

*Mercury[®] OS API
Setup & Reference Guide*

**For: Mercury4, Mercury5 (v2.4.23 and later)
Astra Readers (v4.0.17 and later)**

Government Limited Rights Notice: All documentation and manuals were developed at private expense and no part of it was developed using Government funds.

The U.S. Government's rights to use, modify, reproduce, release, perform, display, or disclose the technical data contained herein are restricted by paragraph (b)(3) of the Rights in Technical Data — Noncommercial Items clause (DFARS 252.227-7013(b)(3)), as amended from time-to-time. Any reproduction of technical data or portions thereof marked with this legend must also reproduce the markings. Any person, other than the U.S. Government, who has been provided access to such data must promptly notify ThingMagic, Inc.

ThingMagic, Mercury, and the ThingMagic logo are trademarks or registered trademarks of ThingMagic, Inc.

Other product names mentioned herein may be trademarks or registered trademarks of ThingMagic, Inc. or other companies.

© Copyright 2000–2008 ThingMagic, Inc. All Rights Reserved

ThingMagic, Inc.
One Broadway, 5th floor
Cambridge, MA 02142
866-833-4069

Revision B
December, 2008

Revision Table

Date	Version	Description
11/9/07	2.0	Editing
10/07	2.0	int tagid_write Added info on tag singulation.
10/07	2.0	int tagdata_write. Gen2 capability.
10/07	2.0	int tag_password. Added Gen2 capability
10/07	2.0	int tagdata_read. Added Gen2 capability
August 2008	3	Added info on thread-safe API changes updated APIs that have been deprecated Added GPIO info for Astra
September 2008	4	Added SNMP section Added Platform specific section

Contents

Mercury[®] OS API Setup	9
API Setup: Writing Code	10
Thread Safe API	10
Building Applications for the Windows Platform	11
Requirements	11
Instructions	11
Building Applications for a Remote Linux PC/Server	11
Requirements	12
Instructions	12
Building Applications to Store and Run on the Reader	12
Requirements	12
Instructions	13
Installing the Cross Compiler and Binary Utilities	13
Creating and Compiling Programs	17
Testing the Program	19
Saving the Program to the JFFS2 File System	20
Ensuring Program Starts on Every System Boot (Optional)	20
API Reference	22
Connecting and Disconnecting	23
Connecting/Disconnecting Data Structure	23
Functions	24
Open a Connection to the Reader	24
Close a Connection to the Reader	25
Multiple Protocol/Multiple Antenna Reads	26
Searching for Tags and Resetting the Data Base	26
Tag Data Base Data Structure	27
Functions	33
Reset the Tag Data Base	33
Initiate a Simple Search Operation	34
Get the Status of a Search	36
Return the current record from the data base [Deprecated]	37

Return the current record and remove it from the data base	38
Initiate an autosearch operation [Deprecated]	39
Get the status of an autosearch operation [Deprecated]	40
Stop an AutoSearch Operation [Deprecated]	41
Initiate a detailed search	41
Stop a detailed search	48
Initiate a conveyor search operation	48
Stop a conveyor search operation	49
Fetch the Available Tags in the Data Base	50
Individual Tag Commands	51
Tag Operation Data Structure	51
Functions	52
Write a new tag id	52
Lock a tag's id	53
Kill a tag	54
Set a tag password	55
Read tag data	56
Write data to a tag	57
Lock tag data	58
GPIO Commands	60
GPIO Data Structure	60
Mercury4/Mercury5 GPIO	60
Astra GPIO	61
Functions	61
Get the State of GPIO Lines	61
Set the State of the GPIO Output Lines	62
Establish an Association with GPIO Lines and a Reader Event	63
Reader Configuration	65
Reader Configuration Data Structure	66
Functions	67
Get Protocol List Supported by Reader	67
Retrieve Antenna List Connected to Reader	68
Get Current Firmware Version	68
Get Current Operating System Version	69
Get Maximum TX Power	69
Set Transmitter Power	70
Set Transmitter Power per Slot [Deprecated]	71
Set the TX Power on a Specified Antenna	71
Get the Current Slot TX Power [Deprecated]	72
Get Current Antenna TX Power	73

Configure Reader to Enable or Disable Reading EPC1 Tags	73
Check Reader Configuration for Reading EPC1 Tags	74
Configure the Reader to Read only 64-Bit EPC1 Tags	75
Return the length of EPC1 tag IDs the reader is configured to read	75
Set the Number of EPC0 Tags the Reader Decodes	76
Get the Number of EPC0 Tags the Reader Decodes	76
Get the Status of EPC0 De-ghosting	77
Set the Status of EPC0 De-Ghosting	77
Get the EPC0 De-Ghosting Length	78
Set the Value of EPC0 De-Ghosting Length	78
Get the Status of EPC1 De-Ghosting	79
Set the Status of EPC1 De-Ghosting	79
Get Time Out for Tag Commands	80
Set the Time Out for Tag Commands	80
Set Reader Parameters	81
Get Reader Parameter Values [Deprecated]	81
Get Reader Parameter Values - Thread-safe	82
Get the List of Valid Parameters [Deprecated]	82
Get the List of Valid Parameters - Thread-safe	83
Perform a Firmware Update	84
Get Firmware Update Status	84
Reboot the Reader after Firmware Update Remotely	85
Status and Error Codes	86
Global Error Code	86
Functions	87
Print a Description of the Last Error	87
SNMP	89
Statistics Data Structures	89
Antenna Statistics	89
Reader Statistics	90
Functions	90
Get Antenna Statistics	90
Get Reader Statistics	91
Reset Statistics	91
.	91
Platform Specific Functionality	93
Astra Readers	94
Shared Memory Usage	94
Supported Protocols	94

Mercury4 & Mercury5 Readers	95
SNMP.....	95

Mercury[®]OS API Setup

The Mercury[®]OS API Setup and Reference Guide provides information about how to setup and pass API (Application Programming Interface) commands through to the reader.

Using the API to pass commands to the reader provides access to the reader at a low level. Customizing almost any parameter or setting is possible using a variety of API commands.

This reference guide assumes that the reader has experience with basic RFID theory, C programming, and building RPC-based applications.

The same APIs and finished applications can be used with the ThingMagic Mercury4, Mercury5 and Astra readers. Cases where an API or other information differs among readers are noted. It is strongly recommended that the [Platform Specific Functionality](#) section be read before beginning any development using these APIs, whether intended for a single platform or cross-platform. That section describes functionality and limitations unique to each platform. In cases where the API Reference differs from the information in that section the [Platform Specific Functionality](#) section information should be used.

API Setup: Writing Code

There are three methods in which to run programs that communicate with the reader. You can choose the method that is best suited to your requirements and whether the program is stored and run directly on the reader. You can also communicate with the reader over a networked connection.

- ◆ [Building Applications for the Windows Platform](#)

The application communicates with the reader over a networked connection. Internally the API uses open-network remote procedure calls (ONC RPC rfc1831) to communicate with the reader. These calls appear to be local C function calls.

- ◆ [Building Applications for a Remote Linux PC/Server](#)

The application communicates with the reader over a networked connection. The x86-linux API communicates with the reader by remote procedure calls using the Linux computer's built-in RPC interface.

- ◆ [Building Applications to Store and Run on the Reader](#)

The reader-based API uses the same programmer interface as the Linux and Windows APIs, except that the implementation of some of the calls is different. The reader-based APIs use shared memory (except on Astra Readers) instead of RPC calls to improve performance. To use the reader-based API, apply the cross-development toolchain provided in the developer kit from an x86-linux development machine and link against the arm-linux API libraries.

Although reader-based programs are more of a challenge to develop and debug, they offer many advantages over networked programs. The much greater speed possible with reader-based API calls makes high-speed conveyor-belt operations possible. In addition, it is possible to do sophisticated filtering on the reader, significantly reducing the network bandwidth necessary. To debug reader-based applications, gdb server can be used to run a debugger remotely.

Thread Safe API

The Mercury OS C API for v2.4.14 and later for the Mercury4 and Mercury5 and all Astra versions are thread safe. Except where noted all APIs are thread safe and do not require any changes from previous versions of the SDK except for linking with -pthread.

Building Applications for the Windows Platform

The following sections explain what is required to run the API application on a Windows® platform.

Requirements

The following items are needed to communicate with a reader over a networked connection.

- ◆ PC running Microsoft Windows 2000 or XP
- ◆ Microsoft Visual Studio, version 6.0 or later
- ◆ Experience writing applications using MS Visual Studio
- ◆ Mercury 4 or 5 Developer's Kit

Instructions

Use the following procedure to set up your program to run correctly on the Windows platform.

To run code using Windows:

1. Launch Microsoft Visual Studio.
2. In Visual Studio, create a new executable C/C++ project. Make sure the preprocessor definition WIN32 is defined for the project.
3. Copy the contents of the ReferenceCode\C\include (m4api.h) and ReferenceCode\C\lib (m4api.lib, m4api.dll, oncrpc.dll) folders to the new project directory.
4. Write the program and include m4api.h (#include "m4api.h") in your source code. Go to the [API Reference](#) for the API interfaces.
5. Compile the program and link against m4api.lib.
6. When running the program, make sure that m4api.dll and oncrpc.dll are in the current path. The program does not run if they are not present.

Building Applications for a Remote Linux PC/Server

The following sections explain what is required to run the API application on a computer with the Linux operation system.

Requirements

The following items are needed to communicate with a reader over a networked connection.

- ◆ PC running Linux
- ◆ GNU make, gcc
- ◆ Mercury Developer's Kit

Instructions

Use the following procedure to set up your program to run correctly on the Linux operating system.

To run programs on Linux systems:

1. Include `m4api.h` in your source code. Go to [API Reference](#) for the API interfaces.
2. Compile the program and link against `x86-libm4api.a` or `libx86m4api.so`.

```
gcc -o foo foo.c -lx86m4api
```

or

```
gcc -o foo foo.c x86-libm4api.a
```

Building Applications to Store and Run on the Reader

The following sections provide information required to set up and run APIs from the reader.

Requirements

The following items are needed to run APIs on the reader.

- ◆ A host computer such as a personal computer (PC) running a modern Linux distribution.
- ◆ A directory on the PC exported via the Network File System (NFS) that the reader can mount remotely. This allows you to share files between the two devices. In our example, we will assume the `/tmp` directory is shared.
- ◆ An M4 or M5 reader.
- ◆ An Ethernet/LAN connection between the PC and reader. Depending on your particular set up, you could require a crossover Ethernet cable.

Instructions

Use the following procedure to set up the reader to run API applications.

To run code on the reader:

1. Install the cross compiler and other binary utilities for the PC. See [Installing the Cross Compiler and Binary Utilities](#).
2. Develop and test your code on the PC.
3. Cross-compile the code on the PC for the target system i.e., the Mercury reader. See [Creating and Compiling Programs](#).
4. Test your code on the reader. See [Testing the Program](#).
5. Include the compiled binary of your code in the reader distribution.
6. Save your binary to the JFFS2 partition on the reader, ensuring it is saved to the flash, and available the next time the reader is started See [Saving the Program to the JFFS2 File System](#) and [Ensuring Program Starts on Every System Boot \(Optional\)](#).

Installing the Cross Compiler and Binary Utilities

Precompiled Linux binaries for the cross compiler are available on the ThingMagic Developer's CD. Copy the file *arm-linux-tools-20030927.tar.gz* to your local file system and extract the files using the following commands:

```
Your-Linux-PC-Prompt> cd /
Your-Linux-PC-Prompt> tar -xzvf /path/to/arm-linux-tools-20030927.tar.gz

root@tm-athena:~# ls
Desktop                               reader_API-1.00-dist.tar.gz
ThingMagicDevEnvAndTools              readtags.py
cpi                                    xsession.knx-hdinstall.backup
root@tm-athena:~# cd ThingMagicDevEnvAndTools/
root@tm-athena:~/ThingMagicDevEnvAndTools# ls
Makefile                               hello.c                                snapgear
arm-linux-tools-20030927.tar.gz         hello.c                                snapgear-3.0.0.tar.gz
hello                                   index.html                             tmlogo.jpg
root@tm-athena:~/ThingMagicDevEnvAndTools#

root@tm-athena:~/ThingMagicDevEnvAndTools# tar -C / -xzvf arm-linux-tools-
20030927.tar.gz
usr/local/include/g++-3/floatio.h
usr/local/include/g++-3/sstream
usr/local/include/g++-3/editbuf.h
usr/local/include/g++-3/builtinbuf.h
usr/local/include/g++-3/PlotFile.h
root@tm-athena:~/ThingMagicDevEnvAndTools#
```

Also, you need a copy of the kernel headers that correspond to the kernel used on the reader. The headers can be found in the snapgear source tree. Copy snapgear-3.2.0.tar.gz to your local file system and extract the files using the commands below and depicted in the screenshot below.

```
Your-Linux-PC-Prompt> tar -xzvf snapgear-3.2.0.tar.gz
```

Then enter the snapgear directory and type **make menuconfig**.

```
training1@tm-athena:~/snapgear_install$ ls -l
total 157212
-rw-r--r-- 1 root root 160820248 Oct 15 02:43 snapgear-3.0.0.tar.gz
training1@tm-athena:~/snapgear_install$ tar -xzf snapgear-3.0.0.tar.gz
training1@tm-athena:~/snapgear_install$ cd snapgear/
training1@tm-athena:~/snapgear_install/snapgear$ ls
COPYING      Makefile    bin         glibc      linux-2.4.x  uClibc
Documentation README     config      lib         linux-2.4.x  user
MAINTAINERS  SOURCE     freeswan    linux-2.0.x tools         vendors
training1@tm-athena:~/snapgear_install/snapgear$ make menuconfig
config/mkconfig > config.in
make -C /home/training1/snapgear_install/snapgear/config/scripts/lxdialog all
make[1]: Entering directory `/home/training1/snapgear_install/snapgear/config/scripts/lxdialog'
cc -DLOCALE -DCURSES_LOC="<ncurses.h>" -c -o checklist.o checklist.c
cc -DLOCALE -DCURSES_LOC="<ncurses.h>" -c -o menubox.o menubox.c
cc -DLOCALE -DCURSES_LOC="<ncurses.h>" -c -o textbox.o textbox.c
cc -DLOCALE -DCURSES_LOC="<ncurses.h>" -c -o yesno.o yesno.c
cc -DLOCALE -DCURSES_LOC="<ncurses.h>" -c -o inputbox.o inputbox.c
cc -DLOCALE -DCURSES_LOC="<ncurses.h>" -c -o util.o util.c
cc -DLOCALE -DCURSES_LOC="<ncurses.h>" -c -o lxdialog.o lxdialog.c
cc -DLOCALE -DCURSES_LOC="<ncurses.h>" -c -o msgbox.o msgbox.c
cc -o lxdialog checklist.o menubox.o textbox.o yesno.o inputbox.o util.o
lxdialog.o msgbox.o -lncurses
make[1]: Leaving directory `/home/training1/snapgear_install/snapgear/config/scripts/lxdialog'
No defaults found
Preparing scripts: functions, parsing...done.
```

This will bring up a menu from which you can choose your settings:

```
uClinux v3.0.0 Configuration

Main Menu
Arrow keys navigate the menu. <Enter> selects submenus --->. Highlighted letters
are hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes features. Press
<Esc><Esc> to exit, <?> for Help. Legend: [*] built-in [ ] excluded <M> module
< > module capable

Vendor/Product Selection --->
Kernel/Library/Defaults Selection --->
---
Load an Alternate Configuration File
Save Configuration to an Alternate File

Select < Exit > < Help >
```

The following steps detail how to select the settings.

To choose the settings:

1. Press **Enter** to drill down into the Vendor/Product Selection menu.
2. Choose Intel as the vendor and IXDP425 as the product.

At this point, the screen should look like the following figure.

```
Vendor/Product Selection
Arrow keys navigate the menu. <Enter> selects submenus --->. Highlighted letters
are hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes features. Press
<Esc><Esc> to exit, <?> for Help. Legend: [*] built-in [ ] excluded <M> module
< > module capable

--- Select the Vendor you wish to target
(Intel) Vendor
--- Select the Product you wish to target
(IXDP425) Intel Products

<Select> < Exit > < Help >
```

3. Press **Esc** to return to the previous menu.
4. Go to the Kernel/Default/Library menu.
5. Select *linux-2.4.x* as the kernel version and *glibc* as the libc library:

```
Kernel/Library/Defaults Selection
Arrow keys navigate the menu. <Enter> selects submenus --->. Highlighted letters
are hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes features. Press
<Esc><Esc> to exit, <?> for Help. Legend: [*] built-in [ ] excluded <M> module
< > module capable

(linux-2.4.x) Kernel Version
(glibc) Libc Version
[ ] Default all settings (lose changes) (NEW)
[ ] Customize Kernel Settings (NEW)
[ ] Customize Vendor/User Settings (NEW)
[ ] Update Default Vendor Settings (NEW)

<Select> < Exit > < Help >
```

6. Click Exit to close the configuration utility.
7. Save the new kernel configuration.

8. Type `make dep` to build the cross compiler.

```
*** End of Linux kernel configuration
*** Check the top-level Makefile for additional configuration.
*** Next, you must run `make dep`.
make[2]: Leaving directory `/home/training1/snapgear_install/snapgear/
config'
make[1]: Leaving directory `/home/training1/snapgear_install/snapgear'
training@tm-athena:~/snapgear_install/snapgear$ make dep
```

When you're compiling code for the reader, the headers to use are in `snapgear/linux-2.3.x/include`.

Creating and Compiling Programs

As you write and compile your programs, follow these steps.

To create, compile, and debug your programs:

1. Compile and debug your program in the PC Linux environment.

It is easier to debug an application on the PC before you cross-compile for the reader.

2. Use `'gdbserver'`, if you need to debug your programs once they are running on the reader.

`'gdbserver'` allows you to debug a program running remotely on the reader, from your local PC.

3. Store data in network byte order.

The reader uses a big-endian kernel. Desktop x86 processors use little-endian architecture. Unless you take this into account, data shared between the two architectures are not interchangeable.

4. Once your program is working in the PC environment, use `Makefile` as a template to cross-compile for the MercuryOS.

5. Check the comments in the file and modify them, if necessary.

6. After the program (`hello.c` as an example) compiles, copy the executable to an NFS-exported directory:

```
cp hello /tmp
```

Note

In order to make your code thread safe you must link with `-pthread`s

Exporting an NFS Mount Point

To export an nfs mount point, add this line to the `/etc/exportfs` file:

```
/tmp *(rw,insecure,no_root_squash,sync)
```

Testing the Program

The simplest way to test the program is to run it from the reader using an NFS mount.

To test your program, use the following procedure:

1. Make sure the PC and the reader are running. To test whether the systems are running:
 - a. Issue a ping *reader_ip_address* from your PC, where *reader_ip_address* is the IP address of the reader.
 - b. Determine the address by using one of the following methods:
 - Configuring the reader to use a static IP address of your choosing
 - Running DHCP and then using the DHCP server
 - Accessing DNS to find the reader by name
 - Using Rendezvous-style services

You should see ping responses from the reader if everything is configured correctly.

2. Telnet to the reader:

```
telnet reader_ip_address.
```

The username is *root* and the password is *blank* or *secure*. (In general, this can vary depending on the reseller that provided your particular reader to you.) You will see a Linux prompt. You are now logged into the reader.

3. Mount the PC's NFS share by issuing the following command:

```
mount -o nolock,vers=2 pc_ip_address:/tmp /mnt.
```

Now your PC's /tmp directory is visible on the reader as /mnt

4. To set permanent mount parameters, add the following line to the /etc/fstab file:

```
pc_ip_address:/tmp /mnt nfs  
rsize=8192,wsize=8192,timeo=14,intr,nolock,mountvers=2
```

5. Run any cross-compiled programs you have put in your /tmp directory. To run the example hello program, simply enter

```
/mnt/hello
```

You see the output of hello on the reader console.

Saving the Program to the JFFS2 File System

The root MercuryOS filesystem runs out of a ramdisk that is loaded from the flash memory device on boot. This filesystem is read/write while the reader is running, however any changes made to the filesystem are gone when you reboot the reader.

The JFFS2 filesystem mounted at /tm is different. Any files you place in the /tm directory or any of its subdirectories are saved to flash and restored when the system is restarted.

ThingMagic binaries are stored in the /tm/bin directory. You can use this directory or create another directory to permanently store your programs. You can copy them to any directory in the /tm filesystem where they are loaded from the flash, the next time the reader is rebooted.

Ensuring Program Starts on Every System Boot (Optional)

Another method to ensure the program is started every time the system boots, you can place it in the inittab file. If you do this, the operating system starts that program when the system boots and restarts it in the event of a crash.

Default inittab

```
demonb:unknown:/bin/demonb
telnetd:unknown:/bin/run_telnetd.sh
portmap:unknown:/bin/portmap -d
tmd:unknown:/tm/bin/tmd
rql:unknown:/tm/bin/rql
ntp:unknown:/bin/run_ntp.sh
webserver:unknown:/bin/run_webserver.sh
rendezvous:unknown:/bin/mDNSResponderPosix -f /var/tmp/rendezvous.conf
sshd:unknown:/bin/run_sshd.sh
```

Modified inittab

```
demonb:unknown:/bin/demonb
telnetd:unknown:/bin/run_telnetd.sh
portmap:unknown:/bin/portmap -d
tmd:unknown:/tm/bin/tmd
rql:unknown:/tm/bin/rql
ntp:unknown:/bin/run_ntp.sh
webserver:unknown:/bin/run_webserver.sh
rendezvous:unknown:/bin/mDNSResponderPosix -f /var/tmp/rendezvous.conf
sshd:unknown:/bin/run_sshd.sh
userprogram:unknown:/tm/bin/userprogram
```

**C A U T I O N !**

Do not delete any lines from this file. The reader may not work properly if all the programs do not start.

API Reference

This section describes the high-level Reader APIs available on the fixed RFID readers running the MercuryOS. The APIs are used to configure and control the fixed RFID readers from an application running on the reader or from an application running on a remote host that has a Sun[®] Microsystems Remote Procedure Call (RPC) library.

On Windows clients, the API uses the Open Network Computing Remote Procedure Call (ONC RPC) library. The ONC RPC library included in the developer's kit as a dynamic link library (DLL) was developed by the European Synchrotron Radiation Facility and is governed by the LGPL license.

This section describe the APIs that control the following reader activities:

- ◆ [Connecting and Disconnecting](#)
- ◆ [Multiple Protocol/Multiple Antenna Reads](#)
- ◆ [Individual Tag Commands](#)
- ◆ [GPIO Commands](#)
- ◆ [Reader Configuration](#)
- ◆ [SNMP](#)

Connecting and Disconnecting

The Mercury fixed reader API is session-oriented. In this architecture, a session is established to the reader and the returned session handle is used in subsequent API calls to that reader.

Commands for Connecting/Disconnecting to the Reader

Command	Description
<code>reader_handle_t open_reader</code>	Opens a connection to the reader.
<code>void close_reader</code>	Closes a connection to the reader.

Connecting/Disconnecting Data Structure

The following short program illustrates connecting to and disconnecting from a reader.

```
main (int argc, char **argv)
{
    reader_handle_t h_reader;
    const char *hostname = "testreader.thingmagic.com";
    char *osv;

    if ((h_reader = open_reader(hostname)) == NULL) {
        reader_perror (hostname);
        exit (1);
    }

    if ((osv = get_os_version(h_reader)) == NULL) {
        reader_perror ("get_os_version");
    } else {
        printf ("%s has OS version '%s'\n", hostname, osv);
    }

    close_reader (h_reader);
}
```

Note

If a connection cannot be established, failure is indicated by a NULL return value. The specific error code is returned in the global variable `reader_erno`. These error codes are listed in `m4api.h`, and may be printed out to the standard error stream by calling `reader_perror()`.

Functions

Open a Connection to the Reader

```
reader_handle_t open_reader (char *hostname);
```

This command opens a connection to the reader specified as *hostname* and returns a **reader_handle_t**. If the call fails, a NULL is returned.

The *hostname* parameter can be a DNS name or an IP address. If the *hostname* parameter is a device name and cannot be resolved using the DNS resolution, the call fails.

An address of “127.0.0.1” (localhost) signals that the connection to the reader is local to allow other API routines to use a more efficient shared memory mechanism for interprocess communications when possible.

Syntax

The following example opens a connection to the reader and assigns that connection's handle to **h_reader** (or prints an error message using **reader_perror()** if the call was unsuccessful).

Example:

```
if ((h_reader = open_reader(hostname)) == NULL) {  
    reader_perror (hostname);  
    exit (1);}
```

Parameters

Parameter	Description
<i>hostname</i>	DNS name or IP address of reader

Returns

Success: reader_handle_t of the connection.

Failure: NULL

Note

When connecting to multiple readers with multiple threads each thread should make its own call to `open_reader` and use its own reader handle.

Close a Connection to the Reader

```
void close_reader (reader_handle_t h);
```

This command closes the connection to the Mercury reader and discards the reader handle after this call.

Syntax

The following code closes a connection to the reader that was opened previously and whose handle was assigned to **h_reader**.

Example:

```
close_reader (h_reader);
```

Parameters

Parameter	Description
<i>h</i>	The handle of the connection to be closed

Returns

None.

Multiple Protocol/Multiple Antenna Reads

The Reader API provides bulk reads that fill a large data base called the tag data base. All of the other tag manipulation commands deal with individual tags. When searching for multiple tags over multiple antennas using multiple protocols, the following procedure should be used (the specific commands are explained following the procedure):

To search for multiple tags:

1. Reset the tag data base using the **tagdb_reset_db** function (except to keep a running record of reads from previous searches).
2. Call a function to initiate a search (usually done via **search_start()** or **detailed_read_start()**, but there are several wrapper functions).
3. Get the results by popping tag records out of the tag data base using the **tagdb_pop()** or **tagdb_consume()** command until there are no more valid records.

Searching for Tags and Resetting the Data Base

The two high-level operations the API exposes are searching for tags and resetting the tag data base. The **search** command starts an RF search for tags using the specified protocols and antennas and populates the tag data base with the results.

Multiple Protocols/Multiple Antenna Commands

Command	Description
int tagdb_reset_db	Resets the tag data base.
search_uid_t search_start	Populates the tag data base.
int operation_status	Gets the status of a search.
Deprecated: int tagdb_pop	Returns a record then removes it from the data base.
int tagdb_consume	Returns a record then removes it from the data base.
search_handle_t auto_search_start	Initiates an autosearch operation.
int search_active	Gets the status of an autosearch operation.
int auto_search_stop	Stops an autosearch operation.

Command	Description
search_uid_t detailed_read_start	Initiates a search operation.
int detailed_read_stop	Stops a search.
search_uid_t converyor_read_start	Initiates a conveyor search operation.
long tags_available	Fetches the available tags in the data base.

Tag Data Base Data Structure

The tag data base is a data structure that maintains a record of tags read. When the tag data base becomes completely filled, the oldest records are overwritten (after 262143 records).

There are two mutually-exclusive data structures that can be used to perform a search. The relatively simple **search_params_t** structure can be used with the function **search_start()** to perform many types of searches. For a greater granularity of control, the more complex search **TupleObj_t** structure can be used with the function **detailed_read_start()**.

- ◆ The following shows the **search_params_t** structure:

```
typedef struct {
    u_short protocol_list_length; /* length of protocol list */
    u_short protocol_list[16];   /* protocols to use          */
    u_short antenna_list_length; /* length of ant. List      */
    u_short antenna_list[16];   /* antennas to search over */
    u_short timeout_ms;        /* timeout in milliseconds */
} search_params_t;
```

- ◆ The following shows the **search TupleObj_t** structure:

```
typedef enum
{
    TTYPE_NULL=0,
    TTYPE_PROTOCOL,
    TTYPE_ANTENNA,
    TTYPE_POWER,      /*Currently Unused */
    TTYPE_DURATION,
    TTYPE_LAST_VALUE
}tupleType_t;
```

```
enum searchMode_t {
    SM_NULL = 0,
    SM_STANDARD,
    SM_QUICK,
    SM_MINIMAL
    SM_LAST_VALUE
};
typedef enum searchMode_t searchMode_t;

struct searchMode_data_t {
    u_short searchMode;
    union {
        quickSearch_params_t quickSearchParams;
        long value[8];
    } searchMode_data_u;
};
typedef struct searchMode_data_t searchMode_data_t;

typedef struct tupleData_t {
    u_short type;
    union {
        long protocol_id;
        durationObj duration;
        long value;
    } tupleData_u;
}tupleData_t;

typedef struct tuple_t {
    tupleData_t data[16];
}tuple_t;

typedef struct tupleObj_t {
    u_short setLength;
    u_short tupleSet[16];
    struct {
        u_int tupleVal_len;
        tuple_t *tupleVal_val;
    } tupleVal;
}tupleObj_t;
typedef struct tagopObj_t {
    u_short dtype;
    u_short dataAddress;
    u_short dataLength;
    u_short dataValue[2048];
```

```
}tagObj_t;

typedef struct tagDataObj_t {
    u_short protocol;
    u_short antenna;
    u_short id_length;
    u_short id_value[16];
    u_long frequency;
    u_long dspmicrosLow;
    u_long lqi;
    tagObj_t *tagInfo;
}tagDataObj_t;

typedef struct atuple_t {
    struct {
        u_int antenna_len;
        u_long *antenna_val;
    } antenna;
}atuple_t;

typedef struct antennaSet_t {
    struct {
        u_int groups_len;
        atuple_t *groups_val;
    } groups;
    struct {
        u_int sequentialSet_len;
        u_long *sequentialSet_val;
    } sequentialSet;
    struct {
        u_int simultaneousSet_len;
        u_long *simultaneousSet_val;
    } simultaneousSet; /*Currently Unused */
}antennaSet_t;

struct antGroupEvent_t {
    unsigned char groupIndex;
    unsigned char useCdca;
    unsigned char carrierDetectCount; /* Currently unused */
    unsigned long carrierDetectThreshold;
    unsigned short gpioLineMask;
    unsigned short gpioLineState;
};
```

```
struct synchObjNtpData_t
{
    unsigned short offset;
};
typedef struct synchObjNtpData_t synchObjNtpData_t;

struct synchObjRegTimeoutData_t
{
    short powerThreshold;
};
typedef struct synchObjRegTimeoutData_t synchObjRegTimeoutData_t;

struct synchObjNtpRegTimeoutData_t
{
    unsigned short offset;
    short powerThreshold;
};
typedef struct synchObjNtpRegTimeoutData_t synchObjNtpRegTimeoutData_t;

struct synchObjData_t
{
    unsigned short type;
    union
    {
        struct synchObjNtpData_t ntpData;
        struct synchObjAutoData_t autoData;
        struct synchObjRegTimeoutData_t regTimeoutData;
        struct synchObjNtpRegTimeoutData_t ntpRegTimeoutData;
        long value[8];
    }
    synchObjData_u;
};
typedef struct synchObjData_t synchObjData_t;

typedef struct searchTupleObj_t {
    antennaSet_t antennas;
    tupleObj_t global;
    tupleObj_t cycle;
    durationObj_t cycleDuration;
    u_short cycleOrder;
    u_short sequenceOrder;
    u_short synchType;
    u_short hopType;
    u_short searchMode;
```

```
    struct synchObjData_t synchData;
    struct searchMode_data_t searchModeData;
    struct {
        u_int antGroupEvents_len;
        struct antGroupEvent_t *antGroupEvents_val;
    } antGroupEvents;
    tagDataObj_t *tagInfo;
}searchTupleObj_t;

typedef enum {
    TA_TAGS_TO_POP,
    TA_TAGS_TO_CONSUME
}TAG_AVAILABILITY_T;

typedef enum
{
    MINIMUM_RF_OUTPUT    =0,
    MAXIMUM_READTIME    =1
} searchModeVariant_t;

struct quickSearch_params_t {
    unsigned short quickSearchObjective;
    int interval;
    int depth;
    unsigned int rfPowerLevel;
    unsigned int quickSearchAntennas;
};
typedef struct quickSearch_params_t quickSearch_params_t;
typedef enum
{
    DUR_NULL=0,          /*Allow TMD to choose a default*/
    DUR_INFINITE=0,     /*Never Timeout*/
    DUR_TIME_AND_COUNT, /*Timeout and Countdown must expire */
    DUR_TIME_OR_COUNT, /*Timeout or Countdown can expire */
    DUR_TIME_ONLY,     /*Only the Timeout must expire to end search*/
    DUR_COUNT_ONLY,    /*Only Countdown must expire to end search*/
    DUR_LAST_VALUE
}durationEnum_t;

typedef struct{
    durationEnum type; /* TIME or COUNT */
    u16 time; /* timeout in ms for time-based operation */
    u16 count; /* iteration count for iteration-based operation */
} durationObj_t;
```

```
typedef enum
{
    ORDER_NULL=0,
    ORDER_IN_ORDER, /*Search in the order given*/
    ORDER_MIN_TOTAL_TIME, /*Attempt to optimize whole search order for
time*/
    ORDER_MIN_PROTOCOL_TIME, /*Attempt to optimize protocol search
order*/
    ORDER_MIN_ANTENNA_TIME, /*Attempt to optimize antenna search
order*/
    ORDER_LAST_VALUE
}seqObj_t;

typedef enum
{
    SYNCH_NULL=0,
    SYNCH_NTP,
    SYNCH_LBT, /* Currently Unused */
    SYNCH_GPIO, /* Currently Unused */
    SYNCH_REGULATORY_TIMEOUT,
    SYNCH_NTP_REGULATORY_TIMEOUT,
    SYNCH_AUTO, /* Currently Unused */

    SYNCH_RESERVED_7,
    SYNCH_RESERVED_8,
    SYNCH_LAST_VALUE
}synchObj_t;

typedef enum
{
    HOP_NULL=0,
    HOP_FREQUENCY,
    HOP_LFSR, /*Currently Unused */
    HOP_CONVEYOR,
    HOP_LAST_VALUE
}hopObj_t;

typedef enum
{
    CYCLE_NULL,
    CYCLE_ANTENNA_GROUP_THEN_PROTOCOL,
    CYCLE_PROTOCOL_THEN_ANTENNA_GROUP,
```

```
        CYCLE_LAST_VALUE
    }cycleObj_t;
```

- ◆ The following enumeration is used by both the **search_params_t** and the **searchTupleObj_t** structures to specify searches:

```
enum PROTOCOL_IDS {
    PROTOCOL_ID_NULL                = 0,
    PROTOCOL_ID_CC915               = 1,
#define PROTOCOL_ID_EPC1          PROTOCOL_ID_CC915
    PROTOCOL_ID_CC1356              = 2,
    PROTOCOL_ID_ISO15693            = 3,
    PROTOCOL_ID_ISO14443            = 4,
    PROTOCOL_ID_CC1356_PHILIPS      = 5,
    PROTOCOL_ID_CC915V0             = 6,
    PROTOCOL_ID_CC915V1             = 7,
    PROTOCOL_ID_I186B               = 8,
    PROTOCOL_ID_EPC0                = 9,
    PROTOCOL_ID_DIAGNOSTIC          = 10,
    PROTOCOL_ID_LBT                  = 11,
    PROTOCOL_ID_GEN2                 = 12,
    /* Obtain the range of valid protocol IDs */
    PROTOCOL_ID_LAST_VALUE
};
```

Functions

Reset the Tag Data Base

```
int tagdb_reset_db (reader_handle_t h);
```

Resets the tag data base, clearing all tag records.

Syntax

The example resets the tag data base on the reader with the handle h1:

It is typically a good idea to call this function immediately before calling **search_start()** or **detailed_search_start()** to start a new search.

```
if (tagdb_reset_db (h1) < 0) {
    reader_perror ("tagdb_reset_db (");
```

}

Parameters

Parameter	Description
<i>h</i>	The handle of the reader to be queried.

Returns

Success: ERROR_SUCCESS

Failure: error code (see [Status and Error Codes](#))

Initiate a Simple Search Operation

[search_uid_t search_start \(reader_handle_t h, search_params_t *s\);](#)

This program populates the tag data base by starting the reader searching for RFID tags using the specified protocols, antennas, and timeout. If the timeout value is too low, the reader may not have a chance to populate the tag data base with all the tags that can be read. It is important to test the location in which the reader and antennas are deployed to find the best timeout value.

Note

The **search_start()** function and the **search_params_t** structure, while very easy to use, do not allow all of the capabilities of the reader to be used. For a greater level of control, see the **detailed_read_start()** function, and accompanying **searchTupleObj_t** structure.

This function is asynchronous and its return value is the unique ID of the search operation. After a search has been started, use the **operation_status()** function to poll for the status of the search or to determine when the search is complete.

Syntax

The following example starts a multi-protocol (EPC0 and GEN2) search on the reader referenced by h1. The timeout for the whole search will be 1000 ms (to specify different timeouts for each protocol, **detailed_read_start()** must be used).

```
/* search using EPC0 and GEN2 tag protocols */
s1.protocol_list_length = 0;
s1.protocol_list[s1.protocol_list_length++] = PROTOCOL_ID_EPC0;
s1.protocol_list[s1.protocol_list_length++] = PROTOCOL_ID_GEN2;
```

```
/* search on all available antennas */
s1.antenna_list_length = 0;
/* search times out after 1000 ms*/
s1.timeout_ms = 1000;

if ((u1 = search_start (h1, &s1)) < 0) {
    reader_perror ("search_start (");
```

Parameters

Parameter	Description
<i>h</i>	The reader handle.
<i>s</i>	The structure holding the search parameters.

Returns

Success: uid of the search process (used in subsequent calls).

Failure: Error code (see [Status and Error Codes](#))

Get the Status of a Search

```
int operation_status (reader_handle_t h, uid u);
```

This function normally is used within a polling loop, after a search has already been started. This command gets the status of a search process. This is necessary, since the functions that start a search process are non-blocking.

Note

This function has been optimized so that if it is called for a search on the local reader, the status look up is performed via shared memory and cannot incur the penalties associated with network RPC (de)marshalling.

Syntax

In the following example loop, assume that a connection to the reader has already been established (its handle assigned to *h1*), and a search has been started on that reader (the search's UID assigned to *u1*):

```
while (1)
{
    int status;
    status = operation_status (h1, u1);
    if (status == OPERATION_DONE ||
        status == OPERATION_TIMED_OUT) {
        break;
    }
    SLEEP (10);
}
```

Parameters

Parameter	Description
<i>h</i>	The reader handle.
<i>u</i>	The unique identifier for the process.

Returns

Queued: OPERATION_WAITING_TO_RUN

In progress: OPERATION_IN_PROGRESS

Done: OPERATION_DONE or OPERATION_TIMED_OUT

Failure: Error Code (see [Status and Error Codes](#))

Return the current record from the data base [Deprecated]

Deprecated: `int tagdb_pop (reader_handle_t h, tagdb_record_t *tp);`

This function fills the structure pointed to by the `*tp` parameter with the first tag record from the tag data base. Each time the function is called, the current record is returned and then removed from the data base. When the record returned has an invalid status (`tagdb_record_t.status == 0`), there are no more records to return.

This function is now a wrapper around **tagdb_consume**. See [int tagdb_consume \(reader_handle_t h, tagdb_record_v3_t *tp\);](#)

Syntax

This function uses the following structure to return the requested information:

```
typedef struct {
    int      status;          /* 1 if the record is valid          */
    u_short  antenna_id;     /* antenna the tag was read from    */
    u_short  protocol_id;    /* protocol of the tag              */
    u_short  num_bits;       /* number of bits in the tag        */
    u_short  tag_id[16];     /* the tag id                       */
    u_short  read_count;     /* number of times the tag was read */
    u_int    khz;            /* Operating Frequency              */
    u_int    tv_sec;         /* Unix Timestamp                   */
    u_int    tv_usec;       /* Unix Timestamp                   */
    u_int64_t dspmicros;     /* DSP Timestamp - microseconds since start */
} tagdb_record_t;
```

Parameters

Parameter	Description
<i>h</i>	The reader handle.
<i>tp</i>	A user-allocated tag database record.

Returns

Success: 0

No more records: `tagdb_record_t` structure has its status set to 0 (invalid record).

Failure: Error code (see [Status and Error Codes](#))

Return the current record and remove it from the data base

```
int tagdb_consume (reader_handle_t h, tagdb_record_v3_t *tp);
```

This function fills the structure pointed to by the *tp* parameter with the first tag record from the tag data base. Each time the function is called the current record is returned and then removed from the data base. When the record returned has an invalid status (`tagdb_record_v3_t.status == 0`), there are no more records to return.

Syntax

This function uses the following structure to return the requested information:

```
typedef struct {
    int      status;          /* 1 if the record is valid          */
    u_short  antenna_id;     /* antenna the tag was read from    */
    u_short  protocol_id;   /* protocol of the tag              */
    u_short  num_bits;      /* number of bits in the tag        */
    u_short  tag_id[16];    /* the tag id                       */
    u_short  read_count;    /* number of times the tag was read */
    u_int    khz;           /* Operating Frequency              */
    u_int    tv_sec;        /* Unix Timestamp seconds           */
    u_int    tv_usec;      /* Unix Timestamp useconds          */
    u_long   dspmicros;     /* DSP Timestamp - useconds since start */
    u_short  hash;         /* Tag CRC                          */
    u_short  lqi;          /* Link Quality Indicator -- Unused  */
} tagdb_record_v3_t;
```

Parameters

Parameter	Description
<i>h</i>	The reader handle.
<i>tp</i>	A user-allocated tag data base record.

Note

“lqi” is currently unused, and is specified as a reservation.

The function is normally called within in a loop that terminates when a tag record is returned that has a value of 0 in its **status** field.

In the following example, the loop uses a print function to display all of the tags in the database of the reader whose handle is `h1`.

For Example:

```
tagdb_record_v3_t tr1;

while (tagdb_consume(h1, &tr1) == ERROR_SUCCESS)
{
    if (tr1.status == 0)
    {
        break;
    }
    else
    {
        printf_tagdb_record (&tr1);
    }
}
```

Returns

Success: 0 No more records: **tagdb_record_t** structure has its status set to 0 (invalid record).

Failure: Error code (see [Status and Error Codes](#))

Initiate an autosearch operation [Deprecated]

Deprecated: [search_handle_t auto_search_start \(reader_handle_t h, search_params_t *s, double T0, double period, double offset, double T1\);](#)

Initiates an autosearch operation based on the given search parameters. The automatic search is initiated at the given time *T0* (or at the next whole-second quantum if *T0* == 0), is repeated with the given period and at the given offset within the search period, and is terminated at the time *T1* (or never if *T1* == 0).

The return value is a handle for the autosearch operation. Use the **search_active** function to determine whether an autosearch is in progress or complete. See [Deprecated: int search_active \(search_handle_t sh\);](#)

This function uses SIGALRM to initiate the search at time *T0*. When *T0* is reached, a **detail_read_start** is used to produce the looping search.

Parameters

Parameter	Description
<i>h</i>	The reader handle
<i>s</i>	A structure holding the search parameters
<i>T0</i>	The time to start the search.
<i>period</i>	The time period at which to repeat searches.
<i>offset</i>	The time offset to initiate searches.
<i>T1</i>	The time to end searches.

Returns

Success: `search_handle_t` of the search process (used in subsequent calls).

Failure: NULL

Get the status of an autosearch operation [Deprecated]

Deprecated: `int search_active (search_handle_t sh);`

This command gets the status of an autosearch process.

Parameters

Parameter	Description
<i>sh</i>	The handle for an autosearch operation

Returns

In progress: 1

Done: 0

Failure: -1

Stop an AutoSearch Operation [Deprecated]

Deprecated: `int auto_search_stop (search_handle_t sh);`

This command stops an autosearch process currently in progress.

Parameters

Parameter	Description
<i>sh</i>	A handle for the autosearch operation that is stopped.
<i>tp</i>	A user-allocated tag database record.

Returns

Initiate a detailed search

`search_uid_t detailed_read_start (reader_handle_t h, searchTupleObj_t *p);`

This function initiates a detailed search operation based on the given search parameters.

The return value is the unique ID of the search operation. Use the **operation_status** function to determine when the search is complete.

The *searchTupleObj_t* and its associate data structures are complex and warrant some explanation. See [Tuple Search Mode Object Parameters](#).

Duration Parameters

durationEnum_t	Duration Type
DUR_NULL	Specifies that tmd should use a default duration (currently infinite)
DUR_INIFINITE	Specifies an infinite search.
DUR_TIME_AND_COUNT	Specifies that both the time and count value must expire to fulfill the duration
DUR_TIME_OR_COUNT	Specifies that either the time or count value must expire to fulfill the duration
DUR_TIME_ONLY	Specifies that only the time value must expire to fulfill the duration

DUR_COUNT_ONLY	Specifies that only the count value must expire to fulfill the duration. Count durations for protocols have special meaning, and currently only supported when attempting to perform “Minimal Searches”.
----------------	--

Search Parameters

searchMode_t	Search Mode
SM_NULL	Search Mode unspecified
SM_STANDARD	Search is the traditional search utilizing anticollision and full search depth
SM_QUICK	currently unused
SM_MINIMAL	Search is very minimal and eliminates as much overhead as possible. Anti-collision logic is removed. This search mode is ideal only when there is only a single tag in the field. A minimal search mode requires the use of count based durations for the protocols.

Sequence Parameters

seqObj_t	Sequence Order
ORDER_NULL	Reordering of the search is unspecified
ORDER_IN_ORDER	Reordering of the search is not to be done
ORDER_MIN_TOTAL_TIME	Reordering of the whole search is to be done in a way that allows the search to complete in the least amount of time
ORDER_MIN_PROTOCOL_TIME	Reordering of the protocols in the search is to be done in a way that allows the search to complete in the least amount of time
ORDER_MIN_ANTENNA_TIME	Reordering of the antennas in the search is to be done in a way that allows the search to complete in the least amount of time

Synchronization Parameters

synchObj_t:	Synch Type
SYNCH_NULL	No Synchronization is done
SYNCH_NTP	Synchronize search via NTP based on slicing time by the total search time plus an offset.
SYNCH_LBT	Currently Unused
SYNCH_GPIO	Currently Unused.
SYNCH_REGULATORY_TIMEOUT	Synchronize regulatory timeouts via NTP.
SYNCH_NTP_REGULATORY_TIMEOUT	Combination of SYNCH_NTP and SYNCH_REGULATORY_TIMEOUT
SYNCH_AUTO	Currently Unused
SYNCH_RESERVED_7	Currently Unused
SYNCH_RESERVED_8	Currently Unused

Hop Style Parameters

hopObj_t	Hop Type
HOP_NULL	Hop style is unspecified
HOP_FREQUENCY	Hop style uses the frequency hop table
HOP_LFSR	Hop style uses LFSR algorithm for frequency hopping (currently unused)
HOP_CONVEYOR	Hop style commonly used in high speed environments, frequency hops only occur at a slot swap.

cycleObj_t	Cycle Order
CYCLE_NULL	Cycle Priority is unspecified
CYCLE_ANTENNA_GROUP_THEN_PROTOCOL	Cycle over the antenna groups then over the protocols
CYCLE_PROTOCOL_THEN_ANTENNA_GROUP	Cycle over the protocols then over the antenna groups

The cycling described by the **cycleObj_t**, occurs when a protocol timeout occurs. It answers the question of what to do when the duration specified by the protocol has been reached. It is important to recognize that this object is referring to antenna groups and not individual antennas.

Tuple Types

tupleType_t	Tuple Type
TTYPE_NULL	Tuple type is unspecified
TTYPE_PROTOCOL	Tuple specifies a protocol
TTYPE_ANTENNA	Tuple specifies an antenna
TTYPE_POWER	Currently Unused
TTYPE_DURATION	Tuple specifies a durationObj_t

Tuple Object Types

tupleObj_t	Tuple Type
setLength	Length of the set describing the tupleVal
tupleSet	Set describing the tupleVal
tupleVal.tupleVal_len	Length of the tupleVal.tupleVal_val
tupleVal.tupleVal_val	Set of tuples

Tuple Data Types

tuple_t	Tuple Data Type
data	The Data array
tupleData_t.type	The type of data in this element (automatically filled based on the corresponding tupleSet value)
tupleData_t.tupleData_u	The actual data element union

Tuple Antenna Types

atuple_t	Antenna Type
antenna.antenna_len	Length of the described antenna array
antenna.antenna_val	Array of antenna IDs (0 based)

Antenna Set Events

antennaSet_t	Antenna Set Event
groups.groups_len	The number of antenna groups in groups.groups_val
groups.groups_val	The array of antenna groups
sequentialSet.sequentialSet_len	The number of sets in sequentialSet.sequentialSet_val
sequentialSet.sequentialSet_val	The array describing the order of the antenna groups in the Sequential Antenna Set
simultaneousSet	Currently unused

Antenna Group Events

antGroupsEvent_t	Antenna Group Event
groupIndex	The Index of the antenna group this is referencing
useCdca	Flag describing whether to allow Carrier Detect Collision Avoidance
carrierDetectCount	Currently Unused
carrierDetectThreshold	Power level detection threshold. This value is used in determining when a carrier is detected.
gpioLineMask	The GPIO line mask used in signal GPIO based RF_OFF
gpioLineState	The state of the GPIO lines required to trigger GPIO based RF_OFF.
antGroupEvents_len	The number of antenna groups events in object
antGroupEvents_val	The array of antenna group events

Synchronous Data Events

<code>_synchObjData_t</code>	Synch Data
<code>data</code>	The Data array
<code>synchObjData_t.type</code>	The type of data in this element (automatically filled based on the corresponding <code>searchTupleObj</code> value)
<code>synchObjData_t.synchObjData_u</code>	The actual data element union

Quick Search Mode Variants

<code>searchModeVariant_t</code>	Search Mode
<code>MINIMUM_RF_OUTPUT</code>	Optimize Quick Search to minimize RF output. If no tags are found, then reader will not transmit for a specified period of time.
<code>MAXIMUM_READTIME</code>	In a multi-protocol search, optimizes Quick Search such that if a tag of a particular protocol is not found, time from its statically allocated time-slot will be reallocated for reading of other protocols' tags.

Quick Search Mode Parameters

<code>quickSearch_params_t</code>	Search Mode
<code>quickSearchObjective</code>	Which variant of quick search to use (one of enum <code>searchModeVariant_t</code>)
<code>interval</code>	Interval between tag presence detection retries (only relevant when <code>quickSearchObjective</code> is <code>MINIMUM_RF_OUTPUT</code>).
<code>depth</code>	Number of pings or scrolls used to determine whether or not tags are present (N/A for EPC1).
<code>rfPowerLevel</code>	RF power level used when quick search is determining whether tags are present (this only affects the RF level used when determining tag presence, not the RF level used to read tags). A value of 0 will result in the default power level being used.
<code>quickSearchAntennas</code>	0 specifies search on all antennas in a group, otherwise just use the first antenna.

Tuple Search Mode Object Parameters

searchTupleObj_t	Search Mode
antennas	The antennas involved in this search
global	Global parameters for this search, currently only TTYPE_DURATION is supported
cycle	The search cycle used in finding the tags, primarily the protocols to use in the search.
cycleDuration	The duration of all of the entire search
cycleOrder	The ordering algorithm used for the protocols and antenna groups in the search cycles
sequenceOrder	The sequencing used within each search cycle, e.g. MIN_TOTAL_TIME
synchType	How are searches synchronized?
synchData	Offset and/or power threshold data pertaining to the synchronization type.
hopType	The frequency hopping algorithm used
searchMode	The search Mode used for the entire search
antGroupEvents	Special trigger events associated with antenna groups
tagInfo	Currently unused

Parameters

Parameter	Description
<i>h</i>	The reader handle
<i>p</i>	A structure holding the search parameters

Returns

Success: uid of the search process (used in subsequent calls).

Failure: Error code (see [Status and Error Codes](#))

Stop a detailed search

```
int detailed_read_stop (reader_handle_t h, search_uid_t sh);
```

This function stops the search of the uid specified in *sh*. The value of 0x00 is a special value specifying that all currently submitted search operations should be stopped/preventing from starting.

The return value is the uid that is stopped.

Parameters

Parameter	Description
<i>h</i>	The reader handle
<i>sh</i>	The handle for an autosearch operation

Returns

Success: *search_uid_t* of the search process that was stopped. If 0 is given as *sh*, then the current *search_uid_t* is returned.

Failure: -1

Initiate a conveyor search operation

```
search_uid_t conveyor_read_start (reader_handle_t h, search_params_t *p);
```

This function initiates a conveyor search operation based on the given search parameters. This will internally call a **detailed_read_start**, and provides a convenient way of performing the most common style of searching in a conveyor belt environment.

Syntax

Aside from the protocol and antennas, the following values are used when passing the search onto `detailed_read_start`.

```
global_params[0].data[0].type=TTYTYPE_DURATION;  
global_params[0].data[0].tupleData_u.duration.type=DUR_COUNT_ONLY;  
global_params[0].data[0].tupleData_u.duration.count=2;  
/* Search until stopped */  
search_params.cycleDuration.type=DUR_INFINITE;  
search_params.cycleOrder=CYCLE_PROTOCOL_THEN_ANTENNA_GROUP;  
search_params.sequenceOrder=ORDER_MIN_TOTAL_TIME;  
search_params.synchType=SYNCH_NULL;  
search_params.hopType=HOP_CONVEYOR;
```

```
search_params.searchMode=SM_MINIMAL;
```

The return value is the unique ID of the search operation. Use the **operation_status** function to determine when the search is complete.

Parameters

Parameter	Description
<i>h</i>	The reader handle
<i>p</i>	The structure holding the search parameters

Returns

Success: uid of the search process (used in subsequent calls)

Failure: Error code (see [Status and Error Codes](#))

Stop a conveyor search operation

```
int conveyor_read_stop (reader_handle_t h, search_uid_t sh);
```

This function stops the search of the *search_uid_t* specified in *sh*. The value of 0x00 is a special value specifying that all currently submitted search operations should be stopped/preventing from starting.

The return value is the uid that is stopped.

Parameters

Parameter	Description
<i>h</i>	The reader handle
<i>sh</i>	The search_uid_t of the search to stop

Returns

Success: *search_uid_t* of the search process that was stopped. If 0 is given as *sh*, then the current uid is returned.

Failure: -1

Fetch the Available Tags in the Data Base

`long tags_available (reader_handle_t h, int atype);`

This command fetches the number of tags that are available in the database. The availability type describes the availability that is being searched, that is consumability.

Note

The `tagdb_pop()` function has been deprecated, so `TA_TAGS_TO_CONSUME` is currently the only valid availability type.

Parameters

Parameter	Description
<i>h</i>	The reader handle
<i>atype</i>	Availability type (TAG_AVAILABILITY_T)

Returns

Success: Number of available tags

Failure: -1

Individual Tag Commands

The Reader API supports a number of individual tag commands. Except for the data read command, all of these commands manipulate the tag state.

Tag Commands

Command	Description
tagid_write	Writes a tag identification
tagid_lock	Locks a tag identification
tag_kill	Kills a tag identification
tag_passwd	Sets a password
tagdata_read	Reads the tag data
tagdata_write	Writes the tag data
tagdata_lock	Locks the tag data

Tag Operation Data Structure

The following tag operations data structure is used for each of these commands:

```
typedef struct {
    u_short protocol;           /* "air" protocol to use */
    u_short antenna;           /* reader antenna port to use */
    u_short id_length;         /* length of the Tag ID */
    u_short id_value[16];      /* the tag ID */
    u_short data_type;         /* TBD */
    u_short data_address;      /* " */
    u_short data_length;       /* length of data */
    u_short data_value[2048]; /* data */
} tagop_args_t;
```

Each of the individual tag commands is described in more detail in this section.

Functions

Write a new tag id

```
int tagid_write(reader_handle_t h, tagop_args_t *a);
```

The **tagid_write** command writes a new ID to a tag. It is important to only have one tag in the RF field or all tags in the RF field will be given the same tag ID or other errors can occur.

Syntax

The following example writes the 64-bit value 0x1234567890abcdef to an EPC1 tag on the first antenna on the reader:

```
tagop_args_t tagop_args;

tagop_args.protocol = PROTOCOL_ID_EPC1;
tagop_args.antenna = 1;
tagop_args.data_type = 0;
tagop_args.data_address = 0;
tagop_args.data_length = 0;
tagop_args.data_value[] = 0;

if (ERROR_SUCCESS ==
    tagid_write(h1, &tagop_args)
    {
    printf("tagid_write successful\n");
    }
else
    {
    printf("tagid_write failed\n");
    }
```

Gen2

To singulate a tag out of a group, place the current EPC into the “data_value” array and set the flag of **a.data_type** bits to 8. The tag is limited to 256 bits which is the size of the data_value array.

To add a password to access the password register of the tag, store the password in the “data_value” array. It appears after the id (if one is supplied) in the “data_value” array. To signal the presence of the password in the array, set the flag of **a.data_type** bits to 4. Adjust the data length to reflect the additional data.

Notice that not all of the fields are needed in the *tagop_args_t* structure when writing the tag ID.

Parameters

Parameter	Description
<i>h</i>	The reader handle
<i>a</i>	A tag operation's data structure with the following fields set: a.protocol : protocol of the tag a.antenna : antenna to use (1 based) a.data_length : non-zero

Returns

Success: 0

Failure: Error code ([Status and Error Codes](#))

Lock a tag's id

```
int tagid_lock (reader_handle_t h, tagop_args_t *a);
```

This command locks a tag's ID for the GEN2 protocol only.

a.data_type: Fills in the "mask" value as specified in the Gen2 specification [here](#). Memory banks 0,1, and 2 are considered tagid data that can be locked.

Note

Memory bank 0 is used for the passwords.

a.data_value[0]: This is used to fill in the "action" value as specified on the Gen2 specification [here](#).

a.data_value[1]: High 16 bits of the password

a.data_value[2]: Low 16 bits of the password

Parameters

Parameter	Description
<i>h</i>	The reader handle
<i>a</i>	A tag operation's data structure with the following fields set: a.protocol : protocol of the tag a.antenna : antenna to use (1 based) a.id_length : length of the tag ID (in bits) a.id_value : ID of the tag a.data_type : page mask for ID section to lock (GEN2 only) a.data_length : length of the data in data_value a.data_value : action and password (GEN2 only)

Returns

Success: 0

Failure: Error code (negative value)

Kill a tag

```
int tag_kill (reader_handle_t h, tagop_args_t *a);
```

This command kills the tag matching the given tag ID.

Parameters

Parameter	Description
<i>h</i>	The reader handle
<i>a</i>	A tag operation's data structure with the following fields set: a.protocol : protocol of the tag a.antenna : antenna to use (1 based) a.id_length : length of the tag ID (in bits) a.id_value : ID of the tag a.data_length : length of password (depends on protocol) a.data_value : password

Returns

Success: 0

Failure: Error code (negative value)

Set a tag password

```
int tag_passwd (reader_handle_t h, tagop_args_t *a);
```

Sets a tag's password.

Gen2

To add a password to **Set** commands, place the password after the new password in the "**a.data_value**" array.

To signal the presence of the password, set the flag bits of **a.data_type** to 4.

Adjust **data_length** to reflect the additional data.

Parameters

Parameter	Description
<i>h</i>	The reader handle
<i>a</i>	A tag operation's data structure with the following fields set: a.protocol : protocol of the tag a.antenna : antenna to use (1 based) a.id_length : length of the tag ID (in bits) a.id_value : ID of the tag a.data_type : 0 specifies a 'kill' password 1 specifies an 'access' password (GEN2 only) Otherwise, assume kill password a.data_length :length of password (in bits) a.data_value :password

Returns

Success: 0

Failure: Error code ([Status and Error Codes](#))

Read tag data

```
int tagdata_read (reader_handle_t h, tagop_args_t *a);
```

Reads a tag and places the data into the tag operations data structure pointed to by parameter *a*.

a.data_type: memory position, usually "normal" data

a.data_address: source address of data

a.data_length: number of bits

a.data_value: data, packed msb first 16 bits wide

Data is read and reported 1 byte at a time.

GEN2

a.data_value[0]: This is used to specify the memory bank of the data read.

To add a password for Gen 2 read commands, place the password starting at **data_value[1]**. Adjust the **data_length** to reflect the extra data.

Parameters

Parameter	Description
<i>h</i>	The reader handle
<i>a</i>	A tag operation's data structure with the following fields set: a.protocol : protocol of the tag a.antenna : antenna to use (1 based) a.id_length : length of the tag ID (in bits) a.id_value : ID of the tag a.data_address : address to read in tag memory a.data_length :length of the data in data_value a.data_type :specifies that data_length is specified in blocks. Otherwise, length is specified in bits.

Returns

Success: 0

Failure: Error code ([Status and Error Codes](#))

Write data to a tag

```
int tagdata_write (reader_handle_t h, tagop_args_t *a);
```

Writes data to a tag.

a.data_value is packed most-significant-byte first, 16 bits wide

Example of writing data to a tag:

```
char_data=5;  
a.data_value= char_data << 8;
```

Gen2

a.data_type: This field corresponds to the `memory_bank` value defined in the Gen2 specification [here](#).

To add a password to Gen2 write commands, place the password after the data to be written in the “`data_value`” array. To signal the presence of the password, set the flag bits of **a.data_type** to 4.

Adjust **a.data_length** to reflect the additional data.

Parameters

Parameter	Description
<i>h</i>	The reader handle
<i>a</i>	A tag operation’s data structure with the following fields set: a.protocol : protocol of the tag a.antenna : antenna to use (1 based) a.id_length : length of the tag ID (in bits) a.id_value : ID of the tag a.data_address : target address in memory, required a.data_length : length of the data in bits a.data_type : memory partition (protocol dependent) a.data_value : the data

Returns

Success: 0

Failure: Error code ([Status and Error Codes](#))

Lock tag data

```
int tagdata_lock(reader_handle_t h, tagop_args_t *a);
```

Locks the data on a tag.

GEN2 Only

a.data_type: This is used to fill in the “`mask`” value as specified in the Gen2 specification [here](#). Memory banks 3 are considered `tag_data` that can be locked.

Note

Memory bank 0 is used for the passwords.

a.data_value[0]: This is used to fill in the “action” value as specified in the Gen2 specification [here](#).

a.data_value[1]: The high 16 bits of the password

a.data_value[2]: The low 16 bits of the password

Parameters

Parameter	Description
<i>h</i>	The reader handle
<i>a</i>	A tag operation’s data structure with the following fields set: a.protocol : protocol of the tag a.antenna : antenna to use (1 based) a.id_length : length of the tag ID (in bits) a.id_value : ID of the tag a.data_address : address of block to lock a.data_length :length of data in data_value a.data_type :page mask for Data section to lock (GEN2 only) a.data_value : action and password (GEN2 only)

Returns

Success: 0

Failure: Error code (negative value)

GPIO Commands

The Reader API provides control of and response to the GPIO (General-Purpose Input/Output) lines on the fixed readers.

The fixed reader GPIO lines are referred to symbolically as GPIO_0, GPIO_1, GPIO_2, GPIO_3 and GPIO_4. Some lines are inputs and some are outputs, as described elsewhere in the fixed reader documentation.

The following table lists the GPIO commands.

GPIO Commands

Command	Description
int GPIO_get	Returns the state of GPIO lines
int GPIO_set	Sets the state of the GPIO output lines
int GPIO_set_reader_event	Establishes an association between GPIO lines and a reader event

GPIO Data Structure

The GPIO lines are enumerated in the following set of definitions:

```
#define GPIO_0 (0x04)
#define GPIO_1 (0x08)
#define GPIO_2 (0x10)
#define GPIO_3 (0x02)
#define GPIO_4 (0x20)
#define GPIO_5 (0x40)
#define GPIO_6 (0x80)
#define GPIO_7 (0x100)
```

Mercury4/Mercury5 GPIO

On the Mercury4 and Mercury5 the GPIO signals are sent/received through the serial port. The GPIO enumeration to serial port pin mapping is as follows:

- ◆ GPIO_0 is output on pin 6 on the serial port
- ◆ GPIO_1 is output on pin 1 on the serial port
- ◆ GPIO_2 is output on pin 9 on the serial port

- ◆ GPIO_3 is input on pin 4 on the serial port
- ◆ GPIO_4 is input on pin 7 on the serial port

Note

GPIO_4 is actually the serial ports RTS line. Checking the status of GPIO_4 is more time costly than checking GPIO_3.

Astra GPIO

On Astra readers GPIO signals are sent/received through a 12-pin screw terminal connector. The GPIO enumeration to pin mapping is as follows:

- ◆ GPIO_0 is output on pin 6
- ◆ GPIO_1 is output on pin 7
- ◆ GPIO_2 is output on pin 8
- ◆ GPIO_5 is output on pin 9
- ◆ GPIO_3 is input on pin 2
- ◆ GPIO_4 is input on pin 3
- ◆ GPIO_6 is input on pin 4
- ◆ GPIO_7 is input on pin 5

Functions

Get the State of GPIO Lines

```
int GPIO_get (reader_handle_t h, int gpio);
```

Returns the state of the given GPIO lines.

The GPIO lines above are enumerated in [GPIO Commands](#).

The *gpio* parameter is actually a bit-mask, so it is possible to check multiple values at once by supplying the result of a bitwise OR operation.

Syntax

In the following example, the status of GPIO inputs 0 and 3 are checked with one call to **GPIO_get()**.

Example:

```
int gpio_rv;
gpio_rv = GPIO_get(h, GPIO_0|GPIO_3);
if (gpio_rv & GPIO_0)
{ printf("GPIO_0 is high\n");}
if (gpio_rv & GPIO_3)
{ printf("GPIO_3 is high\n");}
```

Parameters

Parameter	Description
<i>h</i>	The reader handle
<i>gpio</i>	The GPIO lines to be queried

Returns

Success: 0 or 1 bit in the bit of the requested line, depending on the GPIO line state

Failure: -1

Version: 0.0.111

Set the State of the GPIO Output Lines

```
int GPIO_set (reader_handle_th, int gpio, int value);
```

Sets the state of the specified GPIO output lines.

The *gpio* lines are enumerated in [GPIO Commands](#).

gpio is actually a mask, so it is possible to set multiple values at once by supplying the result of an OR operation as the *gpio* value.

value is actually a mask, so it is possible to set different bit values at once by supplying the result of an OR operation as *value*.

This example sets the values of GPIO_0, GPIO_1 and GPIO_2. GPIO_0 and GPIO_2 are set high, while GPIO_1 is set low.

Example:

```
GPIO_set(h, GPIO_0|GPIO_1|GPIO_2, GPIO_0|GPIO_2);
```

Parameters

Parameter	Description
<i>h</i>	The reader handle
<i>gpio</i>	The GPIO lines to be queried
<i>value</i>	Value to which to set the bits of the GPIO lines

Returns

Success: 0

Failure: -1

Version: 0.0.111

Establish an Association with GPIO Lines and a Reader Event

```
int GPIO_set_reader_event(reader_handle_t h, int gpio, int value, enum
    gpio_reader_event event);
```

Establishes an association between a given GPIO lines (input or output) and a tag reader system event.

The GPIO lines are enumerated in [GPIO Commands](#).

```
enum gpio_reader_event {
    READER_EVENT_IMMEDIATE          /* Special Case: Set GPIO now */
    READER_EVENT_SEARCH_START       /* Reader in active search mode */
    READER_EVENT_SEARCH_STOP        /* Reader not currently searching */
    READER_EVENT_READY              /* TMD is running */
    READER_EVENT_INPUT_SEARCH_START /* Reader must wait for input */
    READER_EVENT_INPUT_SEARCH_STOP  /* Reader must wait for input */
    NUM_READER_EVENTS               /* Length of reader_event list */
};
```

Syntax

The following example permits a search to start when GPIO_0 and GPIO_2 are high and GPIO_1 is low.

Example:

```
GPIO_set_reader_event(h, GPIO_0|GPIO_1|GPIO_2, GPIO_0|GPIO_2,  
    READER_EVENT_INPUT_SEARCH_START);
```

Parameters

Parameter	Description
<i>h</i>	The reader handle
<i>gpio</i>	The GPIO line of interest
<i>value</i>	Value of the GPIO line
<i>event</i>	Event with which to associate the GPIO line

Remarks

gpio is actually a mask, so it is possible to set multiple values at once by supplying the result of an OR operation as the *gpio* value.

value is actually a mask, so it is possible to set different bit values at once by supplying the result of an OR operation as *value*.

event is the event to associate with the set of GPIO lines specified by *gpio*. These events can be those that are monitored (e.g. READER_EVENT_SEARCH_START) or generated due to a change in a GPIO input's state (for example READER_EVENT_INPUT_SEARCH_START).

Returns

Success: 0

Failure: -1

Version: 0.0.111

Reader Configuration

The following generic functions are used to inquire about and manipulate the reader's capabilities.

The following table lists the reader configuration commands.

Reader Configuration Commands

Command	Description
get_supported_protocols	Gets protocol list supported by reader
get_connected_antennas	Retrieves list of connected antennas
get_firmware_version	Returns a string that describes the MercuryOS firmware version
get_os_version	Gets the current operating system version
get_max_reader_power	Gets the maximum allowed TX power setting for this reader.
set_tx_power	Sets the transmitter power in dBm for current session
set_slot_tx_power	Sets the transmitter power (deprecated)
set_antenna_tx_power	Sets the transmitter power in dBm for the current reader
get_antenna_tx_power	Returns the current transmitter power for the antenna
config_set_EPC1_96_bit_support	Configures the reader to read or not read 96-bit EPC1 tags
config_get_EPC1_96_bit_support	Checks that the reader is configured to read 96-bit EPC1 tags
config_set_EPC1_id_length	Configures the reader to read only 64-bit EPC1 tags
TagID_Length_config_get_epc1_id_length	Returns an enum that describes the length of EPC1 tags the reader is configured to read
config_set_EPC0_search_depth	Sets the number of tags that the reader decodes in each search
config_get_EPC0_search_depth	Gets the number of tags that the reader decodes in each search
config_get_EPC0_degghost	Retrieves the status of EPC0 de-ghosting
config_set_EPC0_degghost	Sets the status of EPC0 de-ghosting
config_get_EPC0_degghost_len	Retrieves the current EPC0 de-ghosting length

Command	Description
<code>config_set_EPC0_deghost_len</code>	Sets the value of EPC0 de-ghosting length
<code>config_get_EPC1_deghost</code>	Retrieves the status of EPC1 de-ghosting
<code>config_set_EPC1_deghost</code>	Sets the status of EPC1 de-ghosting
<code>config_get_tagop_retry_timeout_ms</code>	Returns the time set in milliseconds to time out tag commands
<code>config_set_tagop_retry_timeout_ms</code>	Sets the time set in milliseconds to time out tag commands
<code>reader_params_set</code>	Sets the key to value
<code>reader_params_get_r</code>	Gets the key's value
<code>reader_params_list_r</code>	Gets the list of key parameters that are valid for use with reader configuration
<code>reader_fw_update</code>	Issues a request for the reader to initiate a firmware update
<code>reader_fw_update_complete</code>	Fills the user-allocated structure with data from the firmware update
<code>reboot_reader</code>	Remotely reboots the reader to activate new firmware

Reader Configuration Data Structure

They use the following type and constant definitions:

```
typedef struct {
    int num_protocols;
    int protocol_id[MAX_PROTOCOL_ARRAY_LEN];
} protocol_array_t;

typedef struct {
    int num_antennas;
    int antenna_id[MAX_ANTENNA_ARRAY_LEN];
} antenna_array_t;

typedef struct {
    int retval;
    char *package;
    char *version;
    char *maintainer;
    unsigned int flags;
} firmware_desc_t;
```

```
// The following are firmware update flags
    #define FIRMWARE_DOWNGRADE_DISALLOWED (0)
// disallow earlier versions
    #define FIRMWARE_DOWNGRADE_ALLOWED (1<<0)
// allow earlier firmware versions
    #define FIRMWARE_CLEAN_INSTALL (1<<1)
// wipe contents before install

// (does not include tm.conf)
    #define FIRMWARE_RESTORE_SETTINGS (1<<2)
// restore factory settings

// (replace tm.conf)
    #define FIRMWARE_OUTPUT_MSG (1<<3)
// display dots (.) during update
    #define FIRMWARE_AUTOMATIC_HOSTNAME (1<<4)
// get firmware with specific MAC

// address extension
    #define FIRMWARE_REBOOT (1<<5)
// automatically reboot if needed
```

Functions

Get Protocol List Supported by Reader

```
int get_supported_protocols(reader_handle_t h, protocol_array_t *p);
```

Retrieves a list of the protocols supported by the reader.

Parameters

Parameter	Description
<i>h</i>	The reader handle
<i>p</i>	Pointer to a user allocated list of supported protocols

Returns

Success: number of protocols ≥ 0

Failure: -1

Retrieve Antenna List Connected to Reader

```
int get_connected_antennas(reader_handle_t h, antenna_array_t *p);
```

Retrieves a list of the antennas currently connected to the reader.

Parameters

Parameter	Description
<i>h</i>	The reader handle
<i>p</i>	Pointer to a user allocated list of connected antennas

Returns

Success: number of antennas ≥ 0

Failure: -1

Get Current Firmware Version

```
char *get_firmware_version(reader_handle_t h);
```

Returns a string describing the current MercuryOS RFID firmware version.

Memory is allocated internally for the returned string. The API manages this memory and the user should not free the returned string. If users wish for this data to persist, they should make a copy of the return value.

Parameters

Parameter	Description
<i>h</i>	The reader handle

Returns

Success: a string describing the current firmware version

Failure: NULL

Get Current Operating System Version

```
char *get_os_version(reader_handle_t h);
```

Returns a string describing the current MercuryOS operating system version.

Memory is allocated internally for the returned string. The API manages this memory and the user should not free the returned string. If users wish for this data to persist, they should make a copy of the return value.

Parameters

Parameter	Description
<i>h</i>	The reader handle

Returns

Success: a string describing the current operating system version

Failure: NULL

Get Maximum TX Power

```
void get_max_reader_power(reader_handle_t h, short *maxReaderPower);
```

Returns the current transmitter power in dBm on the specified antenna for the specified reader.

Parameters

Parameter	Description
<i>h</i>	The reader handle
<i>maxReaderPower</i>	The maximum allowed TX power setting on this reader

Returns

Success: Number describing the current power setting (in dBm).

Failure: -1.0

Set Transmitter Power

```
double set_tx_power(reader_handle_t h, float power);
```

Sets the transmitter power in dBm for the current reader session and returns the power level actually set. Units of dBm are decibels relative to 1 mW (milliwatt), therefore 0 dBm \equiv 1 mW, 3 dBm \equiv 2 mW, 30 dBm \equiv 1 W.

Note

The standard 25 foot cable distributed with a Mercury4 has a loss factor of approx. 0.1 dB/foot, for a total of 2.5 dB loss, therefore 32.5 dBm power output at the Mercury4 or Mercury5 antenna port will correspond to 30.0 dBm at the end of the cable.

Parameters

Parameter	Description
<i>h</i>	The reader handle
<i>power</i>	Setting for the reader in dBm. The Mercury4 or Mercury5 reader accepts a power setting in the range $0 \leq 32.5$ dBm. Astra accepts power between 5.0 dBm and 30.0 dBm

Returns

Success: Number describing the current power setting (in dBm).

Failure: -1.0

Set Transmitter Power per Slot [Deprecated]

```
double set_slot_tx_power(reader_handle_t h, float power, short slot);
```

This has been deprecated, in favor of [double set_antenna_tx_power\(reader_handle_t h, float power, short antenna\);](#).

Sets the transmitter power in dBm for the current reader session on the specified slot and returns the power actually set. Units of dBm are decibels relative to 1 mW (milliwatt), therefore 0 dBm \equiv 1 mW, 3 dBm \equiv 2 mW, 30 dBm \equiv 1 W.

Note

The standard 25 foot cable distributed with an Mercury4 has a loss factor of approx. 0.1 dB/foot, for a total of 2.5 dB loss, therefore 32.5 dBm power output at the Mercury4 or Mercury5 antenna port will correspond to 30.0 dBm at the end of the cable.

Parameters

Parameter	Description
<i>h</i>	The reader handle
<i>power</i>	Setting for the reader in dBm. The Mercury4 or Mercury5 reader accepts a power setting in the range $0 \leq 32.5$ dBm
<i>slot</i>	The slot this power setting refers to (0 based)

Returns

Success: Number describing the current power setting (in dBm).

Failure: -1.0

Set the TX Power on a Specified Antenna

```
double set_antenna_tx_power(reader_handle_t h, float power, short antenna);
```

Sets the transmitter power in dBm for the current reader session on the specified antenna and returns the power actually set. Units of dBm are decibels relative to 1 mW (milliwatt), therefore 0 dBm \equiv 1 mW, 3 dBm \equiv 2 mW, 30 dBm \equiv 1 W.

Note

The standard 25 foot cable distributed with a Mercury4 has a loss factor of approx. 0.1 dB/foot, for a total of 2.5 dB loss, therefore 32.5 dBm power

output at the Mercury4 or Mercury5 antenna port will correspond to 30.0 dBm at the end of the cable.

Parameters

Parameter	Description
<i>h</i>	The reader handle
<i>power</i>	Setting for the reader in dBm. The Mercury4 or Mercury5 reader accepts a power setting in the range $0 \leq 32.5$ dBm. Astra accepts power between 5.0 dBm and 30.0 dBm
<i>antenna</i>	The antenna this power setting is referring to (0 based)

Returns

Success: number describing the current power setting (in dBm).

Failure: -1.0

Note

On Astra readers the power cannot be set uniquely per antenna. Calling this function sets the power on both antennas (if the 2nd, external antenna is used).

Get the Current Slot TX Power [Deprecated]

```
double get_slot_tx_power(reader_handle_t h,short slot);
```

Returns the current transmitter power in dBm on the specified slot for the specified reader.

Parameters

Parameter	Description
<i>h</i>	The reader handle
<i>slot</i>	The slot this power setting is referring to (0 based)

Returns

Success: Number describing the current power setting (in dBm).

Failure: -1.0

Get Current Antenna TX Power

```
double get_antenna_tx_power(reader_handle_t h, short antenna);
```

Returns the current transmitter power in dBm on the specified antenna for the specified reader.

Parameters

Parameter	Description
<i>h</i>	The reader handle
<i>antenna</i>	The antenna this power setting is referring to (0 based)

Returns

Success: Number describing the current power setting (in dBm).

Failure: -1.0

Configure Reader to Enable or Disable Reading EPC1 Tags

```
int config_set_EPC1_96_bit_support(reader_handle_t h, int on);
```

Configures the fixed readers to read 96-bit EPC1 tags (on == 1) or not to read 96-bit EPC1 tags (on == 0).

Note

Enabling 96-bit EPC1 support incurs a performance penalty because it must decode more bits to determine whether a tag has successfully been read.

Parameters

Parameter	Description
<i>h</i>	The reader handle
<i>on</i>	Enable or disable EPC1 support for 96 bit (Lepton) tags

Returns

Success: 0

Failure: < 0

Check Reader Configuration for Reading EPC1 Tags

```
int config_get_EPC1_96_bit_support(reader_handle_t h);
```

Checks whether the fixed readers are configured to read 96-bit EPC1 tags.

Note

Enabling 96-bit EPC1 support incurs a performance penalty because it must decode more bits to determine whether a tag has successfully been read.

Parameters

Parameter	Description
<i>h</i>	The reader handle

Returns

Success: 1 if enabled

0 if disabled

Failure: -1

Configure the Reader to Read only 64-Bit EPC1 Tags

```
int config_set_epc1_id_length(reader_handle_t h, enum TagID_Length  
    len_enum);
```

Configures the fixed readers to read only 64-bit EPC1 tags (`len_enum == TagIdLength_64`) or to read only 96-bit EPC1 tags (`len_enum == TagIdLength_96Only`) or to read 96-bit and 64-bit EPC1 tags (`len_enum == TagIdLength_96`)

Note

Enabling 96-bit EPC1 support incurs a performance penalty because more bits must be decoded to determine whether a tag has successfully been read.

Syntax

```
typedef enum TagID_Length {  
    TagIdLength_64,  
    TagIdLength_96Only,  
    TagIdLength_96,  
    TagIdLength_LastValue  
} TagID_Length;
```

Parameters

Parameter	Description
<i>h</i>	The reader handle
<i>len_enum</i>	An enum describes the length of EPC1 tags

Returns

Success: 0

Failure: -1

Return the length of EPC1 tag IDs the reader is configured to read

```
enum TagID_Length config_get_epc1_id_length(reader_handle_t h);
```

Returns an enum that describes the length of EPC1 tags the reader is configured to read.

Parameters

Parameter	Description
<i>h</i>	The reader handle

Returns

The enum describing the length of EPC1 tags the reader is configured to read.

Set the Number of EPC0 Tags the Reader Decodes

```
int config_set_EPC0_search_depth(reader_handle_t h, int depth);
```

Sets the number of EPC0 tags that the reader will attempt to decode in each EPC0 search iteration.

Note

A larger search depth results in longer EPC0 search iterations.

Parameters

Parameter	Description
<i>h</i>	The reader handle
<i>slot</i>	Sets the depth of EPC0 anticollision search

Returns

Success: 0

Failure: <0

Get the Number of EPC0 Tags the Reader Decodes

```
int config_get_EPC0_search_depth (reader_handle_t h);
```

Gets the number of EPC0 tags that the reader will attempt to decode in each EPC0 search iteration.

Note

A larger search depth results in longer EPC0 search iterations.

Parameters

Parameter	Description
<i>h</i>	The reader handle

Returns

Success: search depth being used

Failure: <0

Get the Status of EPC0 De-ghosting

```
int config_get_EPC0_degghost(reader_handle_t h);
```

Retrieves the status of EPC0 de-ghosting.

Parameters

Parameter	Description
<i>h</i>	The reader handle

Returns

Success: 1 if deghosting is enabled
0 if deghosting is disabled

Failure: < 0

Set the Status of EPC0 De-Ghosting

```
int config_set_EPC0_degghost(reader_handle_t h, int on);
```

Sets the status of EPC0 de-ghosting

Parameters

Parameter	Description
<i>h</i>	The reader handle
<i>on</i>	Enable or disable EPC0 de-ghosting

Returns

Success: 0

Failure: <0

Get the EPC0 De-Ghosting Length

```
int config_get_EPC0_deghost_len(reader_handle_t h);
```

Retrieves the current EPC0 de-ghosting length.

Parameters

Parameter	Description
<i>h</i>	The reader handle

Returns

Success: length

Failure: <0

Set the Value of EPC0 De-Ghosting Length

```
int config_set_EPC0_deghost_len(reader_handle_t h, int len);
```

Sets the value of EPC0 de-ghosting length.

Parameters

Parameter	Description
<i>h</i>	The reader handle
<i>len</i>	EPC0 de-ghosting length

Returns

Success: 0

Failure: <0

Get the Status of EPC1 De-Ghosting

```
int config_get_EPC1_deghost(reader_handle_t h);
```

Retrieves the status of EPC1 de-ghosting.

Parameters

Parameter	Description
<i>h</i>	The reader handle

Returns

Success: 1 if de-ghosting is enabled;
0 if de-ghosting is disabled;

Failure: <0

Set the Status of EPC1 De-Ghosting

```
int config_set_EPC1_deghost(reader_handle_t h, int on);
```

Sets the status of EPC1 de-ghosting.

Parameters

Parameter	Description
<i>h</i>	The reader handle
<i>on</i>	Enable or disable EPC1 de-ghosting

Returns

Success: 0

Failure: <0

Get Time Out for Tag Commands

```
int config_get_tagop_retry_timeout_ms(reader_handle_t h);
```

Returns the number of milliseconds used when timing out a **tagid_write**, **tagid_lock**, **tag_kill** and **tag_passwd**.

Parameters

Parameter	Description
<i>h</i>	The reader handle

Returns

Success: milliseconds used for timing out

Failure: <0

Set the Time Out for Tag Commands

```
int config_set_tagop_retry_timeout_ms(reader_handle_t h, unsigned short  
    timeout_ms);
```

Sets the number of milliseconds used when timing out a **tagid_write**, **tagid_lock**, **tag_kill**, and **tag_passwd**.

Parameters

Parameter	Description
<i>h</i>	The reader handle
<i>timeout_ms</i>	Milliseconds used when timing out a tag operation

Returns

Success: 0

Failure: <0

Set Reader Parameters

```
int reader_params_set(reader_handle_t h, char* key, char* value);
```

Sets the key to value. For a list of reader parameters and their description see the MercuryOS Advanced User Guide available on <http://www.thingmagic.com/manuals-firmware>.

Parameters

Parameter	Description
<i>h</i>	The reader handle
<i>key</i>	Parameter being set
<i>value</i>	Value that “key” is being set to

Returns

Success: 0

Failure: <0

Get Reader Parameter Values [Deprecated]

```
Deprecated: char* reader_params_get(reader_handle_t h, char* key);
```

Gets the key value. For a list of reader parameters and their description see the MercuryOS Advanced User Guide available on <http://www.thingmagic.com/manuals-firmware>. This version of `reader_params_get` is not thread safe.

Parameters

Parameter	Description
<i>h</i>	The reader handle
<i>key</i>	Parameter being set

Returns

Success: pointer to string representing the value of key

Failure: NULL

Get Reader Parameter Values - Thread-safe

```
char* reader_params_get_r(reader_handle_t h, char* key);
```

Gets the key value. For a list of reader parameters and their description see the MercuryOS Advanced User Guide available on <http://www.thingmagic.com/manuals-firmware>.

Parameters

Parameter	Description
<i>h</i>	The reader handle
<i>key</i>	Parameter being set

Returns

Success: pointer to string representing the value of key

Note: You must free the pointer to the returned memory address when finished with it.

Failure: NULL

Get the List of Valid Parameters [Deprecated]

```
Deprecated: char** reader_params_list(reader_handle_t h, int* listLength);
```

Gets the list of key parameters valid for use with **reader_params_get** and **reader_config_set**. This version of `reader_params_list` is not thread safe.

Parameters

Parameter	Description
<i>h</i>	The reader handle
<i>listLength</i>	Length of the list being returned

Returns

Success: pointer to list of parameter names (an array of string pointers)

Failure: NULL

Get the List of Valid Parameters - Thread-safe

```
char** reader_params_list_r (reader_handle_t h, int* listLength);
```

Gets the list of key parameters valid for use with **reader_params_get_r** and **reader_config_set**. For a list of reader parameters and their description see the MercuryOS Advanced User Guide available on <http://www.thingmagic.com/manuals-firmware>.

Parameters

Parameter	Description
<i>h</i>	The reader handle
<i>listLength</i>	Length of the list being returned

Returns

Success: pointer to list of parameter names (an array of string pointers)

Note: You must free the pointer to the returned memory address when finished with it.

Failure: NULL

Perform a Firmware Update

```
int reader_fw_update (reader_handle_t h, char *fw_uri, int flags);
```

Issues a request for the reader to initiate a firmware update from the given URI.

The supported URI types are “http” and “tftp”.

The possible flag values are enumerated above ([Reader Configuration Data Structure](#)).

Parameters

Parameter	Description
<i>h</i>	The reader handle
<i>fw_uri</i>	A URI describing the location of the firmware update
<i>flags</i>	Bitmask of flags to control the firmware update operation

Returns

Success: 0

Failure: -1

Get Firmware Update Status

```
int reader_fw_update_complete(reader_handle_t h, firmware_desc_t *fp);
```

The user-allocated structure is filled in with data that describes the requested firmware update in progress. When the search completes, the flag data passed into the firmware update request is returned in *fp*.

Parameters

Parameter	Description
<i>h</i>	The reader handle
<i>fp</i>	A user-allocated structure that describes the firmware update

Returns

Success: ERROR_SUCCESS indicates successful completion

ERROR_FIRMWARE_UPDATE_INCOMPLETE indicates the update is still in progress

Failure: -1

Reboot the Reader after Firmware Update Remotely

```
void reboot_reader(reader_handle_t h, unsigned long magic, unsigned int  
    safe_mode_flag);
```

With this API call, you can reboot the reader remotely. It is mainly intended to be used after a firmware update, so that the new firmware can be activated.

Parameters

Parameter	Description
<i>h</i>	The reader handle
<i>magic</i>	The magic number must be equal to 536873248
<i>safe_mode_flag</i>	1 reboot to safe mode; 0 reboot normally

Returns

None.

Status and Error Codes

Operation Status Codes

Code	Description
OPERATION_WAITING_TO_RUN	Operation has been queued
OPERATION_IN_PROGRESS	Operation is in progress
OPERATION_TIMED_OUT	Operation completed due to timeout
OPERATION_DONE	Operation completed due to search completion
OPERATION_ERROR	Error checking the operation status

Error Codes

Code	Description
ERROR_SUCCESS	The operation completed successfully
ERROR_INVALID_ARGUMENTS	API function called with invalid arguments
ERROR_INVALID_ANTENNA	Antenna port does not exist in current configuration
ERROR_UNCONNECTED_ANTENNA	Antenna port exists, but no antenna connected
ERROR_VERIFY_FAILURE	Operation completed but result not as expected

Global Error Code

```
int reader_errno;
```

reader_errno is a global error code summarizing the last error, and takes on one of the error code values listed in [Status and Error Codes](#).

It is not cleared by a successful operation.

Functions

Print a Description of the Last Error

`void reader_perror (char *msg);`Version 0.0.111

Prints a description of the last error to the standard error output stream.

If *msg* is non-NULL, it is printed as a prefix to the error description.

Syntax

The following example prints out a description of the error:

```
/* use reader_perror() print out the operation status */  
reader_perror ("operation_status()");
```

Parameters

Parameter	Type	Description
<i>msg</i>	string pointer	A string that is prepended to the error description

Returns

None.

SNMP

The functions defined in this section are used to access reader statistics as defined by the EPCGlobal Reader MIB.

In addition to getting reader statistics the API can also be used to configure the SNMP settings on the reader using the [Set Reader Parameters](#) function to set SNMP parameters and the [Get Reader Parameter Values - Thread-safe](#) function to get SNMP parameter values as defined in the MercuryOS Advanced User Guide.

The following table lists the reader configuration commands.

SNMP Commands

Command	Description
Get Antenna Statistics	Get antenna statistics
Get Reader Statistics	Get reader statistics
Reset Statistics	Reset all statistics

Statistics Data Structures

The following data structures must be allocated prior to calling the corresponding statistics retrieval function. The value of each statistic will be filled in the by the system before the corresponding call returns:

Antenna Statistics

```
struct antenna_stats {
    int tags_identified;
    int tags_not_identified;
    int memory_read_operations;
    int memory_read_failures;
    int write_operations;
    int write_failures;
    int kill_operations;
    int kill_failures;
    int erase_operations;
    int erase_failures;
    int lock_operations;
    int lock_failures;
    int noise_level;
    int time_energized
```

```
    } antenna_stats_t;
```

Reader Statistics

```
struct source_stats {  
    int read_cycles_per_trigger;  
    int read_duty_cycle;  
    int read_timeout;  
    int glimpsed_timeout;  
    int observed_timeout;  
    int observed_threshold;  
    int lost_timeout;  
    int unknown_to_glimpsed;  
    int glimpsed_to_unknown;  
    int glimpsed_to_observed;  
    int observed_to_lost;  
    int lost_to_glimpsed;  
    int lost_to_unknown;  
} source_stats_t;
```

Functions

Get Antenna Statistics

```
int get_antenna_stats(reader_handle_t h, int antenna_id, antenna_stats_t *stats);
```

Retrieves the statistics for the specified antenna. Copies the statistics into the caller-allocated [Antenna Statistics](#) data structure.

Parameters

Parameter	Description
<i>h</i>	The reader handle
<i>antenna_id</i>	Antenna number to retrieve statistics for. (zero-based)
<i>*stats</i>	pointer to an antenna_stats_t structure

Returns

Success: = 0

Failure: != 0

Get Reader Statistics

```
int get_reader_stats(reader_handle_t h, source_stats_t *stats);
```

Retrieves the statistics for the reader. Copies the statistics into the caller-allocated [Reader Statistics](#) data structure.

Parameters

Parameter	Description
<i>h</i>	The reader handle
<i>*stats</i>	pointer to an <code>source_stats_t</code> structure

Returns

Success: = 0

Failure: != 0

Reset Statistics

```
void reset_epcg_stats(reader_handle_t h);
```

Resets all statistics.

Parameters

Parameter	Description
<i>h</i>	The reader handle

Returns

Success: number of antennas ≥ 0

Failure: -1

Platform Specific Functionality

The Mercury[®]OS C API is intended to be a cross-platform API for use with Mercury4 and Mercury5 readers as well as Astra readers. However, due to hardware differences and different firmware release schedules, not all features are supported on all platforms. This sections describes any limitations or additional functionality available for each platform:

- ◆ [Astra Readers](#)
- ◆ [Mercury4 & Mercury5 Readers](#)

Astra Readers

Shared Memory Usage

When writing “On-Reader” applications the Astra readers does not take advantage of the “Fast APIs’ feature which results in some APIs using shared memory instead of RPC calls to improve performance. On Astra readers RPC calls are used for all APIs whether “on-reader” or remote.

Supported Protocols

Astra readers currently only support the Gen2 protocol.

SNMP

The SNMP Implementation on the M4/M5 readers is a Beta release. Some functionality may not work or may not work as documented. Please use with caution.

Mercury4 & Mercury5 Readers

SNMP

The SNMP Implementation on the M4/M5 readers is a Beta release. Some functionality may not work or may not work as documented. Please use with caution.

