding the number of TAG handles an application can have and no built-in restriction regarding simultaneous connections to the same tagd server. However, there are of course practical limitations concerning the number of TAG handles and simultaneous connections.

## 2.2 TAGHUB

A TAGHUB extends the use of a TAG handle. A TAG handle is only a reference to a tagd connection and is passed as argument to functions that for instance set the colour of the visual indicator. The TAG handle cannot report events back to the application, so in order to receive data from a Reader, the TAG handle must be connected to a TAGHUB.

TAGHUB is a central concept when developing software with taglib. Visualise the TAGHUB as a communication hub to which it is possible to connect peers.

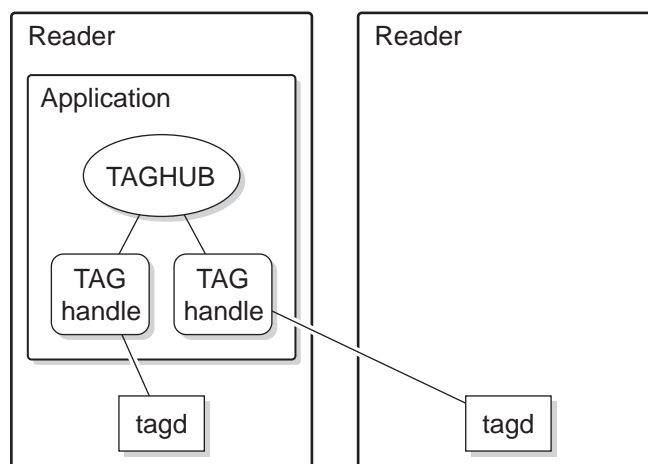


Figure 2 TAGHUB overview

The TAG handle is automatically connected to a TAGHUB during the TAG handle initialisation. A TAG handle cannot be created if there is no TAGHUB.

The TAGHUB buffers events from a Reader until the application is ready to handle the events.

The TAGHUB facilitates simultaneous connections of numerous clients. The TAGHUB also make it possible to integrate connections to serial interfaces and network sockets.

The following are some general design concepts regarding TAGHUBS:

- First create a single TAGHUB and get a handle returned to that, using `tag_hub_init()`.
- Then add communication peers to that hub. A peer can be a Reader or a file descriptor.
- Peers can be removed from the hub at any time.
- Events are handled on a per-TAGHUB basis.
- When communicating with Readers, functions calls are addressed directly to a specific peer and not through the hub.

A TAGHUB might seem complex and unnecessary presupposed that the TAG handle could directly be polled for events. But polling use excessive system resources and is slow, especially when connecting tens or hundreds of Readers.

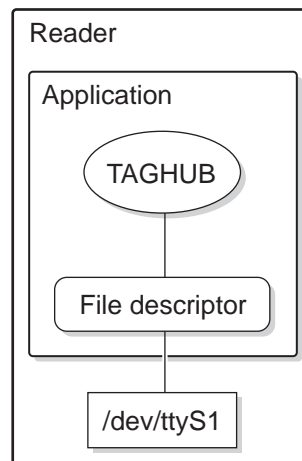
### 2.2.1 TAGHUB and File Descriptors

A file descriptor is basically a reference to an open file. Not only ordinary text files are treated as files by Linux, but also character devices such as serial interfaces are considered to be files.

A file must be opened to be read, written, or modified. When a file is opened, the system uses a number to keep track of that open file. This number is the file descriptor, typically a non-negative integer.

Status changes on file descriptors can be monitored using a TAGHUB. A file descriptor can be added to a TAGHUB with the taglib function `tag_hub_add_fd()`, see section 3.3.8. A file descriptor can be removed from a TAGHUB with the taglib function `tag_hub_del_fd()`, see section 3.3.9.

Status changes of file descriptors are reported as taglib events. The accesses are still done with standard read and write system calls and the status of the file descriptor can be polled.



07-064 01

Figure 3 TAGHUB and file descriptor overview

A TAGHUB can be configured so that it notifies if an attached file descriptor is readable or writable, or both.

## 2.2.2 TAGHUB Pending

The `select()` Linux system call is used when waiting for a number of file descriptors to change status, for example input data has become available or receiver has become ready to receive data. The system call `select()` is considerably faster than implementing a loop that uses `read()` on each file descriptor.

The system call `select()` will block until a file descriptor changes status, a signal is received, or a timeout period expires. With `select()` there is no need to `fork()` or use threads; it is possible to create powerful single-threaded servers.

The TAGHUB concept adopts `select()` in the form of a taglib function called `tag_hub_pending()`, which basically is a `select()` system call.

The `tag_hub_pending()` function takes a TAGHUB and a timeout as arguments, see section 3.3.5. The function blocks until an event has been received or the timeout expires.

An event driven application that performs actions when required, is favourable when dealing with many TAG handles and file descriptors. Dealing with only one TAGHUB leads to a less complex and faster application.

Note that designing an application that polls a TAGHUB and file descriptor is possible, but not recommended.

## 2.3 Event

Events are used to notify the application that something has occurred. The `tag_hub_pending()` function waits for events to occur. When `tag_hub_pending()` returns, there is typically an event to handle, unless there was a timeout.

The `tag_hub_poll()` function fetches the first event from the queue, and as the name implies, this function can be used to periodically poll the queue.

The taglib software library defines a number of event types:

- ID-tag read
- Tampering switch changed state
- Input changed state
- File descriptor status changed
- Talk message received
- ID-tag write attempt completed
- Errors

Each event type is associated with an event data structure, which holds information about the event such as time, type, TAG handle, and data.

The following pseudo-code exemplifies the handling of ID-tag events, which are events that occur when an ID-tag has been read.

```
main()
  H = tag_hub_init()
  R = tag_reader_init(H, "localhost", 9999)
  loop
    /* wait for event, set timeout to 10 seconds */
    tag_hub_pending(H, 10000)
    /* fetch event */
    E = tag_hub_poll(H)
    if E.type = tag_event
      /* it is a tag */
    end if
  end loop
end main
```

A TAGHUB "H" is created and initialised, then a TAG handle "R" is created and initialised with a specified Reader.

The `tag_hub_pending()` function waits for an event and returns after the specified timeout or when an event occurs. The `tag_hub_poll()` function polls the event queue for events possibly residing in the queue. Information about an event is stored in an event data structure.

Event types not handled by the code are simply polled from the queue and discarded.

## 2.4 Event Callbacks

Using callbacks is an alternative approach when handling events. A callback is basically an event-handler, that is a piece of code that is executed each time a specific type of event occurs.

A callback is only called during the execution of a `tag_hub_pending()` function, so there is no need to consider race conditions or mutual exclusions.

The function `tag_hub_callback()` is used to specify what function to automatically call each time a specific type of event occurs. Each type of event can have a callback function. There is also a default callback that takes care of all events, except those already associated with a callback.

The main advantage with using callbacks is that the code can be made comprehensible and modular, which leads to easier code to read, less risk of errors, and simplifies code re-use.

The following pseudo code exemplifies the handling of ID-tag events using a callback.

```
main()
  H = tag_hub_init()
  R = tag_reader_init(H, "localhost", 9999)
  /* setup a tag event call-back */
  tag_hub_callback(H, tag_event, my_callback)
  /* never return, set timeout to 0 */
  tag_hub_pending(H, 0)
end main

my_callback(E)
  /* we got a tag */
end my_callback
```

Mixing callbacks with standard event handling is allowed.

## 2.5 Talk Messages

Talk messages allow two or more clients connected to the same tagd server, to exchange data.

Talk messages are useful for distributed applications and in situations when it is necessary to implement a custom application-to-application communication protocol.

Before a client application can send or receive talk messages it must comply with the following:

- Register an unique name using the `tag_reader_set_name()` function.
- Explicitly tell tagd that it wants to be available for talk messages using the `talk_reader_settalk()` function.

When a client application has sent a talk message, the message will appear at the receiving application as a talk event.

## 3 Functions Overview

This section describes the functions defined by taglib. taglib contains various categories of functions covering library-related functions, hub-related functions, and Reader-related functions.

Functions used to control Reader settings require a TAG handle as first argument.

taglib functions return a tagerr code if not explicitly stated otherwise. Other return values are typically received through function parameters.

**Note:** Discard the returned parameter values if the function returns an error code.

### 3.1 tagerr Return Codes

A taglib function typically returns TAGE\_OK (equal to zero) after normal execution. Error return codes (tagerr) are indicated by the negative value of the return code stated in the table below. For example, code -TAGE\_SET is returned if a set command failed.

Table 1 Return codes overview

Constant Name	Description
TAGE_OK	Normal execution
TAGE_OUT_OF_MEMORY	Out of memory
TAGE_RESOLVE_FAILED	Resolve failed
TAGE_SOCKET	socket() failed
TAGE_CONNECT	connect() failed
TAGE_HELLO	Got no reply or negative reply to the TAGP server connection setup message
TAGE_EMPTYQUEUE	No messages in the event queue, typically returned by tag_read()
TAGE_SET	Erroneous response or timeout from a set command
TAGE_DISCONNECT	tagd server disconnected
TAGE_TAGP	TAGP protocol syntax error
TAGE_RESPTIMEOUT	Timed out waiting for a TAGP confirmation message
TAGE_SEND	send() failed
TAGE_NEGATIVE	The server replied a negative response, denying the previous request or operation
TAGE_BADARG	A public function was given an illegal input argument
TAGE_NOCHANNEL	Requested frequency does not match a valid channel number
TAGE_RECURSIVE	Calling a taglib function recursively when not allowed
TAGE_BADFD	The file descriptor in a function argument could not be found or was bad for other reasons.

## 3.2 Library Functions

This subsection describes library-related functions, which are help functions that generally make application development easier.

### 3.2.1 tag\_libversion()

```
int tag_libversion(void)
```

The function returns the version number of taglib, which comprises three numbers; major, minor and micro. The version number is returned according to the following formula:

Version number = major × 10000 + minor × 100 + micro.

This function is useful when tag\_reader\_getsysteminfo() is not applicable, for instance when the application is not connected to a tagd server.

### 3.2.2 tag\_free()

```
void tag_free(void *data)
```

The function frees “data” returned by functions calls that allocate memory, such as the function tag\_reader\_getname().

### 3.2.3 tag\_marktoint()

```
int tag_marktoint(unsigned char *mark)
```

The function returns the 8-digit id code for the specified “mark”.

The size of the mark array must be at least five characters.

### 3.2.4 tag\_ismarkopen32()

```
bool tag_ismarkopen32(unsigned char *mark)
```

The function returns true or false depending on if the specified “mark” is an Open32 formatted ID-tag mark.

The size of the mark array must be at least five characters.

### 3.2.5 tag\_marktoopen32()

```
int tag_marktoopen32(unsigned char *mark)
```

The function returns the 32-bit hexadecimal Open32 id code for the specified “mark”.

The size of the mark array must be at least five characters.

## 3.3 TAGHUB Functions

This subsection describes TAGHUB control functions, which, for example, create and close hubs, add and remove file descriptors, manage events, and declare event types.

There are several types of events that can be reported, either by a Reader or by a file descriptor. Each type of event generates information that is stored in different types of data structures.

### 3.3.1 Event-specific structs

The data structures holding information about an event type are defined as follows:

#### Write attempt completed

```
struct writeevent {
    time_t secs;           /* time as number of seconds
                           since 1 jan 1970 */
    int usecs;            /* microsecond fraction */
};
```

#### Accurate position detected

```
struct positionevent {
    time_t secs;           /* time as number of seconds
                           since 1 jan 1970 */
    int usecs;            /* microsecond fraction */
};
```

#### Read an ID-tag

```
struct tagevent {
    time_t secs;           /* time as number of seconds
                           since 1 jan 1970 */
    int usecs;            /* microsecond fraction */
    unsigned char mark[9]; /* 70 bits */
    unsigned char ctrl[2]; /* 10 bits */
    unsigned char status; /* 7 bits */
    unsigned char data[160]; /* up to 1280 bits */
    size_t datalen;        /* number of used bytes in
                           data[] (including CRC) */
};
```

The *ctrl*, *status* and *data* fields contain data extracted from the tag info as received over TAGP, and that data is documented in the "TAGP Protocol Specification" section 6.1.1

#### Input Change

```
struct inputevent {
    time_t secs;           /* time as number of seconds
                           since 1 jan 1970 */
    int usecs;            /* microsecond fraction */
    int source;           /* input source */
    char value;           /* 1 or 0 */
};
```

#### Talk Message Received

```
struct talkevent {
    unsigned char from[33]; /* name of the sender */
    unsigned char data[1025]; /* data */
    size_t datalen;        /* number of bytes in 'data'
                           */
};
```

#### File Descriptor Readable, Writable or Both

```
struct fdevent {
    int fd;
```

```

    int action; /* bitmask using the same bits the
                tag_hub_add_fd() function uses in its
                'mode' argument. States whether the fd
                became readable, writable, or both */
};

```

### Errors

```

enum readererr {
    READERERR_NONE,          /* not an error */
    READERERR_DISCONNECTED, /* connection to reader was lost */
    READERERR_LAST          /* not an error, just a marker */
};

```

```

/* event, an error has occurred */
struct errorevent {
    enum readererr error; /* */
};

```

## 3.3.2 The Generic Event struct

The functions `tag_hub_poll()` and `tag_hub_callback()` are used to receive information about events, see section 3.3.6 and section 3.3.7. Both functions take an argument called “data”, which is filled in with information about an event.

The general event struct holds information about the type of the event (see Table 2) and a pointer to one of the different event type data structures described above.

Different event types are defined as constants.

Table 2 *TYPE\_definitions*

Constant Name	Description
TYPE_DEFAULT	Using this type makes the registered function called for all types of events not associated with their own callback; only usable with <code>tag_hub_callback()</code>
TYPE_TAG	ID-tag event
TYPE_TALK	Talk event
TYPE_FD	Externally set file descriptor
TYPE_INPUT	Input event
TYPE_TAMPER	Tamper switch event
TYPE_WRITTEN	ID-tag was written successfully
TYPE_POSITION	ID-tag position was established
TYPE_ERROR	An error occurred

The general event data structure is defined as follows:

```

struct event {
    int type; /* one of the types defined below */
    void *userp; /* pointer set with the tag_reader_init()
                 function that added the peer this data

```

```

                                originates from */

/*
 * The data union below is filled in/used depending on
 * what 'type' this event is:
 *
 * TYPE_TAG          - tag
 * TYPE_POSITION    - accurate position
 * TYPE_TALK        - talk
 * TYPE_FD          - fd
 * TYPE_INPUT       - input
 * TYPE_TAMPER      - tamper
 * TYPE_WRITTEN     - write
 * TYPE_ERROR       - error
 */
union {
    struct tagevent *tag;
    struct positionevent *position;
    struct inputevent *input;
    struct inputevent *tamper;
    struct talkevent *talk;
    struct fdevent *fd;
    struct writeevent *write;
    struct errorevent *error;
} data;

/*
 * The origin union below is filled in/used depending on
 * where this event originates from. If the event
 * is sent from a reader, the 'reader' is filled in and
 * if it is from a file descriptor the 'fd' is filled
 * in:
 *
 * TYPE_FD          - fd
 * TYPE_TAG         - reader
 * TYPE_POSITION    - reader
 * TYPE_TALK        - reader
 * TYPE_INPUT       - reader
 * TYPE_TAMPER      - reader
 * TYPE_WRITTEN     - reader
 * TYPE_ERROR       - reader
 */
union {
    TAG *reader;
    int fd;
} origin;
};

```

### 3.3.3 tag\_hub\_init()

```
int tag_hub_init(TAGHUB **hub)
```

The function creates and initialises a new TAGHUB.

The argument “hub” returns a reference to the TAGHUB object.

### 3.3.4 tag\_hub\_close()

```
void tag_hub_close(TAGHUB *hub)
```

The function closes the TAGHUB object specified by “hub”.

This function did not exist until taglib 1.3

### 3.3.5 tag\_hub\_pending()

```
int tag_hub_pending(TAGHUB *hub, int timeout)
```

The function waits “timeout” milliseconds for “hub” to report an event. The function waits forever if the argument “timeout” is 0. The function returns instantly if “timeout” is -1.

While waiting, callback functions associated with “hub” are called if an event occurs, presupposed that the type of the occurring event have a matching callback function.

The function returns a negative integer if an error occurs, zero if timed out or a positive integer if an event has occurred but that event is not associated to a callback function.

If tag\_hub\_pending() returns a positive integer, there is an event residing in the event queue waiting to be polled. If that event is not polled, the event will be discarded next time tag\_hub\_pending() is called.

**Note:** Recursive calls to this function are not allowed, for example tag\_hub\_poll cannot be called from within a callback function.

### 3.3.6 tag\_hub\_poll()

```
int tag_hub_poll(TAGHUB *hub, struct event *datap)
```

The function fetches and removes the first event from the event queue of “hub”. The argument “datap” is an event struct, which is filled with information about the event.

The function returns zero if the event was successfully stored in the event struct. The function returns the negative value of the tagerr code TAGE\_EMPTYQUEUE if the event queue is empty.

### 3.3.7 tag\_hub\_callback()

```
int tag_hub_callback(TAGHUB *hub, unsigned int type,  
    void (*func)(struct event *data, void *userp),  
    void *userp)
```

The function sets a callback function for “hub” which is automatically called if an event matching “type” occurs in the TAGHUB.

A callback function is only invoked through tag\_hub\_pending(). By setting several callbacks covering all types of events, the application is made fully event based. The function tag\_hub\_pending() can be set to wait forever and if all event types are associated with callback functions, tag\_hub\_pending() will never return.

The callback function must have a prototype matching:

```
void func(struct event *data, void *userp)
```

When an event type matching the callback occurs and the callback function is called, it receives an event struct holding information about the event. The callback function also receives a pointer to arbitrary user data, which has been defined by the argument “userp” when calling the tag\_hub\_callback() function.

The callback use for a particular type of events is disabled by setting the callback function to NULL instead of a function pointer and specifying the type of the event.

### 3.3.8 tag\_hub\_add\_fd()

```
int tag_hub_add_fd(TAGHUB *hub, int fd, int mode, void *userp)
```

The function adds the file descriptor “fd” to “hub”. Changes to the file descriptor are reported as events when tag\_hub\_pending() is called with “hub”.

If the file descriptor was already added, this function will return an error.

The argument “mode” is a bitmask specifying what file descriptor changes generate an event. Possible values for “mode” are readable, writable, or a combination of both. At least one value must be specified.

Table 3 TAGFD\_definitions

Constant Name	Description
TAGFD_WRITE	Wait for file descriptor to become writable
TAGFD_READ	Wait for file descriptor to become readable

For example using “TAGFD\_WRITE | TAGFD\_READ” as bitmask, will generate an event if the file descriptor becomes writable or readable.

### 3.3.9 tag\_hub\_del\_fd()

```
int tag_hub_del_fd(TAGHUB *hub, int fd)
```

The function removes the file descriptor “fd” from the list of file descriptors associated with “hub”. That is, any changes to the specified file descriptor will not be reported as events after this function has been called.

Calling this function with a file descriptor that isn’t already added will make it return an error.

## 3.4 TAG Handle Functions

This subsection describes Reader-related functions, which are used to create and close TAG handles.

The names of the Reader-related functions start with “tag\_reader\_”.

### 3.4.1 tag\_reader\_init()

```
int tag_reader_init(TAGHUB *hub, TAG **reader, void *userp,
    char *hostname, int port)
```

This function creates a TAG handle to a specific Reader.

The argument “reader” returns a handle which must be used in all further function calls when communicating with that Reader.

This function also does the initial TAGP handshake communication, which is sending a HELO message. See TAGP Protocol Specification [2] for further information about the initial TAGP handshake communication.

The argument “hub” is a pointer to an existing hub object to which a Reader is attached.

The argument “userp” is a private pointer that will be passed along with all events that originate from this Reader later on, as a help for applications to map events with Reader instances and so forth. The argument “userp” can be set to NULL if not being used.

The argument “hostname” is the hostname to connect to, or NULL to use the local host. It can also be a numerical string of a standard dotted IP-address, for instance “192.168.0.102”.

The argument “port” is the port number of the hostname to connect to. The default port 9999 is used if the argument is set to 0.

### 3.4.2 tag\_reader\_close()

```
void tag_reader_close(TAG *reader)
```

This function did not exist until taglib 1.3

The function removes the TAG handle “reader” from the TAGHUB it was previously attached to and disconnects the TAG handle from the Reader. No further functions can be used on the closed handle.

The Reader is also disconnected.

## 3.5 Client Functions

This subsection describes client-related functions, which, for example, control client names, locking, and client-to-client communication.

The taglib data structure “tagstring” is used to hold client names. The data structure “tagstring” is defined as follows:

```
struct tagstring {
    char *str; /* points to a zero terminated string */
    size_t len; /* useful for when the string contains binary
                content and strlen() is not good enough */
};
typedef struct tagstring tagstr;
```

The taglib data structure “tagstringlist” is used to hold a list of several clients. The data structure “tagstringlist” is defined as follows:

```
struct tagstringlist {
    int entries; /* number of entries in the entry array */
    tagstr entry[1]; /* 'entries' number of array members */
};
```

```
};  
typedef struct tagstringlist tagstrlist;
```

The names of the client-related functions start with “tag\_reader\_”.

### 3.5.1 tag\_reader\_setname()

```
int tag_reader_setname(TAG *reader, char *name)
```

The function sets the argument “name” as the name of “reader”, which is the name of the application as seen from other applications connected to the same tagd server.

### 3.5.2 tag\_reader\_getname()

```
int tag_reader_getname(TAG *reader, tagstr **str)
```

The argument “str” returns the name of “reader”.

**Note:** tag\_free() must be called with “str” when it is no longer used.

### 3.5.3 tag\_reader\_getclients()

```
int tag_reader_getclients(TAG *reader, tagstrlist **list)
```

The argument “list” returns a list of names of all clients currently connected to the same tagd server as “reader”. The list includes the name of “reader”, consequently there is always at least one name in the list.

Call tag\_free() with “list” when it is no longer used to free allocated memory.

### 3.5.4 tag\_reader\_lock()

```
int tag_reader_lock(TAG *reader)
```

The function requests exclusive access to the tagd server resources for “reader”. After a successful lock, “reader” will be the only client that can set variables in the global name space of the server. Other clients will be able to get information about the settings.

The function returns zero if the lock request was granted and a non-zero tagerr code if the request was denied.

### 3.5.5 tag\_reader\_unlock()

```
int tag_reader_unlock(TAG *reader)
```

The function releases a previous lock by “reader” and other clients are allowed access to the tagd server resources.

### 3.5.6 tag\_reader\_settalk()

```
int tag_reader_settalk(TAG *reader, bool enable)
```

The function enables or disables talk messages for “reader”. If the argument “enable” is set to true, the client accepts talk messages.

### 3.5.7 tag\_reader\_talk()

```
int tag_reader_talk(TAG *reader, char *client,
```

```
char *data, size_t len)
```

The function sends a talk message from “reader” containing “data” with the size “len” to the named “client” via the tagd server.

Communicating clients must have talk messages enabled and only clients connected to the same tagd server can exchange talk messages.

### 3.5.8 tag\_reader\_seteavesdrop()

```
int tag_reader_seteavesdrop(TAG *reader, bool allow)
```

The function controls whether the TAGP communication of “reader” can be eavesdropped by another client or not. If “allow” is true, “reader” allows eavesdropping.

**Note:** Allowing eavesdropping is only recommended while debugging. For security reasons, eavesdropping is disabled by default.

### 3.5.9 tag\_reader\_geteavesdrop()

```
int tag_reader_geteavesdrop(TAG *reader, bool *allow)
```

The argument “allow” returns the eavesdropping setting from “reader”.

## 3.6 Radio-related Functions

This subsection describes radio-related functions, which, for example, control frequency channel, frequency hopping, and reading range.

When using Frequency-Hopping Spread Spectrum (FHSS), sub-band definitions are specified by a bitmask that is created with constant values defined as bitwise operators.

TagMaster sub-bands “A”–“P” are 5 MHz wide and have frequencies separated by 200 kHz, that is 25 different channel frequencies in each sub-band.

Table 4 TAG\_FHSS\_BAND\_definitions

Constant Name	Description
TAG_FHSS_BAND_A	Sub-band “A”: 2402.0–2406.8 MHz
TAG_FHSS_BAND_B	Sub-band “B”: 2407.0–2411.8 MHz
TAG_FHSS_BAND_C	Sub-band “C”: 2412.0–2416.8 MHz
TAG_FHSS_BAND_D	Sub-band “D”: 2417.0–2421.8 MHz
TAG_FHSS_BAND_E	Sub-band “E”: 2422.0–2426.8 MHz
TAG_FHSS_BAND_F	Sub-band “F”: 2427.0–2431.8 MHz
TAG_FHSS_BAND_G	Sub-band “G”: 2432.0–2436.8 MHz
TAG_FHSS_BAND_H	Sub-band “H”: 2437.0–2441.8 MHz
TAG_FHSS_BAND_I	Sub-band “I”: 2442.0–2446.8 MHz
TAG_FHSS_BAND_J	Sub-band “J”: 2447.0–2451.8 MHz
TAG_FHSS_BAND_K	Sub-band “K”: 2452.0–2455.8 MHz

TAG_FHSS_BAND_L	Sub-band “L”: 2457.0–2461.8 MHz
TAG_FHSS_BAND_M	Sub-band “M”: 2462.0–2466.8 MHz
TAG_FHSS_BAND_N	Sub-band “N”: 2467.0–2471.8 MHz
TAG_FHSS_BAND_O	Sub-band “O”: 2472.0–2476.8 MHz
TAG_FHSS_BAND_P	Sub-band “P”: 2477.0–2481.8 MHz
TAG_FHSS_BAND_ALL	All available sub-band channels
TAG_FHSS_BAND_DEFAULT	Default FHSS sub-bands are band G, band H, band I, band J, band K, and band L.

Possible modes of the FHSS functionality are defined as constant values for the data type “tag\_fhssmode”.

*Table 5 Possible values of tag\_fhssmode enumeration*

Constant Name	Description
TAG_FHSS_OFF	Frequency-hopping is disabled
TAG_FHSS_ON	Frequency-hopping is enabled
TAG_FHSS_ADAPTIVE	Adaptive frequency hopping is used.

### 3.6.1 tag\_reader\_setchannel()

```
int tag_reader_setchannel(TAG *reader, int channel)
```

The function sets the output carrier channel of “reader” to “channel”. Valid channels are in the range 5–97. Channels are separated by 300 kHz, which is backwards compatible with TagMaster’s previous generation of Readers.

Note that if frequency hopping is enabled, a call to this function will return an error. Setting a channel while hopping is not possible.

### 3.6.2 tag\_reader\_getchannel()

```
int tag_reader_getchannel(TAG *reader, int *channel)
```

The argument “channel” returns the current output carrier channel from “reader”.

The function returns an error if the output carrier frequency is outside the valid channel band.

### 3.6.3 tag\_reader\_setfreq()

```
int tag_reader_setfreq(TAG *reader, int frequency)
```

The function sets the output carrier frequency of “reader” to “frequency”. Valid frequencies are 2.4361–2.4641 GHz expressed as integer values in the range 24361–24641.

For example, to set the frequency to 2.451 GHz, the argument “frequency” is expressed as 24510.

Note that if frequency hopping is enabled, a call to this function will return an error. Setting a frequency while hopping is not possible.

### 3.6.4 tag\_reader\_getfreq()

```
int tag_reader_getfreq(TAG *reader, int *frequency)
```

The argument “frequency” returns the current output carrier frequency of “reader”.

### 3.6.5 tag\_reader\_setfhss()

```
int tag_reader_setfhss(TAG *reader, tag_fhssmode mode,  
int bandmask)
```

The function sets the frequency hopping sub-bands and mode of “reader”. At least one sub-band must be defined if frequency hopping is enabled. The argument “bandmask” specifies which sub-bands to use according to TAG\_FHSS\_BAND\_ definitions. Each bit in the mask corresponds to a sub-band.

If the argument “mode” is off, the sub-band mask is ignored and frequency hopping is disabled.

Using the functions tag\_reader\_setfreq() or tag\_reader\_setchannel() to set the output carrier frequency while hopping is not possible.

Writing ID-tags is not possible while hopping. Stop frequency hopping before attempting to write an ID-tag.

After disabling frequency hopping, the output carrier frequency is set to the last frequency used while hopping. Specifying the carrier frequency using tag\_reader\_setfreq() or tag\_reader\_setchannel() is recommended directly after disabling frequency hopping.

### 3.6.6 tag\_reader\_getfhss()

```
int tag_reader_getfhss(TAG *reader, tag_fhssmode *mode,  
int *bandmask)
```

The arguments “bandmask” and “mode” respectively return the frequency hopping sub-bands and mode from “reader”. Use TAG\_FHSS\_BAND\_ definitions to decode individual sub-bands from the returned “bandmask”.

If “mode” is off, “bandmask” is always zero.

### 3.6.7 tag\_reader\_setreadrange()

```
int tag_reader_setreadrange(TAG *reader, int range)
```

The function sets the reading range of “reader” to “range”. Valid reading ranges are 1–4, where 4 is the maximum reading range.

Note that changing the read range parameter does not affect the output power of the Reader.

### 3.6.8 tag\_reader\_getreadrange()

```
int tag_reader_getreadrange(TAG *reader, int *range)
```

The argument “range” returns the current reading range of “reader”.

### 3.6.9 tag\_reader\_setreadlevel()

```
int tag_reader_setreadlevel(TAG *reader, int level)
```

The function sets the reading level of “reader” to “level”. Valid reading levels are 0–100, where 100 gives the maximum reading range.

Note that changing the read level parameter does not affect the output power of the Reader.

### 3.6.10 tag\_reader\_getreadlevel()

```
int tag_reader_getreadlevel(TAG *reader, int *level)
```

The argument “level” returns the current reading level of “reader”.

### 3.6.11 tag\_reader\_setcarrier()

```
int tag_reader_setcarrier(TAG *reader, bool carrier)
```

The function sets the output carrier of “reader” to on or off. The output carrier is on if the argument “carrier” is true.

Setting the carrier to off will drastically reduce the reading range of the Reader.

### 3.6.12 tag\_reader\_getcarrier()

```
int tag_reader_getcarrier(TAG *reader, bool *carrier)
```

The argument “carrier” returns the current output carrier setting of “reader”.

## 3.7 ID-tag Functions

This subsection describes functions concerning ID-tags, which, for example, control ID-tag filter, ID-tag speed, and writing data to ID-tags.

Possible modes of the ID-tag filter are defined as constant values for the data type “tag\_filermode”.

Table 6 Possible values of tag\_filermode enumeration

Constant Name	Description
TAGFILTER_OFF	ID-tag filter is off
TAGFILTER_ONCE	An ID-tag is reported once and before it is reported again, it must be out of the Reader’s reading range for a specified timeout period.
TAGFILTER_PERIODIC	An ID-tag is reported periodically with a specified timeout period.
TAGFILTER_REPORT	Works in the same manner as TAGFILTER_ONCE, but an ID-tag event is also sent when an ID-tag is leaving the reading lobe for more than a specified timeout period.

TAGFILTER_LAST	Just a marker, not an actual mode
----------------	-----------------------------------

### 3.7.1 tag\_reader\_settagspeed()

```
int tag_reader_settagspeed(TAG *reader, bool highspeed)
```

The function sets the ID-tag decoder of “reader” to decode high speed or low speed ID-tags. High speed ID-tags are decoded if the argument “highspeed” is true.

Note that the Reader is not able to read low speed ID-tags if the decoder is set to decode high speed ID-tags, and the opposite is also true.

### 3.7.2 tag\_reader\_gettagspeed()

```
int tag_reader_gettagspeed(TAG *reader, bool *highspeed)
```

The argument “highspeed” returns the current ID-tag decoder speed setting of “reader”.

### 3.7.3 tag\_reader\_settagfilter()

```
int tag_reader_settagfilter(TAG *reader, tag_filtermode mode,
    int timeout)
```

The function sets the tag filter mode and tag filter timeout of “reader”. The argument “mode” specifies the tag filter mode as described in Table 6. The argument “timeout” is expressed in milliseconds. If “mode” is off, the “timeout” is not used.

The use of tag filter is recommended for most applications. Not using tag filter will create numerous ID-tag events. Disabling tag filter is recommended only for test purposes.

### 3.7.4 tag\_reader\_gettagfilter()

```
int tag_reader_gettagfilter(TAG *reader, tag_filtermode *mode,
    int *timeout)
```

The arguments “mode” and “timeout” respectively return the tag filter mode and tag filter timeout of “reader”. If tag filter mode is off, “timeout” returns zero.

### 3.7.5 tag\_reader\_flushtagfilter()

```
int tag_reader_flushtagfilter(TAG *reader)
```

Flush the ID-tag filter, that is remove all ID-tags currently residing in the filter. This function was added in taglib 1.3.0.

### 3.7.6 tag\_reader\_writetag()

```
int tag_reader_writetag(TAG *reader, unsigned char *mark,
    char *options, unsigned char *userdata,
    size_t datalen, size_t timeout)
```

The function formats and writes user data to a writable ID-tag through “reader”. The function can be used in two different ways. The first way is to have the function block until a write attempt is successfully completed or until the timer expires, depending on whichever comes first. The second way is to start writing and immediately return. In the latter case, an event is received when the ID-tag writing is completed successfully.

The argument “mark” is a 9-byte array holding a binary 70-bit ID-tag mark that addresses a specific ID-tag. If “mark” is NULL, the first writable ID-tag that appears after a call to this function will be addressed.

The argument “options” is a series of control characters, which specify different ID-tag options as described in the tables below. For example, if “options” is “QR8H”, the writable ID-tag is formatted with quarter user data memory size, randomly distributed intervals, eight intervals, and high data speed.

An alternative to specify the ID-tag format, “options” can be “\*” (an asterisk). The current ID-tag format is then kept and only the user data is updated.

Table 7 Options for user data memory size

Control Character	Description
“M”	Mini user data memory size, 14 bits (46 bits if no CRC is used)
“Q”	Quarter user data memory size, 154 bits (186 bits if no CRC is used)
“F”	Full user data memory size, 574 bits (606 bits if no CRC is used)
“X”	Extended user data memory size, 1180 bits (1212 bits if no CRC is used), this user data memory size is a future feature

Table 8 Options for random or continuous intervals

Control Character	Description
“R”	Randomly distributed interval lengths
“C”	Constant interval lengths

Table 9 Options for number of intervals

Control Character	Description
“0”	Continuous operation, which means no idle period
“4”	Four intervals
“8”	Eight intervals
“6”	16 intervals

Table 10 Options for ID-tag data speed

Control Character	Description
“H”	High speed, 16 kbps
“L”	Low speed, 4 kbps

Table 11 Option for CRC generation

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:** If the automatic CRC generation is disabled and an ID-tag is successfully written, that ID-tag cannot be read by a Reader unless `tag_reader_setdiscard()` has been called with “`datacrc`” set to false. Not using the CRC makes the client responsible for performing necessary ID-tag data integrity checks.

The argument “`userdata`” is the byte array holding the data to write.

The argument “`datalen`” is the size of the data to write. If the size of the provided data is greater than the specified user data size (see Table 7), the data is truncated to fit the corresponding user data size. If the size of the provided data is less than the specified user data size, the data is padded with zero-bits.

The argument “`timeout`” specifies the allowed time in milliseconds for the write process to complete, which means that this function blocks until the writing is completed.

Set “`timeout`” to zero to omit the timeout, in which case the application receives a write complete event when the write attempt is completed. The write process can be cancelled with `tag_reader_writecancel()`.

A started write attempt must be completed or cancelled before a new write attempt can be started.

**Note:** Writing to ID-tags is not possible while frequency hopping is enabled or the write option is unavailable.

### 3.7.7 `tag_reader_writecancel()`

```
int tag_reader_writecancel(TAG *reader)
```

The function immediately cancels an ongoing ID-tag write attempt by “`reader`”.

The function returns an error if there is no ongoing write attempt.

### 3.7.8 `tag_reader_settagdiscard()`

```
int tag_reader_settagdiscard(TAG *reader, bool markcrc,
                             bool datacrc)
```

The function controls whether to discard ID-tags that have been read with bad CRC or not, by “`reader`”. ID-tags read with bad CRC are discarded and not reported to the application by default.

If the argument “`markcrc`” is true, ID-tags read with bad mark CRC are discarded.

If “`datacrc`” is true, writable ID-tags with bad user data CRC are discarded.

To have both CRC discards enabled is strongly recommended. Not using the CRC discards is only recommended for testing purposes or when the application is responsible for doing user data CRC checks.

### 3.7.9 `tag_reader_gettagdiscard()`

```
int tag_reader_gettagdiscard(TAG *reader, bool *markcrc,
```

```
bool *datacrc)
```

The arguments “markcrc” and “datacrc” respectively return the ID-tag mark CRC discard setting and ID-tag user data CRC discard setting from “reader”.

## 3.8 Reader Indicator Functions

This subsection describes functions concerning the visual indicator and the buzzer.

Possible colours of the visual indicator are defined as constant values for the data type “tag\_ledcolor”.

Table 12 Possible values of tag\_ledcolor enumeration

Constant Name	Description
TAGLED_OFF	Visual indicator is off
TAGLED_GREEN	Visual indicator is green
TAGLED_RED	Visual indicator is red
TAGLED_YELLOW	Visual indicator is a mix of green and red

### 3.8.1 tag\_reader\_setled()

```
int tag_reader_setled(TAG *reader, tag_ledcolor color)
```

The function sets the colour of the visual indicator of “reader” to “color”.

Note that a call to this function will override and stop a started tag\_reader\_blink().

### 3.8.2 tag\_reader\_getled()

```
int tag_reader_getled(TAG *reader, tag_ledcolor *color)
```

The argument “color” returns the current colour of the visual indicator from “reader”.

### 3.8.3 tag\_reader\_blink()

```
int tag_reader_blink(TAG *reader, tag_ledcolor first,
                    int duration, tag_ledcolor second)
```

The function flashes the visual indicators of “reader”. The indicators are set to colour “first” for the number of milliseconds specified by “duration”, and then set to colour “second”.

### 3.8.4 tag\_reader\_setbuzzer()

```
int tag_reader_setbuzzer(TAG *reader, bool on)
```

The function activates or deactivates the buzzer of “reader”. The buzzer will sound until it is deactivated if the argument “on” is true.

Note that a call to this function will override and stop a started tag\_reader\_beep().

### 3.8.5 tag\_reader\_getbuzzer()

```
int tag_reader_getbuzzer(TAG *reader, bool *on)
```

The argument “on” returns the current buzzer setting of “reader”.

Note that if `tag_reader_beep()` function is used the returned setting is somewhat misleading as it will not take the read beep buzzer use into account.

### 3.8.6 tag\_reader\_beep()

```
int tag_reader_beep(TAG *reader, int ms)
```

The function sounds the buzzer of “reader” for the number of milliseconds specified by “ms”.

### 3.8.7 tag\_reader\_setreadbeep()

```
int tag_reader_setreadbeep(TAG *reader, bool mode)
```

The function sets the read beep mode of “reader” to on or off. The read beep mode is on and the Reader will automatically make a short beep each time reading an ID-tag if the argument “mode” is true.

### 3.8.8 tag\_reader\_getreadbeep()

```
int tag_reader_getreadbeep(TAG *reader, bool *mode)
```

The argument “mode” returns the read beep mode of “reader”.

## 3.9 Reader I/O Functions

This subsection describes functions concerning inputs and outputs.

When using taglib, inputs and outputs are specified by a bitmask that is created with constant values defined as bitwise operators.

Table 13 TAG\_OUTPUT\_definitions

Constant Name	Description
TAG_OUTPUT_1	Output 1
TAG_OUTPUT_2	Output 2
TAG_OUTPUT_EXP1	Expansion output 1
TAG_OUTPUT_EXP2	Expansion output 2
TAG_OUTPUT_WICLK	Wiegand clock output
TAG_OUTPUT_WIDATA	Wiegand data output
TAG_OUTPUT_WILOAD	Wiegand load signal output

Table 14 TAG\_INPUT\_definitions

Constant Name	Description
TAG_INPUT_1	Input 1
TAG_INPUT_2	Input 2
TAG_INPUT_3	Input 3

TAG_INPUT_EXP1	Expansion input 1
TAG_INPUT_EXP2	Expansion input 2
TAG_INPUT_EXPIRQ	Expansion interrupt input

### 3.9.1 tag\_reader\_setoutput()

```
int tag_reader_setoutput(TAG *reader, int output, bool high)
```

The function sets the value for a single output of “reader” to high or low. The argument “output” specifies which output to set using TAG\_OUTPUT\_definitions. If the argument “high” is true, the specified output value is high.

The Wiegand/Mag-stripe interface outputs can be accessed through this function, which is useful only if the Wiegand/Mag-stripe interface is intended to operate as ordinary outputs and not with the Wiegand or Mag-stripe protocol.

Do not use this function to set a new value to the Wiegand/Mag-stripe interface outputs if using the functions tag\_reader\_sendwiegand() or tag\_reader\_sendmagstripe(), because the Wiegand/Mag-stripe functionality will be interfered and stopped until next Reader reset.

### 3.9.2 tag\_reader\_getoutput()

```
int tag_reader_getoutput(TAG *reader, int output, bool *high)
```

The argument “high” returns the current value of a single output of “reader”. The argument “output” specifies the requested output using TAG\_OUTPUT\_definitions. If “high” is true, the specified output value is high.

### 3.9.3 tag\_reader\_setoutputs()

```
int tag_reader_setoutputs(TAG *reader, int bitmask)
```

The function sets the values for all outputs of “reader” individually to high or low. The argument “bitmask” is a bitmask that defines how the outputs are set. If a bit is set in the bitmask, the corresponding output is set to high.

For example using “TAG\_OUTPUT\_1|TAG\_OUTOUT\_EXP2” as bitmask, will set output 1 and expansion output 2 to high and set all other outputs to low.

Note that it is not possible to set the Wiegand/Mag-stripe interface outputs using this function. However, the current values of these outputs can be retrieved using the function tag\_reader\_getoutputs().

For direct access to the Wiegand/Mag-stripe interface output use the functions tag\_reader\_sendwiegand() or tag\_reader\_sendmagstripe().

### 3.9.4 tag\_reader\_getoutputs()

```
int tag_reader_getoutputs(TAG *reader, int *bitmask)
```

The argument “bitmask” returns the values of all outputs of “reader” in a single bitmask.

Use TAG\_OUTPUT\_definitions to decode individual output values from the returned bitmask. If a bit is set, the corresponding output is high.

### 3.9.5 tag\_reader\_getinput()

```
int tag_reader_getinput(TAG *reader, int input, bool *high)
```

The argument “high” returns the value of a single input from “reader”. The argument “input” specifies the requested input using TAG\_INPUT\_definitions. If “high” is true, the specified input value is high.

### 3.9.6 tag\_reader\_getinputs()

```
int tag_reader_getinputs(TAG *reader, int *bitmask)
```

The argument “bitmask” returns the values of all inputs of “reader” in a single bitmask.

Use TAG\_INPUT\_definitions to decode individual input values from the returned bitmask. If a bit is set, the corresponding input is high.

### 3.9.7 tag\_reader\_setinputevents()

```
int tag_reader_setinputevents(TAG *reader, int bitmask)
```

The function sets the global input events bit mask of “reader”. The bitmask is used to control which inputs notify clients about changed values. The argument “bitmask” specifies which inputs will notify the client and is expressed using TAG\_INPUT\_bit definitions. If a bit is set in the bitmask, the corresponding input will report events.

By default all input events are disabled, which corresponds to bitmask 0x00.

TAG\_INPUT\_EXP1 and TAG\_INPUT\_EXP2 have no effect because they are not interrupt pins.

### 3.9.8 tag\_reader\_getinputevents()

```
int tag_reader_getinputevents(TAG *reader, int *bitmask)
```

The argument “bitmask” returns which inputs report events of “reader”, in a single bitmask.

Use TAG\_INPUT\_definitions to decode individual input values from the returned bitmask. If a bit is set, the corresponding input report events.

### 3.9.9 tag\_reader\_gettamper()

```
int tag_reader_gettamper(TAG *reader, bool *state)
```

The argument “state” returns the current state of the tampering switch from “reader”. If “state” is true, the tampering switch is closed, that is pressed down.

### 3.9.10 tag\_reader\_setrelay()

```
int tag_reader_setrelay(TAG *reader, bool open)
```

The function closes or opens the relay of “reader”. If “open” is true, the relay will open.

Note that a call to this function will override and stop a started tag\_reader\_openrelay().

### 3.9.11 tag\_reader\_getrelay()

```
int tag_reader_getrelay(TAG *reader, bool *open)
```

The argument “open” returns the current relay setting from “reader”.

### 3.9.12 tag\_reader\_openrelay()

```
int tag_reader_openrelay(TAG *reader, int ms)
```

The function opens the relay of “reader” for a number of milliseconds specified by “ms”.

### 3.9.13 tag\_reader\_setfanoutput()

```
int tag_reader_setfanoutput(TAG *reader, bool on)
```

The function controls the fan output of “reader”. If “on” is true, the fan output is activated.

### 3.9.14 tag\_reader\_getfanoutput()

```
int tag_reader_getfanoutput(TAG *reader, bool *on)
```

The argument “on” returns the current output setting from “reader”.

## 3.10 Reader System Functions

This subsection describes functions concerning Reader system information.

The timeval data structure is specified in the <sys/time.h> header.

The Reader system information is specified in a data structure, as follows:

```
struct tag_systeminfo {
    int tagd;          /* tagd version number =
                       major * 10000 + minor * 100 + micro */
    int tagmod;       /* tagmod version number */
    int taglib;       /* taglib version number */
    int rf_revision;  /* rf-unit revision */
    int rf_serno;     /* rf-unit serial number */
    int cb_type;      /* controller board product type number */
    int cb_revision;  /* controller board revision */
    int cb_serno;     /* controller board serial number */
};
```

### 3.10.1 tag\_reader\_getsysteminfo()

```
int tag_reader_getsysteminfo(TAG *reader, struct tag_systeminfo
*info)
```

The argument “tag\_systeminfo” returns the system information from “reader” as specified by the tag\_systeminfo struct.

### 3.10.2 tag\_reader\_setsystemtime()

```
int tag_reader_setsystemtime(TAG *reader, struct timeval *tv)
```

The function sets the system time for “reader”. The argument “tv” specifies the current time.

This function will also set the new time to the battery backed up RTC, if available.

### 3.10.3 tag\_reader\_getsystemtime()

```
int tag_reader_getsystemtime(TAG *reader, struct timeval *tv)
```

The argument “tv” returns the system time from “reader”.

### 3.10.4 tag\_reader\_gettemperature()

```
int tag_reader_gettemperature(TAG *reader, int *temperature)
```

The argument “temperature” returns the chip temperature from “reader” as degrees Celsius.

The temperature is an estimate and is most likely not the same as the temperature of the surrounding environment due to heat from the electronic components.

## 3.11 Communication Interfaces Functions

This subsection describes functions concerning the RS485 communication interface and the Wiegand/Mag-stripe communication interface.

### 3.11.1 tag\_reader\_setrs485fullduplex()

```
int tag_reader_setrs485fullduplex(TAG *reader, bool fullduplex)
```

The function sets the RS485 serial communication interface of “reader” to operate in full-duplex mode (4-wire) or half-duplex mode (2-wire). The RS485 interface operates in full-duplex mode if the argument “fullduplex” is true.

### 3.11.2 tag\_reader\_getrs485fullduplex()

```
int tag_reader_getrs485fullduplex(TAG *reader, bool *fullduplex)
```

The argument “fullduplex” returns the RS485 serial communication interface setting of “reader”.

### 3.11.3 tag\_reader\_sendwiegand()

```
int tag_reader_sendwiegand(TAG *reader, char *data,  
    size_t datalen)
```

The function sends Wiegand data to the Wiegand/Mag-stripe interface of “reader”.

The argument “data” is the raw Wiegand data to send. Parity calculation and zero padding must be handled by the application. The most significant bit in the first data character is sent first.

The argument “datalen” is the number of bits to send.

**Note:** If using the Wiegand/Mag-stripe interface, do not access the Wiegand or Mag-stripe output pins individually using tag\_reader\_setoutput(), because it will disable the Wiegand functionality.

### 3.11.4 tag\_reader\_sendmagstripe()

```
int tag_reader_sendmagstripe(TAG *reader, unsigned char *data,  
    size_t datalen)
```

The function sends Mag-stripe data to the Wiegand/Mag-stripe interface of “reader”.

The argument “data” is the raw Mag-stripe data to send. Parity calculation and zero padding must be handled by the application. The most significant bit in the first data character is sent first

The argument “datalen” is the number of bits to send.

**Note:** If using the Wiegand/Mag-stripe interface, do not access the Wiegand or Mag-stripe output pins individually using `tag_reader_setoutput()`, because it will disable the Wiegand functionality.

## 3.12 Special Functions

This subsection describes functions concerning special features, such as Accurate Positioning (APOS).

Two constants regarding APOS are defined in taglib.

Table 15 TAG\_APOS\_definitions

Constant Name	Description
TAG_APOS_THRESHOLD_DEFAULT	Default APOS threshold value
TAG_APOS_TIMEOUT_DEFAULT	Default APOS timeout value

### 3.12.1 tag\_reader\_sendrawtagp()

```
int tag_reader_sendrawtagp(TAG *reader, char *message,
    char **reply)
```

The function sends a raw TAGP message and waits for reply through “reader”. The arguments “message” and “reply” are respectively the message to send and the received reply.

TAGP messages are new-line and NULL terminated strings. The message frame must be escaped according to the TAGP Protocol specification [1].

**Note:** “reply” must be freed by the application after a call to this function.

### 3.12.2 tag\_reader\_setaccurateposition()

```
int tag_reader_setaccurateposition(TAG *hnd, bool mode,
    int timeout, int threshold, bool output)
```

The function controls the Accurate Positioning (APOS) settings for “reader”.

The argument “mode” is used to enable or disable APOS detection. If “mode” is true, APOS is enabled and APOS events will be received each time an APOS has been established.

The argument “threshold” specifies the signal threshold. Valid thresholds are in the range 1–60000.

The argument “timeout” specifies the exact time in milliseconds between APOS detection and APOS reporting. Valid timeouts are in the range 1 – 655.

If “output” is true, each APOS is reported by a pulse on output 1, with a delay specified by “timeout”.

Note that APOS is an optional feature and that the algorithm settings are strongly dependent on the Reader installation.

### 3.12.3 tag\_reader\_getaccurateposition()

```
int tag_reader_getaccurateposition(TAG *hnd, bool *mode,  
    int *timeout, int *threshold, bool *output)
```

The function retrieves the current APOS settings from “reader”.

The arguments “mode”, “timeout”, “threshold”, and “output” respectively return mode, timeout, signal threshold, and output as described in section 3.12.2.

### 3.12.4 tag\_reader\_config()

```
int tag_reader_config(TAG *hnd, char *conf)
```

This function sends a new configuration string to the reader and reconfigures it accordingly. The string needs to be a correctly generated string created by TagMaster, and a string is only valid for a particular reader with a specified serial number.

After reconfiguration, the target shall be rebooted for the changes to take effect.

This function was added in taglib 1.3.0

## 4 Examples

The following examples demonstrate taglib with source code written in C. The examples cover the TAGHUB concept, event handling, serial communication, and handling multiple Readers.

Note that the code has been written for demonstration purposes and for simplicity. No error handling has been implemented.

The following setup is assumed:

- Client 1: Local Reader application running on the same host as the tagd server
- User have terminal access to client 1 through the service interface
- Client 2: Remote Reader with IP address 193.15.235.111

### 4.1 First Application

This application creates a TAGHUB and connects to the local Reader. A connection to the local Reader is established by setting hostname to NULL and port to 0.

The function `tag_hub_pending()` is used to wait for events. A switch statement is used to decide whether an event, an error or a timeout has occurred. A second switch statement is used to decide if an ID-tag was read.

```
/*
 * Example 01: First application
 *
 * Jonas Romfelt, TagMaster AB, 2006, 2007
 *
 * Please note that this is an example for demonstration
 * purposes only, this code should be rewritten before used in a
 * critical application. Anyone is free to copy, modify,
 * publish, use, compile, sell, or distribute this example code,
 * either in source code form or as a compiled binary.
 */

#include <stdio.h>

#include "taglib.h"

int main(void)
{
    TAGHUB *hub;
    TAG *reader;
    struct event evnt;
    int rc;

    /* create TAGHUB and TAG reader handle with hostname=NULL and
       port=0 to use default: localhost/127.0.0.1 and 9999 */
    tag_hub_init(&hub);
    tag_reader_init(hub, &reader, NULL, NULL, 0);

    /* set some reader properties */
```

```
tag_reader_setchannel(reader, 77);
tag_reader_settagfilter(reader, TAGFILTER_PERIODIC, 1000);
tag_reader_setreadbeep(reader, false);

while (1) {
    /* start pending and wait for events, set timeout to 5 s */
    rc = tag_hub_pending(hub, 5000);

    if (rc < 0) {
        printf("Pending returned error, exit\n");
        return rc;
    }
    else if (rc == 0) {
        printf("Pending timeout\n");
    }
    else {
        /* handle received event */
        tag_hub_poll(hub, &evnt);

        switch(evnt.type) {
            case TYPE_TAG:
                printf("Tag event: %08d\n",
                    tag_marktoint(evnt.data.tag->mark));
                break;
            default:
                printf("Unknown event\n");
                break;
        }
    }
}

return 0;
}
```

To test this application, present an ID-tag to the Reader and follow the output on the terminal connected to the service interface. Press the tamper switch to generate an unknown event and trigger the default callback function.

## 4.2 Using Callbacks

This application creates a TAGHUB and connects to the local Reader.

A callback is registered to take care of ID-tag events. Note that the callback is only called while in `tag_hub_pending()`. If timeout is set to zero, `tag_hub_pending()` will only return when an event is received that is not associated with a callback.

The callback function will beep the buzzer for 300 ms every time an ID-tag is read.

If `tag_hub_pending()` returns when an event has been received and is called again prior to a call to `tag_hub_poll()`, the event that caused `tag_hub_poll()` to return, will be removed from the event queue.

```
/*
 * Example 02: Application using call-back for tag events
 */
```

```
* Jonas Romfelt, TagMaster AB, 2006, 2007
*
* Please note that this is an example for demonstration
* purposes only, this code should be rewritten before used in a
* critical application. Anyone is free to copy, modify,
* publish, use, compile, sell, or distribute this example code,
* either in source code form or as a compiled binary.
*
*/

#include <stdio.h>

#include "taglib.h"

static void tag_callback(struct event *evnt, void *userp)
{
    (void)userp;
    tag_reader_beep(evnt->origin.reader, 300);
    if (tag_ismarkopen32(evnt->data.tag->mark))
        printf("Tag call-back: 0x%08X\n",
               tag_marktoopen32(evnt->data.tag->mark));
    else
        printf("Tag call-back: %08d\n",
               tag_marktoint(evnt->data.tag->mark));
}

int main(void)
{
    TAGHUB *hub;
    TAG *reader;

    /* create TAGHUB and a TAG reader handle */
    tag_hub_init(&hub);
    tag_reader_init(hub, &reader, NULL, NULL, 0);

    /* set some reader properties */
    tag_reader_setchannel(reader, 77);
    tag_reader_settagfilter(reader, TAGFILTER_PERIODIC, 1000);
    tag_reader_setreadbeep(reader, false);

    /* register callback, i.e. when a tag event is received the
       function tag_callback() is automatically called from
       tag_hub_pending() */
    tag_hub_callback(hub, TYPE_TAG, tag_callback, NULL);

    /* blink with visual indicators */
    tag_reader_blink(reader, TAGLED_GREEN, 500, TAGLED_OFF);

    while (1) {
        tag_hub_pending(hub, 0);
        printf("Unknown event\n");
        /* please not that it is not necessary to use tag_hub_poll()
           To remove this event from queue, it is automatically
           removed when tag_hub_pending() is called again */
    }
}
```

```
    return 0;
}
```

To test this application, present an ID-tag to the Reader and follow the output on the terminal connected to the service interface. Press the tamper switch to generate an unknown event and trigger the default callback function.

### 4.3 Adding an RS232 File Descriptor

This application utilises the RS232 serial communication interface and is fully callback-oriented.

The application opens `/dev/ttyS1` and configures it as 9600 baud, eight data bits, one stop bit, and no parity. The application creates a TAGHUB and connects to the local Reader.

The RS232 file descriptor is added to the TAGHUB. Each time the file descriptor becomes readable, TAGHUB will notify the application.

Two callback functions are registered. The first callback is called when file descriptor status changes, that is when there is incoming data on the RS232 interface. The second callback is a default callback, which is called on all events not associated with the first callback.

A infinite `tag_hub_pending()` loop is created by setting the timeout to zero. All events have a callback, consequently pending will never return due to a thrown event. The function `tag_hub_pending()` will only exit if an error occurs.

```
/*
 * Example 03: 100% call-back oriented application that opens,
 * configures and adds the RS232 interface's file descriptor to
 * TAGHUB.
 *
 * Jonas Romfelt, TagMaster AB, 2006, 2007
 *
 * Please note that this is an example for demonstration
 * purposes only, this code should be rewritten before used in a
 * critical application. Anyone is free to copy, modify,
 * publish, use, compile, sell, or distribute this example code,
 * either in source code form or as a compiled binary.
 */

#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <termios.h> /* serial interface control bits */

#include "taglib.h"

static void fd_callback(struct event *evnt, void *userp)
{
    char buffer[200];
    int len;
```

```
(void)userp;

/* read data from file descriptor */
len = read(evnt->data.fd->fd, buffer, sizeof(buffer)-1);
buffer[len] = '\\0'; /* zero terminate string */

printf("fd(%d) call-back: %s\\n", evnt->data.fd->fd, buffer);
}

static void default_callback(struct event *evnt, void *userp)
{
    (void)userp;
    (void)evnt;
    printf("Default call-back\\n");
}

int main(void)
{
    TAGHUB *hub;
    TAG *reader;
    struct termios port;
    int fd;

    /* open RS232 serial interface */
    fd = open("/dev/ttyS1", O_RDWR | O_NOCTTY | O_NONBLOCK);
    printf("RS232 is accessed via fd=%d\\n", fd);

    /* setup interface: 9600, 8, 1, n */
    tcgetattr(fd, &port); /* get current settings */
    port.c_cflag = B9600 | CS8 | CLOCAL | CREAD;
    port.c_iflag = 0; /* raw input, no input processing */
    port.c_oflag = 0; /* raw output, no output processing */
    port.c_lflag = 0; /* disable canonical input */
    port.c_cc[VMIN] = 1; /* block until 1 character, i.e. don't
                        block */
    port.c_cc[VTIME] = 0; /* don't use inter-character timer */
    port.c_line = 0; /* line discipline: TTY */
    tcflush(fd, TCIFLUSH);
    tcsetattr(fd, TCSANOW, &port); /* activate new settings */

    /* create TAGHUB and TAG reader handle */
    tag_hub_init(&hub);
    tag_reader_init(hub, &reader, NULL, NULL, 0);

    /* register file descriptor at TAGHUB, so that hub tells
       application when serial interface is readable or/and
       writable */
    tag_hub_add_fd(hub, fd, TAGFD_READ, NULL);

    /* register file descriptor call-back function */
    tag_hub_callback(hub, TYPE_FD, fd_callback, NULL);

    /* register default call-back, takes care of all events not
       explicitly associated with a call-back function */
    tag_hub_callback(hub, TYPE_DEFAULT, default_callback, NULL);
}
```

```

    /* enter a forever pending, set timeout=0. for a pending to
       never exit, all events must be associated with a call-back
       */
    tag_hub_pending(hub, 0);

    return 0;
}

```

To test this application, connect a terminal emulator to the RS232 interface and input some text. Note the output on the service interface. Present an ID-tag to the Reader to call the default callback function.

## 4.4 Talk Messages

This application is fully callback-oriented and listens to talk messages. The application creates a TAGHUB and connects to the local Reader.

Two callback functions are registered. The first callback is registered as a talk callback. Received talk messages are printed on stdout. The second callback is a default callback, which is called on all events not associated with the first callback.

Talk is enabled and a client name is set. Talk is by default disabled and clients connected to the tagd server must explicitly state that they want to talk. All talking clients must also register a unique name. Note that sending talk messages to several clients at the same time is possible.

A infinite tag\_hub\_pending() loop is created by setting the timeout to zero. All events have a callback, consequently pending will never return due to a thrown event. The function tag\_hub\_pending() will only exit if an error occurs.

```

/*
 * Example 04: Talk messages, this application will enable talk
 * and print received messages to stdout
 *
 * Jonas Romfelt, TagMaster AB, 2006, 2007
 *
 * Please note that this is an example for demonstration
 * purposes only, this code should be rewritten before used in a
 * critical application. Anyone is free to copy, modify,
 * publish, use, compile, sell, or distribute this example code,
 * either in source code form or as a compiled binary.
 *
 */

#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <termios.h> /* serial interface control bits */

#include "taglib.h"

static void talk_callback(struct event *evnt, void *userp)
{

```

```
(void)userp;
printf("Talk call-back: '%s' sent '%s'\n",
      evt->data.talk->from,
      evt->data.talk->data);
}

static void default_callback(struct event *evnt, void *userp)
{
    (void)userp;
    (void)evnt;
    printf("Default call-back\n");
}

int main(void)
{
    TAGHUB *hub;
    TAG *reader;
    tagstrlist *list;
    int i;

    /* create TAGHUB and TAG reader handle */
    tag_hub_init(&hub);
    tag_reader_init(hub, &reader, NULL, NULL, 0);

    /* register talk call-back function */
    tag_hub_callback(hub, TYPE_TALK, talk_callback, NULL);

    /* register default call-back, takes care of all events not
       explicitly associated with a call-back function */
    tag_hub_callback(hub, TYPE_DEFAULT, default_callback, NULL);

    /* enable talk messaging and set application name */
    tag_reader_settalk(reader, true);
    tag_reader_setname(reader, "foo");

    /* get a list of connected clients */
    tag_reader_getclients(reader, &list);
    printf("Connected clients (%d):\n", list->entries);
    for (i = 0; i < list->entries; i++)
        printf("%s\n", list->entry[i].str);

    /* must free the returned list */
    tag_free(&list);

    /* enter a forever pending, set timeout=0 */
    tag_hub_pending(hub, 0);

    return 0;
}
```

To test this application, create a second application or manually interact with tagd using TAGP, as described in TAGP Protocol Specification [2].

## 4.5 Multiple Readers

This application connects to multiple Readers. The application creates a TAGHUB and connects to the local Reader. A TAG Reader handle to the remote client is created and connected to the same TAGHUB as the local Reader.

The application utilises user pointers holding a name for each Reader, which is passed to the callback.

Rather than using a global variable, a callback user pointer is used for data that has to be passed to the callback function from main().

This example can easily be increased to handle even more Readers.

```
/*
 * Example 05: Multi-reader application that use user defined
 * pointers when registering tag handles and call-backs.
 *
 * Jonas Romfelt, TagMaster AB, 2006, 2007
 *
 * Please note that this is an example for demonstration
 * purposes only, this code should be rewritten before used in a
 * critical application. Anyone is free to copy, modify,
 * publish, use, compile, sell, or distribute this example code,
 * either in source code form or as a compiled binary.
 */

#include <stdio.h>

#include "taglib.h"

static void default_callback(struct event *evnt, void *userp)
{
    int *counter = userp;
    printf("Event #%d from %s\n",
          ++*counter,
          (char *)evnt->userp);
}

int main(void)
{
    TAGHUB *hub;
    TAG *reader_local, *reader_remote;
    int evntcounter=0;

    /* create TAGHUB and TAG reader handle to local reader, set
       IP address NULL */
    tag_hub_init(&hub);
    tag_reader_init(hub, &reader_local, "local", NULL, 0);

    /* create a TAG reader handle to remote reader, connect it to
       same TAGHUB. Note that it is necessary to change the IP
       address in order for this example to work! */
    if (tag_reader_init(hub, &reader_remote, "remote",
```

```
        "193.15.235.111", 0)) {
    printf("Unable to connect to remote reader\n");
    return -1;
}

/* register default call-back, called when events are received
   from both readers, with pointer to an integer */
tag_hub_callback(hub, TYPE_DEFAULT, default_callback,
                 &evntcounter);

/* enter a forever pending, set timeout=0 */
tag_hub_pending(hub, 0);

return 0;
}
```

To test this application, a second Reader is required. Present an ID-tag to each Reader and note the output on the service interface of the Reader running this application. Change the IP address "192.15.235.111" to the actual IP address of the second Reader.

## 4.6 Using Inputs and Outputs

This application creates a TAGHUB and connects to the local Reader and is fully callback oriented.

Callbacks are registered to take care of tampering switch events and input change events.

The tampering switch callback shows how to use the event time to a readable calendar time format.

Input events mask is setup so that changes on input 1, input 2, and input 3 will result in events. The input source is decoded in the input events callback.

When the application starts, the relay is opened and output 2 is set high. The current output values are retrieved and the expansion outputs 1 and 2 are set high.

```
/*
 * Example 06: Using I/O, this example shows how to set up input
 * event and tampering switch call-backs. It also shows how to
 * set the relay, and set individual outputs as well as
 * addressing all outputs using a mask.
 * In the tampering switch call-back it is also demonstrated how
 * the event time-stamp can be converted into a readable
 * calendar time format.
 *
 * Jonas Romfelt, TagMaster AB, 2006, 2007
 *
 * Please note that this is an example for demonstration
 * purposes only, this code should be rewritten before used in a
 * critical application. Anyone is free to copy, modify,
 * publish, use, compile, sell, or distribute this example code,
 * either in source code form or as a compiled binary.
 */
```

```
 */
#include <stdio.h>
#include <time.h>

#include "taglib.h"

static void tamper_callback(struct event *evnt, void *userp)
{
    struct tm *dt;
    time_t t = evnt->data.tamper->secs;
    (void)userp;

    /* convert the event time (which is returned as seconds since
       1970 etc.) to calendar time, see time.h for more
       information. Note that the returned event time is the
       system time at the reader where event occurred */
    dt = localtime(&t);

    /* do some nice formatted output */
    printf("Tamper call-back at "
           "%04d-%02d-%02d %02d:%02d:%02d.%03d, "
           "switch changed to %s\n",
           dt->tm_year+1900,
           dt->tm_mon+1,
           dt->tm_mday,
           dt->tm_hour,
           dt->tm_min,
           dt->tm_sec,
           evnt->data.tamper->usecs / 1000, /* convert µs to ms */
           evnt->data.tamper->value?"closed":"open");
}

static void input_callback(struct event *evnt, void *userp)
{
    int in;
    (void)userp;

    switch (evnt->data.input->source) {
    case TAG_INPUT_1:
        in = 1;
        break;
    case TAG_INPUT_2:
        in = 2;
        break;
    case TAG_INPUT_3:
        in = 3;
        break;
    }

    printf("Input call-back, input %d changed to %d\n",
           in, evnt->data.input->value);
}

static void default_callback(struct event *evnt, void *userp)
```

```
{
    (void)userp;
    (void)evnt;
    printf("Default call-back\n");
}

int main(void)
{
    TAGHUB *hub;
    TAG *reader;
    int m;

    /* create TAGHUB and TAG reader handle */
    tag_hub_init(&hub);
    tag_reader_init(hub, &reader, NULL, NULL, 0);

    /* set some reader properties */
    tag_reader_setchannel(reader, 77);
    tag_reader_settagfilter(reader, TAGFILTER_PERIODIC, 1000);
    tag_reader_setreadbeep(reader, false);

    /* register call-back functions for tampering switch, input
       change, and a default call-back that takes care of all
       other events */
    tag_hub_callback(hub, TYPE_TAMPER, tamper_callback, NULL);
    tag_hub_callback(hub, TYPE_INPUT, input_callback, NULL);
    tag_hub_callback(hub, TYPE_DEFAULT, default_callback, NULL);

    /* enable events from inputs 1 to 3 */
    tag_reader_setinputevents(reader,
        TAG_INPUT_1 | TAG_INPUT_2 | TAG_INPUT_3);

    /* set relay to open */
    tag_reader_setrelay(reader, true);

    /* set output 2 high */
    tag_reader_setoutput(reader, TAG_OUTPUT_2, true);

    /* set expansion outputs 1 and 2 high, first get current
       output values and set the corresponding bits to 1 */
    tag_reader_getoutputs(reader, &m);
    m |= (TAG_OUTPUT_EXP1 | TAG_OUTPUT_EXP2);
    tag_reader_setoutputs(reader, m);

    /* enter a forever pending */
    tag_hub_pending(hub, 0);

    return 0;
}
```

To test this application, connect inputs to push-buttons and outputs to visual indicators (for instance LEDs). Pushing the push-buttons creates output on the service interface that reports the input that changed. Opening and closing the tampering switch will

output a time stamped event report on the service interface. All other event types, for instance ID-tag events, will be handled by the default callback.

## 5 Contact

For any further inquiries, please contact TagMaster AB.

### 5.1 Technical Support

Phone: + 46 8 632 19 50

Fax: +46 8 750 53 62

E-mail: [support@tagmaster.com](mailto:support@tagmaster.com)

### 5.2 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](mailto:sales@tagmaster.com)

Web: [www.tagmaster.com](http://www.tagmaster.com)

## 6 Glossary

<b>API</b>	Application Programming Interface A source code interface that a computer system or program library provides in order to support requests for services to be made of it by a computer program.
<b>APOS</b>	Accurate Positioning
<b>CPU</b>	Central Processing Unit
<b>CRC</b>	Cyclic Redundancy Check A type of hash function used to produce a checksum, in order to detect errors in transmission or storage
<b>MCU</b>	Microcontroller Unit A single chip that contains a processor, RAM, ROM, clock and I/O control unit.
<b>MIPS</b>	Million Instructions Per Second A measure of microprocessor speed
<b>Open32</b>	A special TagMaster ID-tag format
<b>RTC</b>	Real-Time Clock The clock that keeps the current time while a computer is turned off.

## 7 References

- [1] *GEN4 Reader User's Manual*  
Doc no. 06-118
- [2] *TAGP Protocol Specification*  
Doc no. 05-172