RigiSystems
Connected Solutions Services

vTuner API
Reference Implementation

SRS

Testcases

System Requirement Specification

System Test Definitions

Version: 1.0 – 16

RigiSystems AG
Bundesstrasse 3
CH-6300 Zug
Switzerland
+41 41 712 10 44

Contents

1 Scope

1.1 Identification

Version	Name, Date	Comments
1.0 – 01	RSr, 2008-05-28	First draft, API suggestion
1.0 – 02	RSr, 2008-06-03	Added Datastructure. ResultSet modified completely
1.0 – 03	RSr, 2008-06-05	BackupURL added to datastructures, cache and bookmark specification added.
1.0 – 04	RSr, 2008-06-06	Added projects file structures
1.0 – 05	RSr, 2008-06-16	Adjusted requirements regarding the XML parser and the http client.
1.0 – 06	RSr, 2008-06-16	Added reference application
1.0 – 07	AY, 2008-07-11	Added bunch of test cases
1.0 – 08	AY, 2008-07-12	Minor modification of few test cases description
1.0 – 09	YK, 2008-07-21	Changed the output of mtrack command.
1.0 – 10	YK, 2008-07-22	Added vTuner_getLastItem() function
1.0 – 11	AY, 2008-07-22	Added search testcase
1.0 – 12	AY, 2008-07-22	Modified description of language support test according to realities of its operation.
1.0 – 13	AY, 2008-07-23	Modified test cases section.
1.0 – 14	AY, 2008-07-24	Modified some test cases. Added test directory structure.
1.0 – 15	AY, 2008-07-25	Added few test cases description.
1.0 – 16	RSr, 2008-11-12	Adjusted test case description to match with vTuner test directory

Table 1: Document History

1.2 System Overview

The target of the vTuner Reference API is to have a reference implementation, which is easily portable to any C / C++ application.

A reference application is built to show the usage of the API as an example implementation.

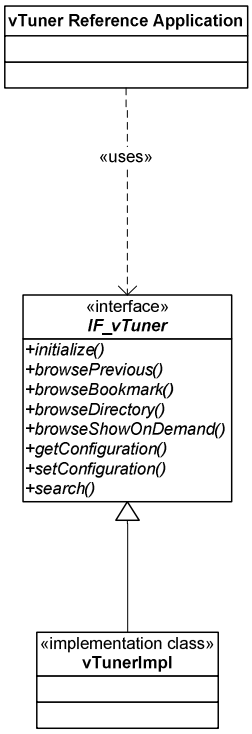


Figure 1: Overview

2 Referenced Documents

2.1 References

[1] vTuner API Specification:
\\rigi\pub\Customers\vTuner\documentation\vTuner API Generic Rev 5_7B.pdf

[2] “expat” XML parser
http://expat.sourceforge.net/

[3] “curl” HTTP client
http://curl.haxx.se/

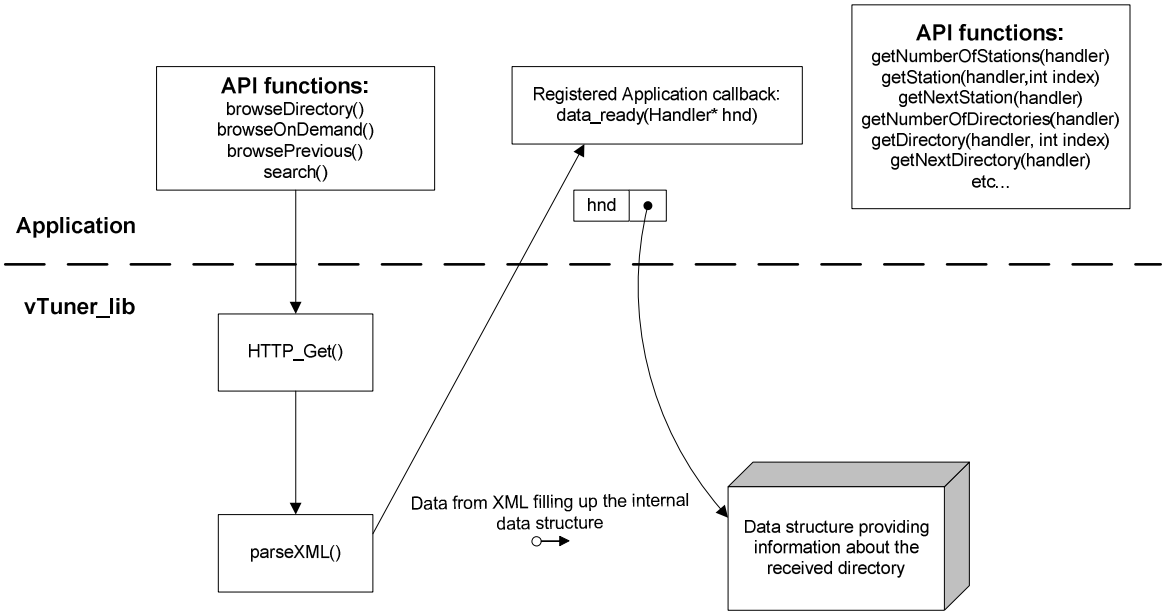
3 Requirements

3.1 Global Requirements

Implementation must be held in pure ANSI C, only predefined datatypes must be used (uint8, uint16, uint32, sint8, sint16, sint32), to maximize portability to different architectures.	R(1)	
Must be possible to configure the build regarding memory consumption. The result list can hold maximum or minimum needed information. That needs to be selectable through configuration	R(2)	
The module shall provide cache functionality. Need for a mechanism to avoid non cacheable pages!	R(3)	
Uses standard BSD functions for network access	R(4)	
For XML-Parsing “Expat” is shipped along with the vTuner library. This is free software under the MIT license. The user can register his own XML parser to keep the memory consumption low.	R(5)	
Dynamic memory can be used. A “vT_malloc()” shall be offered in the header file which can be implemented by the user.	R(6)	
As http client “CURL” is shipped along with the vTuner library. This is free software under the MIT license. The user can register his own httpClient so keep the memory consumption low.	R(7)	
If a proprietary OS (not windows or Linux based) is used, the user is self responsible to port CURL onto his system. CURL requires BSD socket functions to be ported.	R(8)	
Shell based test/reference application must be included in the shipment	R(9)	

4 API documentation

4.1 Overview



There are 4 API calls which result in a HTTP Get() request: browseDirectory, browseOnDemand, browsePrevious and search. The resulting XML is put into the XML parser, which is responsible to fill up the data into the internal datastructure. When the parsing is finished, a previously registered callbackfunction (data_ready) is called, passing a handler as argument. This handler must be passed as first argument to all the data access functions.

4.2 Initialization

Before the vTuner database can be browsed, the module needs to be initialized. This happens by calling the following API function:

```
uint8 vTuner_initialize(uint8* cPrimaryURL,
                        uint8* cBackupURL,
                        uint8* cMACAddress,
                        uint8* cBlowfishKey,
                        uint8* cBlowfishIV,
                        uint8* cLanguageCode,
                        uint8 (*result_ready)(struct directory_handle*));
```

This function takes the primary and backup URL which shall be used for browsing requests. The MAC address is taken as well as the Blowfish key credentials. As last parameter, a function pointer to the result_ready function is passed and registered in the vTuner module.

The function then requests a session token from vTunerServer, and creates the login credentials based on the retrieved session token and the blowfish parameters.

See chapter Error Codes for definition of the return values.

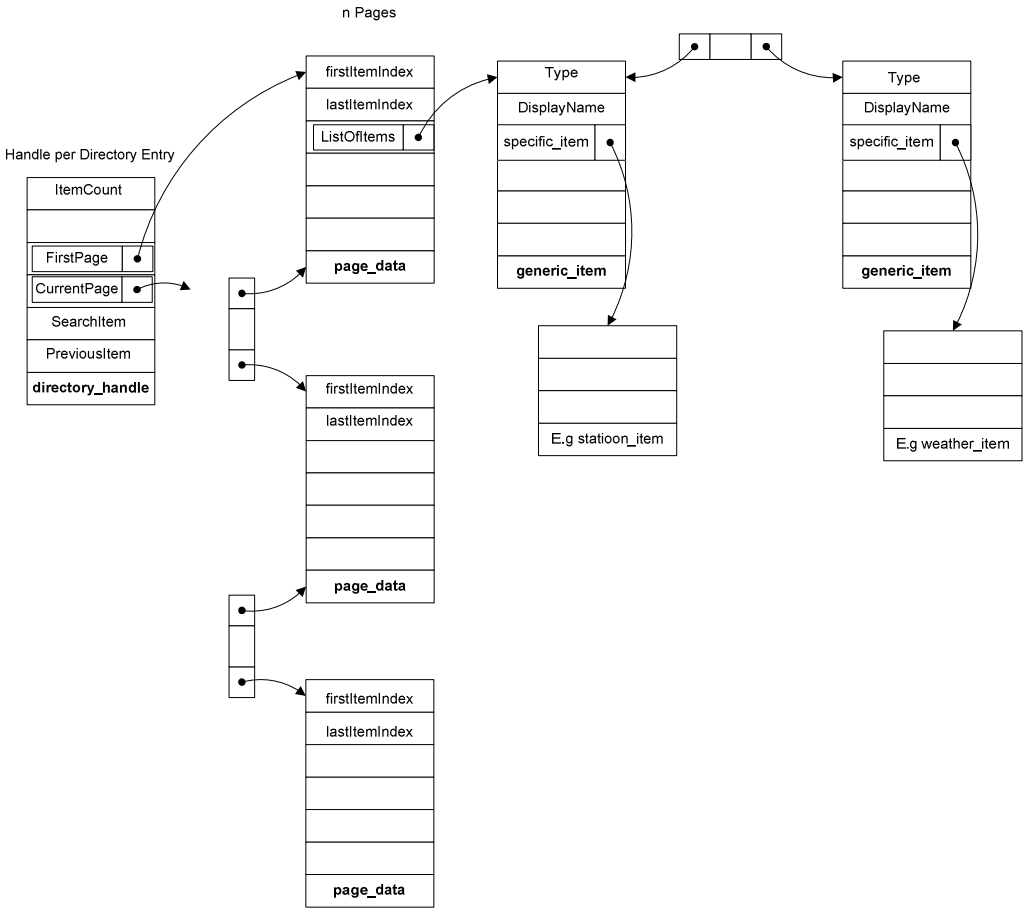
4.3 Internal Data Structure

The main task of the vTuner module is to fill up a data structure with the data, which is provided by the XML structure. API access functions are defined to browse these internal data structures.

For each directory a handler is assigned, which is equivalent with the pointer to the structure that describes that directory. From there, there is a pointer to a double linked list, which provides data access to the individual pages within a directory level. If paging isn't used, there is only one page.

The page structure contains a pointer to an arrays of pointers, each pointing to an item structure.

Internal Datastructure of vTuner Directory Data




```
struct bookmark_item
{
    uint8   *pBookmarkURL;
    uint8   *pBookmarkURLBackup;
};


struct generic_item
{
    struct generic_item *pNext;
    struct generic_item *pPrev;

    uint8   iItemType;
    uint8   cDisplayName[SHORT_TEXT_LENGTH];
    void    *pItem;
};
```

```
struct station_item
{
    uint8   cStationID[MIME_TYPE_LENGTH];
    uint8   cStationName[STATIONNAME_LENGTH];
    uint8   *pStationURL;
    uint8   *pStationDescription;
    uint8   cStationFormat[SHORT_TEXT_LENGTH];
    uint8   cStationLocation[SHORT_TEXT_LENGTH];
    uint8   cStationBandwidth[MIME_TYPE_LENGTH];
    uint8   cStationMimeType[MIME_TYPE_LENGTH];
    uint8   cStationLanguage[SHORT_TEXT_LENGTH];
    uint8   cStationCity[SHORT_TEXT_LENGTH];
    uint8   cStationState[SHORT_TEXT_LENGTH];
    uint8   cStationZIP[SHORT_TEXT_LENGTH];
    uint8   cStationCountry[SHORT_TEXT_LENGTH];
    uint8   cStationTimezone[MIME_TYPE_LENGTH];
    uint8   cStationBroadcast[SHORT_TEXT_LENGTH];
    uint8   cStationFrequency[SHORT_TEXT_LENGTH];
    uint8   cStationBand[SHORT_TEXT_LENGTH];
    uint8   cStationReliability[MIME_TYPE_LENGTH];
    uint8   cStationSoundQuality[MIME_TYPE_LENGTH];
    uint8   *pStationHomeURL;
    uint8   *pStationLogoURL;
    struct bookmark_item sStationBookmark;
};

struct directory_item
{
    uint8   cDirectoryTitle[DIRNAME_LENGTH] ;
    uint8   *pDirectoryURL;
    uint8   *pDirectoryURLBackup;
};

struct showondemand_item
{
    uint8   cShowOnDemandID[MIME_TYPE_LENGTH];
    uint8   cShowOnDemandName[SHOWONDEMAND_LENGTH];
    uint8   *pShowOnDemandURL;
    uint8   *pShowOnDemandURLBackup;
    struct  bookmark_item sShowOnDemandBookmark;
};

struct episode_item
{
    uint8   cEpisodeID[SHORT_TEXT_LENGTH];
    uint8   cEpisodeName[STATIONNAME_LENGTH];
    uint8   *pEpisodeDescription;
    uint8   *pEpisodeURL;
    uint8   *pShowDescription;
    uint8   cShowFormat[SHORT_TEXT_LENGTH];
    uint8   cShowMimetype[SHORT_TEXT_LENGTH];
    uint8   cShowLanguage[SHORT_TEXT_LENGTH];
    uint8   cShowCity[SHORT_TEXT_LENGTH];
    uint8   cShowState[SHORT_TEXT_LENGTH];
    uint8   cShowCountry[SHORT_TEXT_LENGTH];
    uint8   *pShowLogoURL;
    struct bookmark_item sShowBookmark;
};


struct display_item
{
    uint8   *pDisplayText;
};
```

```
struct search_item
{
    uint8   *pSearchURL;
    uint8   *pSearchURLBackup;
    uint8   cSearchCaption[SHORT_TEXT_LENGTH];
    uint8   cSearchTextbox[SHORT_TEXT_LENGTH];
    uint8   cSearchButton[SHORT_TEXT_LENGTH];
    uint8   cSearchCancel[SHORT_TEXT_LENGTH];
};

struct weather_item
{
    uint8   cWeatherID[SHORT_TEXT_LENGTH];
    uint8   cWeatherDayName[SHORT_TEXT_LENGTH];
    uint8   cWeatherForecast[SHORT_TEXT_LENGTH];
    uint8   cWeatherTempCurrent[SHORT_TEXT_LENGTH];
    uint8   cWeatherTempApparent[SHORT_TEXT_LENGTH];
    uint8   cWeatherTempFeel[SHORT_TEXT_LENGTH];
    uint8   cWeatherHumidity[SHORT_TEXT_LENGTH];
    uint8   cWeatherWindDirection[SHORT_TEXT_LENGTH];
    uint8   cWeatherWindSpeed[SHORT_TEXT_LENGTH];
    uint8   cWeatherPressure[SHORT_TEXT_LENGTH];
    uint8   cWeatherVisibility[SHORT_TEXT_LENGTH];
    uint8   *pWeatherLogo;
};

struct rss_item
{
    uint8   cRssID[SHORT_TEXT_LENGTH];
    uint8   cRssTitle[SHORT_TEXT_LENGTH];
    uint8   cRssDescription[LONG_TEXT_LENGTH];
    uint8   *pRssLink;
};

struct previous_item
{
    uint8   *pPreviousURL;
    uint8   *pPreviousURLBackup;
};

struct page_data
{
    struct page_data *pNext;
    struct page_data *pPrev;

    uint16 iNrOfItems;
    uint16 iFirstItemIndex;
    uint16 iLastItemIndex;
    struct generic_item *pItemList; /* pointer to the first item in the list */
    struct generic_item *pCurrentGenericItem; /* pointer to the current item in the list */
};

struct directory_handle
{
    struct page_data    *pFirstPage;
    struct page_data    *pCurrentBrowsingPage;
    int16  iNumberOfItems;

    struct previous_item    *pPreviousItem;   /* Browse previous without cache */
    struct search_item    *pSearchItem;

    struct directory_handle *pPrevious;   /* For cache purpose. Must be 0 if */
    /* “NoCache” flag is set!*/
};
```

```
struct cache_item
{
    struct cache_item       *pNext;
    struct directory_handle *pHandle;
    uint8                   *pURL;
};

struct cache_list
{
    struct cache_item  *pCacheList;
};
```

NOTE: This lists might become extremely large if the configuration is set to gather all possible information out of the XML! The user of the vTuner interface is self responsible to set the configuration in a way that the memory retrictions of the system are not exceeded.

4.4 Access Data

Along with the callback function `uint8 (*result_ready)(struct directory_handle*)`, the user gets a pointer to the directory handler.

A couple of functions is defined to access the data, and perform new requests to the server as required (accessing data outside the pages).

```
uint16                  vTuner_getNumberOfItems(struct directory_handle*);
struct generic_item*    vTuner_getFirstItem(struct directory_handle*);
struct generic_item*    vTuner_getLastItem(struct directory_handle*);
struct generic_item*    vTuner_getNextItem(struct directory_handle*);
struct generic_item*    vTuner_getPrevItem(struct directory_handle*);
struct generic_item*    vTuner_getItem(struct directory_handle*, uint16 iIndex);
struct search_item*     vTuner_getSearchItem(struct directory_handle*);
struct previous_item*   vTuner_getPreviousDirectoryItem(struct directory_handle*);
```

4.5 Browsing

To browse the vTuner database one of the following API functions can be invoked:

```
uint8 vTuner_browseDirectory(struct directory_item *pDirItem);
uint8 vTuner_browseShowOnDemand(struct showondemand_item *pShowItem);
uint8 vTuner_browsePrevious(struct previous_item *pPreviousItem);
uint8 vTuner_search(struct search_item *pSearchItem,
                            uint8 *cSearchString,
                            uint8 *iSearchType);
```

4.6 Caching

Caching happens automatically for pages within a directory. Furthermore, a cache list is provided that maps directory_handlers to URLs. Each browser request checks for presence in the cache list. If en entry

exist, the corresponding directory_handler is passed to the application through data_ready() callback. The application therefore immediately can access the data.

Special care must be given to the NoCache flag, which can be present in each of the XML files. If that flag is set to “YES”, the cache should be deleted when browsing to the previous directory.

The API function:

```
uint8 vTuner_emptyCache();
```

can be used for that.

4.7 Preload pages

If paging is used, the system shall preload the pages in the background, while the user browses the first page, in order to keep the browsing smooth. The first page can have a different number of items to be loaded than the following pages, so the initial feedback is very fast. The following pages can be loaded while the user is browsing the first few items.

4.8 Bookmarks

vTuner provides a way to store stations or episodes into a favourites directory. The API function for this is:

```
uint8 vTuner_bookmark(struct bookmark_item *pBookmark);
```

After that call, `data_ready()` will be called, which serves an answer to the request.

4.9 Configuration

The behaviour of the implementation can be configured to a certain extend. The following two functions are used to get and set the configuration parameters.

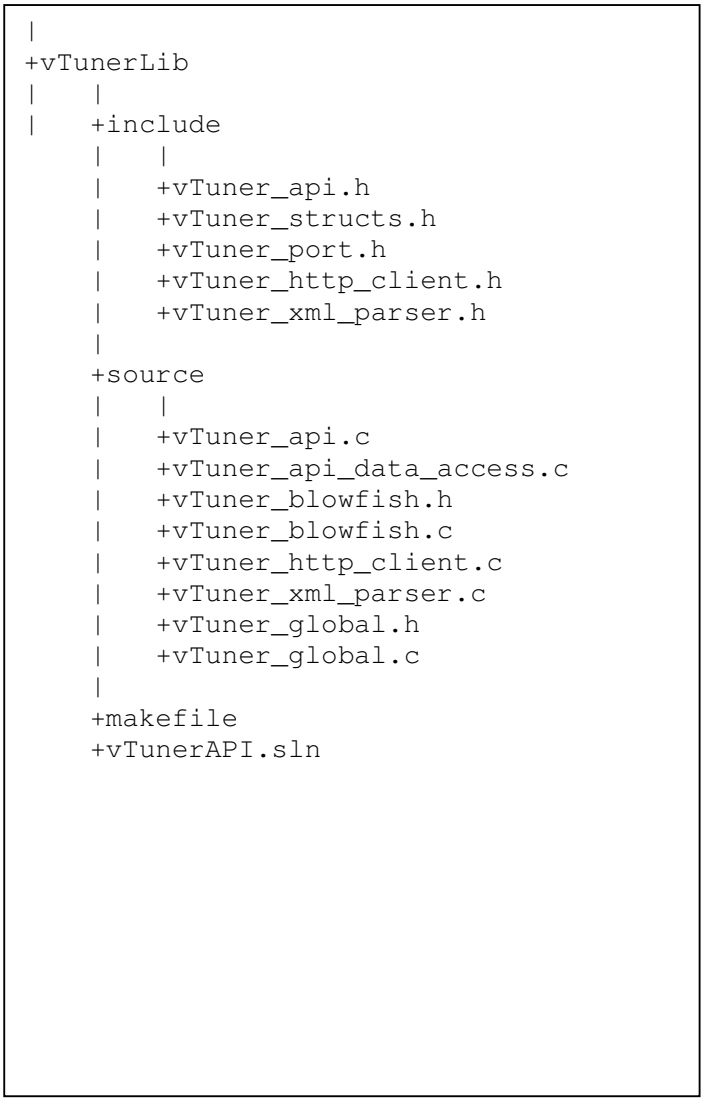
```
struct struct_configuration* vTuner_getConfiguration();
uint8    vTuner_setConfiguration(struct struct_configuration*);

struct struct_configuration
{
    uint16  iItemsPerPage;
    uint16  iItemsOnFirstPage;
    uint16  iLocationCode;      /* To get weather information */
    uint16  iLanguageCode;
    uint16  iParseWeather;      /* Switch On/Off */
    uint16  iParseRssFeeds;     /* Switch On/Off */
    uint16  iUseCache;          /* Switch On/Off */
};
```

4.10 Error Codes

All API functions return either a pointer to a data-structure, or an integer that indicates the success or failure of the action. Also the `ResultSet` carries en `iErrorCode` which can be checked for actual status of the last browse action. The error numbers are defined here:

```
enum return_value
{
    vTuner_OK = 0,
    vTuner_NotDefinedError = -1,
    vTuner_NoConnection = -2,
    vTuner_LoginFailed = -3,
    vTuner_BrowseCommandInProgress = -4,
    vTuner_NoAnswerFromServer = -5
};
```

5 Project file structure



5.1 The projects public interface

The public interface is reflected by the header files within `\include` directory.

File	Description
`vTuner_api.h`	Contains all functions, defined in this SRS
`vTuner_structs.h`	Contains the modules specific structs, defined in this API

vTuner_port.h	Defines the datatypes, used by the vTuner lib module. Needs to be adjusted for different Architectures
vTuner_http_client.h	Abstraction of a general http_client interface. **Not defined yet!** The user of the lib needs to implement the glue logic between this interface to the actual httpClient he's using within vTuner_http_client.c in the sources.
vTuner_xml_parser.h	Abstraction of a general xml_parser interface. **Not defined yet!** vTuner_xml_parser.c will implement this interface with the modules internal XML parser. The user could exchange this implementation by his own XML parser he might already be using in his project

5.2 The projects source files

The \source directory reflectes an internal structure of library.

File	**Description**
vTuner_api.c	Implement the public library interface which is defined in the vTuner_api.h header file.
vTuner_api_data_access.c	Implement data access part of library API.
vTuner_blowfish.h	Defines the encrypt/decrypt functions of blowfish algorithm which are implemented in the vTuner_blowfish.c file.
vTuner_blowfish.c	Implement the encrypt/decrypt functions of blowfish algorithm which will be used in the login process.
vTuner_global.h	Export global variables.
vTuner_global.c	Defines global variables.
vTuner_http_client.c	Implement the interface to the actual httpClient. The User have to register the http_get function to be called by library while performing the request.
vTuner_xml_parser.c	Implement the interface to the actual XML parser and the internal XML parser.
Makefile	Implement the library build rules under the different platform.
vTunerAPI.sln	MS VS Solution file.

6 Sample Application

To check the functionality of the API functions and to give the customer a reference implementation of how the vTuner API shall be used, a shell based test application is provided alogn with the library.

The application implements following commands:

- `help`

Prints out help screen containing list of commands and their short description.

- `login <predefined number>`

Login to a certain vTuner library. `"login"` without an argument lists the predefined vTuner URLs, when called with a number, the login procedure for the respective set is executed.

- `up, down, right, left`

To browse the actual database

- `bookmark`

Executes the command bookmark to the server.

- `page`

Prints current page contents. It is not necessary to write whole command. Simply pressing “p” key executes this command.

- `current`

Prints currently selected item. It is not necessary to write full command. Simply pressing “c” key executes this command.

- `libverbose <0/1>`

Selectes verbosity level of the library. Currently levels 0 and 1 supported (1 is more verbose, 0 is less verbose). Instead of typing full command user can simply type `lv <0/1>`

- `appverbose <0/1>`

Selectes verbosity level of the test application. Currently levels 0 and 1 supported (1 is more verbose, 0 is less verbose). Instead of typing full command user can simply type `av <0/1>`

- `ls`

Lists the complete content of this directory

- `lang <three letter language code>`

Selectes language for the library http requests.

- `search <single word to search for>`

This command performs search for the specified word. Please note, you can search only in the folder that allow to do that. (i.e. Root folder, or Root/Test/Test Search folder)

- `mtrack dump <nr nr>`

This command aids in memory leaks tracking.

- `quit`

Exits test application.

Exits test application. When application is compiled in debug configuration additionally to specified above it has following commands:

- `null-check`

This debug command performs API calls with NULL parameter instead of valid pointers to check that APIs check against such wrong functions usage.

- `wrong-config`

This debug command tries to set unvalid values to the configuration to make sure that the library API has ability to verify if configuration is valid before applying it.

- `expire`

This debug command forces login token to expire to make sure that application is capable of handling such situation when it appears for real.

The shell prompt shall show the current directory which is being browsed. E.g: `Genres/Blues>`

7 Test procedure

To test the ported vTuner library for its functionality the following tests must be passed:

7.1 Login procedure

Purpose:

Test login procedure as described in the vTuner specification in chapter 1 and 2.

Test tool:

Test application in debug mode.

Test description:

The `login` command is used to login to 2 different vTuner login sets and to do repeated logins test.

As soon as application starts type `av 1` (To see “login successful” or “login failed” messages)

a) company.vtuner.com with correct blowfish key set
b) company.vtuner.com with incorrect blowfish key set
    a. Blowfish key is wrong (you can set corrupted blowfish key by using `break-key 1` command. You can set valid key by using `break-key 0` command)
    b. Initial vector is wrong(you can set corrupted blowfish iv by using `break-iv 1` command. You can set valid iv by using `break-iv 0` command)
c) multiple repeated login retries

The different login sets are only available in the application when built in debug configuration. Appropriate messages shall be printed out in debug mode, to clearly check the internal success or correct failure in case b).

Test result:

In case a), the login process must be followed correct and the root directory is presented. In case b, the process must fail at some point. But it must be possible afterwards to login again with condition a), without restarting the program.

In case c), every successive login attempt should result in receiving root directory structure xml.

7.2 Browse Directories and Previous item

Purpose:

Test the properly linked structure by browsing into directories and backwards. Reference in vTuner specification: Type 1 XML and Type 3 XML (Page 8 and 11) .

Test tool:

Test application in debug mode.

Test description:

After login to the test environment of vTuner, the root directory is displayed. The tester now browses into each directory down to the very bottom and up again. It must be possible to return to the root directory by pressing '<' few times.

Test result:

No irregularities in browsing should be observed. Check the reference directory structure at the appendix.

7.3 Browse Items

Purpose:

This test is to confirm that all the items offered by vTuner XML are parsed correctly, and the information can be accessed through the API access functions.

Test tool:

Test application in debug mode.

Test description:

Browse the directory to find all kind of items. Browse each of them, to see that all the data is properly shown on the shell.

Test result:

Check the respective chapter in the vTuner specification:

7.3.1 Stations

Described as Type 2 XML on page 9

7.3.2 Show on Demand

Described as Type 4 XML on page 12

7.3.3 Show Episode

Described as Type 5 XML on page 14

7.3.4 Display Item

Described as Type 6 XML on page 16

Browse to Root/Test/Test Display directory. Make sure that list of 7 items displayed. Example:

DisplayText: 1
DisplayText: 2
DisplayText: 3
DisplayText: 4
DisplayText: 5
DisplayText: 6
DisplayText: 7

7.3.5 Search item

Described as Type 7 XML on page 17

7.3.6 Weather

Described as Type 8 XML on page 19

7.3.7 RSS Feeds

Described as Type 9 XML on page 20

7.4 Bookmark

Purpose:

This test is to confirm that bookmarking operates as described on page 15.

Test tool:

Test application in debug mode.

Test description:

a) Browse to any station. Bookmark it (by using `bookmark` command). Browse to Root/Favorites/My Favorites and find bookmarked station. Press right key.
b) Browse to Root/Favorites/My Favorites folder and select previously bookmarked station. Delete station from Favorites (by using `bookmark` command again)

Test result:

Bookmarked station appears in the Favourites directory and is selectable (i.e. pressing right key when it is selected leads some information about the station)..

7.5 Paging

Purpose:

This test is used to check whether the paging works ok under all circumstances. Background paging, scrolling over the list break (from last item to first item, and from first item to last item) etc.

Test tool:

Test application in debug mode.

Test description:

Browse to Root/Test/Test No Paging directory with long list.

a) Browse over the last item and through the whole list again few times in circles.
b) Browse up from first item and through the whole list in that direction few times in circles.
c) Execute `lv 1` command. Reopen the Root/Test/Test No Paging directory. Type `ls` command – make sure background task doesn't load pages on the background.
d) Execute `lv 1` command. Browse to Root/Test/Test Local US directory (it has many items in there). Type `ls` command – observer background task adding pages to the tree.

Test result:

a) Directory correctly displayed thru all items properly wrapping around first and last items.
b) Directory correctly displayed thru all items properly wrapping around first and last items.
c) Background paging task does not add items because they all should be loaded on the first browse attempt.
d) Background paging task properly adds items to the list as they are being received from vTuner server.

7.6 Cache

Purpose:

This test is used to check if caching works and cache is being used instead of making http requests when user hits back button (left key in test application)

Test tool:

Test application

Test description:

a) Browse to Root/Test/Test No-Cache directory. Interrupt network connection (unplug the cable, for instance). Hit back button.
b) Browse to Root/Tests/Test No-Cache/Directory1 directory. Interrupt network connection (unplug the cable, for instance). Hit back button.

Test result:

a) Application should load cached data and display right contents even though there is no active connection to vTuner server.
b) Application shows the folder of "New Stations", as this is entered as PreviousURL, which is being taken for going back when the NoCache flag is set.

7.7 Wrong parameters in calls to API functions

Purpose:

This test is used to check if API functions check input parameters pointers against NULL and if APIs can handle correctly wrong parameters in config structure.

Test tool:

Test application in debug mode

Test description:

a) Type in `null-check` command to run the test and hit enter key.
b) Type in `wrong-config` command to run the test and hit enter key.

Test result:

In both cases if application doesn't crash and user sees "Test completed" output – test is passed. (even though it might print bunch of errors in between)

7.8 Backup server URL operation

Purpose:

This test is used to check if library correctly uses backup url when primary server is not available.

Test tool:

Test application.

Test description:

Browse to the Root/Test/Test Timeout directory.

Test result:

Application should wait approximately 60 seconds(because primary url is erroneous) and then request data from backup URL. At last request from backup URL should succeed and user should see items in the folder.

7.9 Token expiration

Purpose:

This test is used to check if library correctly re-logins and obtains new token when currently used token expires.

This test was taken out, as vTuner doesn't make use of that feature yet.

7.10 Languages support

Purpose:

This test is used to check if library correctly supports various languages.

Test tool:

Test application.

Test description:

Login to the vTuner server. Select language by executing `lang` command (example: `lang dan`). Observe the root folder items. Repeat with other languages until all languages are tested.

Below is the list of supported language codes:

dan, dut, eng, fin, fre, ger, ita, jpn, nor, por, rus, spa, swe, chi, chs, pol, tur, kor

Test result:

All languages should be displayed correctly.

NOTE: The test site (company.vtuner.com) only supports English. This test command therefore can only be used for your own customized vTuner-site.

7.11 Exhaustive browsing

Purpose:

This test is used to check whether library has memory leaks or some programming logic errors.

Test tool:

Test application.

Test description:

execute the command "`browsetest`" on a certain directory. All items will be visited recursively.

Test result:

Test should run without error notices.

7.12 Memory usage control

Purpose:

This test is to track down memory usage, as well as possible memory leaks.

Test tool:

Test application in debug configuration

Test description:

This test requires that vTuner_malloc() and vTuner_free() log all the memory consumption and provide further information about its usage. (See Chapter 8). The command `mtrack` is used to print the current state about the memory consumption.

Login to vTuner server. Execute `mtrack` command. Remember corrunet memory consumption value. Browse various folder for a while. Relogin. Execute `mtrack` command.

Test result:

Memory consumption value from first and second execution of mtrack command should be the same.

7.13 Search test

Purpose:

This test should be used to test searching functionality.

Test tool:

Test application.

Test description:

Login to vTuner server. Browse to Root/Test/Test search folder. Execute `search bone` command.

Test result:

Test should find few stations or show on demand folders containing word “bone” in it. Make sure search doesn’t return falsely found stations (those that do not contain searched word).

7.14 Test Expire

This is additional folder for the Tests that will be described later.

7.15 Test Domain

Purpose:

This test should be used to test if library implementation is domain-independent.

Test tool:

Test application.

Test description:

Login to vTuner server. Browse to Root/Test/Test Domain folder.

Test result:

Tester should see DisplayText: Success item inside the Test Domain folder.

7.16 Test Path

Purpose:

This test should be used to test if library implementation is path-independent (it doesn’t rely on information where on the server vTuner script is running).

Test tool:

Test application.

Test description:

Login to vTuner server. Browse to Root/Test/Test Path folder.

Test result:

Tester should see DisplayText: Success item inside the Test Path folder.

7.17 Test Local US

This is additional folder for the Tests that will be described later.

7.18 URL Length Test

Purpose:
This test should verify that vTuner library has enough memory allocated for long URLs.

Test tool:
Test application.

Test description:
Browse to Root/Test/Test URL length folder.

Test result:
Tester should see DisplayText: Success item (as opposed to crashing application).

7.19 No Data Returned Test

Purpose:
This test should verify that vTuner library properly handles situation when for any reason there is no data returned.

Test tool:
Test application.

Test description:
Browse to Root/Test/ Test No data returned folder. After that browse to any other folder or station.

Test result:
Application should fail to enter Test No Data folder, but consequent browsing to other normal folders should succeed.

8 Memory tracking tool

8.1 Changes to vTuner_malloc / vTuner_free()

To track down the memory consumption and possible memory leaks, each allocation and free must be tracked and stored in a separate list. This list then can be checked and printed to see the current memory situation.

The following chained list is to be used to track the debugging information:

```
#ifdef  _DEBUG

struct memory_note{
       memory_note*   pNext;
       void*          p;
       uint16         iSize;
};

memory_note* memory_note_head;

#endif
```

vTuner_port.h must be changed the following way:

```
#ifdef _DEBUG

#define vTuner_malloc(x) vTuner_debug_malloc((x))
#define vTuner_free(x)  vTuner_debug_free(x);x=0


#else

#define vTuner_malloc(x) malloc((x))
#define vTuner_free(x)  free(x);x=0

#endif
```

The `vTuner_debug_malloc()` then performs an additional malloc with the size of a `memory_note` structure, within the size and the pointer of the original malloc is stored.

The `vTuner_debug_free()`, deletes such an entry, also freeing the `memory_note` itself `J`.

The shell function `mtrack()` needs to be implemented to print a summary of the memory consumption:

Example of the table:

```
#               Pointer     Size[byte]
-------------------------------------
1               0x34bd56ac        124
2               0x01234567       2024
-------------------------------------
current mem usage:                2148
max mem usage:                    2848
```

The command `mtrack [nr]` can be used to binary dump the respective memory block. This might help to find the caller for that block.

9 Appendix

9.1 vTuner directory structure of test environment

```
root
  + Favorites
  + Location
  + Genre
  + New Stations
  + Most Popular Stations
  + Podcasts By Genre
  + Podcasts By Location
  + Weather 10011 Postal Code Sample
  + Yahoo RSS Feed
  + Test
  |   +  Test URL Length
  |   +  Test Timeout
  |   +  Test No data returned
  |   +  Test Expire
  |   +  Test Domain
  |   +  Test Path
  |   +  Test No-Cache
  |   +  Test Display
  |   +  Test Search
  |   +  Test No Paging
  |   +  Test Local US
  + DisplayText: Time 7/24/2008 4:56:40 AM
  + DisplayText: Your ID# is 00080294102B
```