Ryft Open API Library User Guide

Ryft Document Number: 769

Document Version: 2.0.2

Revision Date: 20 May 2015

Revision History

Date	Reason for Change	Version
20150520	Changed name of document.	2.0.2
20150409	Added case-insensitive term frequency API calls.	2.0.1
20150316	First release to the public of the Ryft ONE Library API Users' Guide.	2.0.0
Various	Internal R&D working copies	0.x – 1.x

Table of Contents

Glossary

The following acronyms and abbreviations are used throughout this document and are defined here for convenience.

Word / Term	Definition	Brief Description
API	Application Programming Interface	The open interface that allows an end user to interact with the Ryft ONE.
CCC	Command and Control/Compiler	Manages the Ryft ONE analytics engine, providing the interface between user code and the Ryft ONE analytics hardware.
RDF	Record Definition File	Provides information needed by RHFS to properly chunk a given file type.
ROL	Ryft Open API Library	The library that abstracts and manages Ryft ONE hardware resources.
RHFS	Ryft Hybrid File System	The file system that manages data flow into and out of the Ryft ONE's advanced array of 48 solid state drives (SSDs).

1 Scope

This document explains the usage of the Ryft Open API Library. It describes the functions available in the API and explains the details of their usage. Additionally, usage examples are provided as a guide.

2 Overview

The Ryft Open API Library ("ROL") is the primary interface between an end user and the Ryft ONE analytics system. The ROL is provided as a standard Linux shared-object dynamic library (e.g., libryftone.so), and provides a set of hardware-accelerated primitive operations that can be performed on a set of data. User programs can then be written to execute a sequence of primitive operations on a set of data files in order to extract, modify, or analyze the data.

The ROL is provided as a C language library, and can be natively accessed using either the C or C++ languages. Additionally, language bindings for both Python and Java will be provided. Other languages, such as R and Scala, are easily supported using standard wrapper function mechanisms.

3 Concept of Operation

The analytics system and ROL have been designed to provide hardware accelerated processing of raw and record-based ASCII text data at the maximum possible speed. In order to achieve high speed processing, all input and output files that are used by ROL functions must be located in the Ryft Hybrid File System (RHFS). The RHFS is a purpose-built Linux file system which manages data striping across the solid state drives (SSDs) internal to the Ryft ONE 1U chassis. Files stored on RHFS can be accessed using any available Linux-supported protocol (such as scp/tftp/etc) or programmatically via standard file operations, such as the C library calls fopen(), fread(), and fwrite().

Once data has been loaded onto the RHFS, user programs can be executed to analyze, modify, or reduce the data using a variety of analytics primitives. Each user program processes a set of input files, and produces one or more output files containing the resultant data.

4 Available Functions

The ROL provides a basic set of functions:

- Open File
- Write To File
- Search (Exact Search and Fuzzy Search)
- Term Frequency
- Return Error String
- Operation Statistics

Each of these are discussed in the following sections.

4.1 Ryft Open API Library Data Type

While many parameters used by the ROL are standard C language types, `rol_result_t` is not. The `rol_result_t` data type is an opaque data type that is used to reference the data that is being operated on by calls to the Ryft Open API Library. For example, when using the library's Search function, the data to be searched will be of the `rol_result_t` type and the result of that search will also be of the `rol_result_t` type. The `rol_result_t` type cannot be used outside of the ROL.

NOTE: All library functions aside from the "Return Error String" function have an integer return value. The return value indicates whether the function call was a successful (return value of zero) or failed (non-zero value). Possible error (non-zero) return values are listed in *Appendix H – Error Codes*.

4.2 Open File

In order to operate on a set of data, the corresponding set of data files must first be opened. The `rol_process_files` function is used to open a set of data files, using this syntax:

```
int rol_process_files(
     rol_result_t*          result,
     const char*            list_of_files[],
     const uint16_t         number_of_files);
```

- **result** is a representation of the data contained in the input files.
- **list_of_files** is an array of pointers to the RHFS filenames to be processed.
  NOTE: All filenames are expected to be relative to the RHFS mount point, which is /ryftone by default. The RHFS mount point may be modified in the Ryft ONE Settings Database. The Ryft ONE Settings Database is described in more detail in *Appendix G - Ryft ONE Settings Database*.
- **number_of_files** is an integer specifying the number of filenames in the **list_of_files** array.

Any operations performed on the result of a `rol_process_files` call will be performed on all of the data contained in all of the specified RHFS files in the set.

NOTE: Only one `rol_process_files` call can be made in a user program, and it **must** be the first API call made in the user program. If it is not the first API call made, then a suitable error message will be generated at runtime.

4.3 Write To File

Once a set of data files is processed, the results must be written to a file on the RHFS. A search operation can output a set of files: an index file and a data file.

- The index file contains one set of data describing each match, with the set of data containing at least the offset of the match, length of the match, and match distance.
- The data file includes the actual match data, in the same format as the input.

The `rol_set_index_results_file` and `rol_set_data_results_file` functions are used to store result data to a file, using this syntax:

```
int rol_set_index_results_file (
     const rol_result_t*   input_data,
     const char*           index_file_location);

int rol_set_data_results_file (
     const rol_result_t*   input_data,
     const char*           data_file_location);
```

- **input_data** is the data or results to be written, which were generated during a previous step in the user program.
- **index_file_location** and **data_file_location** are the names of the files on the RHFS to which the index data and results are written, respectively.

  NOTE: All filenames are expected to be relative to the RHFS mount point, which is "/ryftone" by default. The RHFS mount point may be modified in the Ryft ONE Settings Database.

One, or both files, can be produced and written to disk. If a file is not required, set that filename to NULL, or don't call the associated function.

Index Results Files will have the following format; a single line for each match:

```
Filename,File Offset,Match Length,Actual Matched Distance
Filename,File Offset,Match Length,Actual Matched Distance
Filename,File Offset,Match Length,Actual Matched Distance
…
Filename,File Offset,Match Length,Actual Matched Distance
```

The example line from a search index results file, below, indicates that a match was found in the input file "/ryftone/passengers.txt" at character 31 in the file, that the match was 3 characters long, and that there was a "0" Hamming distance for the search.

```
/ryftone/passengers.txt,31,3,0
```

Similarly, the example line below, indicates that a fuzzy match was found where the resulting Hamming distance for the fuzzy match was 1.

```
/ryftone/passengers.txt,31,3,1
```

Note that index files have meaning only for search operations. If you attempt to create an index file for other primitives, such as term frequency, then an error will be generated at runtime.

Data Results Files will have the following format, with a line for each match:

```
[Data before Match Data][Match Data][Data after Match Data]
[Data before Match Data][Match Data][Data after Match Data]
[Data before Match Data][Match Data][Data after Match Data]
…
[Data before Match Data][Match Data][Data after Match Data]
```

The amount of data before and after the match data can be modified by changing the "surrounding width" parameter of the search function (see section *Search* below). Search results, regardless of whether they come from fuzzy or exact searches, will both follow this format. Note that fuzzy search result data files, unlike index fuzzy search result files, will not contain any indication of the distance parameter or matched distance.

Term frequency operation results are formatted differently than search operation results. A term frequency operation will analyze the input file and produce a single output file containing the unique terms, followed by three spaces, the frequency of each unique term, and finally a line feed (0x0A). For example, for a simplistic input text file that contains the data "the quick brown fox jumps over the lazy dog", the term frequency output file would be:

```
the   2
quick   1
brown   1
fox   1
jumps   1
over   1
lazy   1
dog   1
```

4.4 Search

The search function is used to search a set of input data for items that match a set of criteria. The input data can be organized as either unstructured raw data or record-based data. The output of the search can be either the matching data (in the same format as the input) or a set of indexes that describe where the match occurred within the input, or both.

The `rol_search` family of analytics primitives are used to search data, using this syntax:

```
int rol_search_exact(
     rol_result_t*          result,
     const rol_result_t*    input_data,
     const char*            match_criteria_query_string,
```

```
    const uint16_t          surrounding_width,
    const char*             delimiter_string);

int rol_search_fuzzy(
    rol_result_t*           result,
    const rol_result_t*     input_data,
    const char*             match_criteria_query_string,
    const uint16_t          surrounding_width,
    const uint8_t           fuzziness,
    const char*             delimiter_string);
```

- **result** is a representation of the results from the search.
- **input_data** is a representation of the data to be searched, which was configured via a prior call to `rol_process_files`.
- **match_criteria_query_string** specifies the data expression to be matched.
- **surrounding_width** specifies the number of characters before the match and after the match that will be returned when the input specifier type is raw text (see section *Search Query Criteria Specification* below for further details). For anything other than raw text, this field is ignored.
- **fuzziness** specifies a Hamming distance for a fuzzy data match operation.
- **delimiter_string** specifies text to append to the results output between search results. For example, it may be useful to place two carriage returns or some other text after each search result to assist with human readability, or with downstream machine parsing of results. Use NULL if you do not wish to have delimiter text inserted between search results.

`rol_search_exact` will search **input_data** using a match criteria of **match_criteria_query_string**. Only exact matches will be returned. When the format of the input data is specified as raw text, the **surrounding_width** parameter specifies how many characters around the match will be returned as part of the matched data results. The **surrounding_width** can be useful to assist a human analyst or a downstream machine learning tool to determine the contextual use of a specific matched term.

`rol_search_fuzzy` operates in exactly the same manner as `rol_search_exact` except that matches do not have to be exact. Instead, the **fuzziness** parameter allows the specification of a "close enough" value to indicate how close the input must be to **match_criteria_query_string**. Specifically, "close enough" is specified as a Hamming distance. The Hamming distance between two strings of equal length is the number of positions at which the corresponding symbols are different. To put it another way, Hamming distance measures the minimum number of substitutions required to change one string into the other, or the minimum number of errors that could have transformed one string into the other. In addition, similar to exact search, the **surrounding_width** mechanism can aid in downstream analysis of contextual use of the fuzzy match results.

4.4.1 Search Query Criteria Specification

Each `rol_search` family function requires a **match_criteria_query_string** parameter to specify how the search should be performed. A search criteria is made up of one or more relational expressions, connected using logical operations. A relational expression takes the following form:

```
(input_specifier relational_operator expression)
```

input_specifier specifies how the input data is arranged. The possible values are:

input_specifier Values	Description
RAW_TEXT	The input is a sequence of raw bytes with no implicit formatting or grouping
RECORD	The input is a series of records. Search all records.
RECORD.<field_name>	The input is a series of records. Search only the field called <field_name> in each record.

relational_operator specifies how the input relates to the expression. The possible values are:

relational_operator Values	Description
EQUALS	The input must match expression either exactly for an exact search or within the specified Hamming distance for a fuzzy search, with no additional leading or trailing data
NOT_EQUALS	The input must be anything other than expression
CONTAINS	The input must contain expression, and may contain additional leading or trailing data
DOES_NOT_CONTAIN	The input must not contain expression

expression specifies the expression to be matched. The possible values are:

expression Values	Description
A quoted string	Any valid C language string, including escaped characters. For example, “match this text\n”. This can also include hex representation, such as “\x48\x69\x20\x54\x68\x65\x72\x65\x21\x0D”.
A HEX value	Any value, expressed as a sequence of 2-character upper case hexadecimal, up to a maximum length of TBD. No leading ‘0x’ or trailing ‘h’ is required. This may be simpler to use in some cases than standard C language hex expansion strings. For example, 0A1B2C
A Wildcard	A ‘?’ character used to denote “any **single** character will match”. A ‘?’ can be inserted at any point(s) between quoted strings. For example, “match th”?”s text\n”
Any combination of the above	For example, “match\x20th”?”s string”0D

logical_operator allows for complex collections of relational expressions. The possible values are:

logical_operator Values	Description
AND	The logical expression (**a** AND **b**) evaluates to true only if both the relational expression **a** evaluates to true and the relation expression **b** evaluates to true.
OR	The logical expression (**a** OR **b**) evaluates to true if either the relational expression **a** evaluates to true or the relation expression **b** evaluates to true.
XOR	The logical expression (**a** XOR **b**) evaluates to true if either the relational expression **a** evaluates to true or the relation expression **b** evaluates to true, but not both.

Multiple relational_expressions can be combined using the logical operators AND, OR, and XOR. For example:

```
( record.city EQUALS "Rockville" ) AND ( record.state EQUALS "MD" )
```

Parentheses can also be used to control the precedence of operations. For example:

```
( ( record.city EQUALS "Rockville" ) OR ( record.city EQUALS "Gaithersburg" )
) AND ( record.state EQUALS "MD" )
```

4.5 Term Frequency

Term frequency is a data analytics tool. It peruses a document or set of documents and identifies unique terms using a set of known delimiters between terms. It also tabulates the number of times each of those unique terms appear in the document or set of documents. Ryft ONE implements the unique term delimiters as standard punctuation. So whitespace, periods, commas, and so on, will delimit terms, making the use of term frequency somewhat language agnostic. The result of the operation is a set of terms and their frequency of occurrence, thus the name “term frequency.”

This can be an extremely useful analytics tool for thinning large datasets, and comparing the resulting term frequency documents to other term frequency documents arrived at from other source data. For example, if two documents are found to mention the word ‘dog’ many times, then chances are there is some type of correlation between those two documents. This can be very useful for a variety of graphing applications, especially as they pertain to social media discoveries. This can also be useful for machine learning applications.

The term frequency family of functions computes the term frequency on a set of input data. The Ryft ONE supplies a grand total of six different term frequency functions, which are divided up to specify whether to compute a case-sensitive or case-insensitive term frequency, as well as to specify which input data format is used, which will in turn dictate how the data is interpreted.

Use this syntax with the term frequency primitive family:

```
int rol_rawtext_based_term_frequency(
     rol_result_t*          result,
     const rol_result_t*    input_data);

int rol_rawtext_based_term_frequency_i(
     rol_result_t*          result,
     const rol_result_t*    input_data);

int rol_record_based_term_frequency(
     rol_result_t*          result,
     const rol_result_t*    input_data,
     const char*            key_field_name);

int rol_record_based_term_frequency_i(
     rol_result_t*          result,
     const rol_result_t*    input_data,
     const char*            key_field_name);

int rol_field_based_term_frequency(
     rol_result_t*          result,
     const rol_result_t*    input_data,
     const char*            field_name,
     const char*            key_field_name);

int rol_field_based_term_frequency_i(
     rol_result_t*          result,
     const rol_result_t*    input_data,
     const char*            field_name,
     const char*            key_field_name);
```

- **result** is a representation of the term frequency results.
- **input_data** represents the data to be analyzed, which was configured via a prior call to `rol_process_files`.
- **field_name** specifies the field of each record to analyze.
- **key_field_name** specifies the name of the field to be used as the term frequency key.

Function names that end with "_i" are used for computing a case-insensitive term frequency, and functions that do not end in "_i" are used for case-sensitive term frequency. As an example, a case-insensitive term frequency analysis will consider the words "SMITH" or "Smith" as the same and combine the occurrences for the final output. On the other hand, a case-sensitive term frequency analysis would count occurrences of "SMITH" and "Smith" separately, along with any other possible capitalizations (e.g. "smith").

Below is a summary of the different input data formats as well as the output for each given function:

Term Frequency Input Data Type	Description	Output
Rawtext-based (rol_rawtext_based_term_frequency)	The input is a sequence of raw bytes with no implicit formatting or grouping.	The output is a list of term/frequency pairs separated by newlines with each line having three spaces between the unique term and its frequency of occurrence
Record-based (rol_record_based_term_frequency)	The input is a series of records. Compute the frequency of terms in the entire record.	The output is a list of unique terms and their respective frequency separated by three spaces **within each record** of the input, with filename and the value of the **key_field_name** included. Each record is analyzed separately. Example output: [filename][**key_field_name** value]: Term Frequency … Term Frequency
Field-Based (rol_field_based_term_frequency)	The input is a series of records. Compute the frequency of terms only in the field named <field_name>.	The output is a list of unique terms and their respective frequency separated by three spaces **within the specified field** of each record of the input, with filename and the value of the **key_field_name** included. Only the specified field of the record is analyzed, and all other fields are ignored. Each record is analyzed separately. Example output: [filename][**key_field_name** value]: Term Frequency … Term Frequency

4.6 Execute Algorithm

Once the appropriate input, output and algorithmic functions have been configured, the algorithm is executed via the `rol_execute_algorithm` function, using the syntax:

```
rol_result_t rol_execute_algorithm(
     const uint16_t        nodes_to_process
     void*                 (*percentage_callback)(uint8_t));
```

- **nodes_to_process** is an integer specifying the number of processing nodes that the algorithm should use. A minimally configured appliance ships from the factory with one processing node, and a maximally configured appliance ships with four processing nodes.
- **percentage_callback** references a user-defined callback function for handling incoming percentage completion messages. For example, a user could provide a callback function for printing the current status of the execution to standard output.
  Note: Providing a callback function is not required. To ignore percentage updates, provide a NULL parameter.

If the requested number of nodes are not available at the time of execution, the program is placed into a waiting queue and executed once the specified number of processing nodes become available. This allows a team sharing a maximally loaded appliance to manage resource allocation amongst the team, thereby providing a means to allow for multiple user operations to run simultaneously. Or, as performance needs dictate, all resources can be requested for an operation by specifying the total number of processing nodes available in the system.

The program queue is operated as a first-in/first-out construct. Therefore, once a program is placed into the queue, all programs after that will be placed into the queue behind that program even if they could be run first, to avoid starvation of the initially queued program. Terminating a program removes it from its place in the queue and any of its associated resources are released.

4.7 Return Error String

When a ROL function returns non-zero it is possible to retrieve an error message using the `rol_get_error_string` function. If any of the user program function calls to the Ryft ONE Library return non-zero, calling this function will provide more details as to what the issue is and where it originated. Use this syntax to request error details:

```
char* rol_get_error_string(void);
```

Error messages include the origin file of the error, the line at which the error occurred, as well as a brief description of the error. Errors will be reported for invalid user inputs as well as for any internal errors that occur.

As an example, suppose you've written a program using term frequency but have attempted to output an index file, which of course makes no sense for term frequency. In such a case, you might see a set of error output that resembles:

```
--->ProcessFile node (ID = 9):    /ryftone/passengers.txt
|------>Term Frequency node (ID = 10, parent = 9): RAW_TEXT
          input =
          output_index = ,
          output_results =
|---------->WriteResultsFile node (ID = 11, parent = (10):
              index =
              result = /ryftone/patout.txt
|---------->WriteResultsFile node (ID = 12, parent = (10):
              index = /ryftone/index_patout.txt
              result =

Compiler: Checking Tree For Errors ...     errors found:
Term Frequency node errors: (ID = 10):  Compiler: Logic Error: TermFreq (ID =
10) Term Frequency output can't be written in index format.
```

More detailed error information is available in `/var/log/syslog` or `/var/log/ryft/ryftone.log` if logging to these files has been enabled.

4.8 Operation Statistics

Statistics can be obtained for Search and Term Frequency operations using the `rol_get_statistics` function, using the syntax:

```
void rol_get_statistics(
     rol_result_t*           result,
     const char*             statistic_name,
     void*                   output)
```

- **result** is a representation of the operation for which to obtain results.
- **statistic_name** is a string representing the type of statistic to read.
- **output** is a void pointer to the statistic requested for the given operation. **Note:** The user will need to pay careful attention to re-cast to the data type of the requested statistic as statistics with varying data types may be added in the future.

NOTE: In order to obtain valid results, calls to the `rol_get_statistics` function should be made after the call to the `rol_execute_algorithm` function. If you run the statistics before executing the algorithm, the returned statistics are undefined.

Possible values for **statistic_name** are:

statistic_name Values	Data Type	Description	Units	Valid Usage with Operation
START_TIME	uint64_t	Time at which the operation began **Note:** Represented as Epoch time*	Milliseconds	Search Term Frequency

statistic_name Values	Data Type	Description	Units	Valid Usage with Operation
EXECUTION_DURATION	uint64_t	Amount of time taken to complete the specified operation	Milliseconds	Search Term Frequency
TOTAL_BYTES_PROCESSED	uint64_t	Amount of input data processed	Bytes	Search Term Frequency
TOTAL_NUMBER_OF_MATCHES	uint64_t	Number of matches found	N/A	Search
NUMBER_OF_TERMS	uint64_t	Total number of unique terms found in the input	N/A	Term Frequency

*Epoch time is defined here as the number of milliseconds that have elapsed since 00:00:00 UTC Thursday, January 1, 1970.

5 Creating and Executing a User Program

Let's consider a complex example where you have structured, tagged data and want to efficiently analyze it. For example, consider the following example which searches a set of XML-like databases that contain a list of employees, and finds all of the employees that live in the state of Maryland.

Step 1: Create Record Definition File

Create a record definition file that specifies the format of the data stored in the employee databases:

```
# Employee database record definition

# the size of a file chunk, in MB.  The file will be
# broken up into chunks of this size when loaded
# onto the analytics system
chunk_size_mb     = 1024;

# the file extension associated with this type of
# file and the type of record
file_glob         = "*.xml";
field_type        = "tagged";

# the start and end of the record can be identified
# using these strings
record_start      = "<Applicant>";
record_end        = "</Applicant>";

tags = (
  first_name      = ( "<first_name>", "</first_name>" ),
  last_name       = ( "<last_name>",  "</last_name>"  ),
  address         = ( "<address>",    "</address>"    ),
  zip_code        = ( "<zip>",        "</zip>"        )
);
```

NOTE: The system has built-in Record Definition Files for both .txt and .raw file extensions, so if the goal is to search unstructured data, there is no need to create RDFs if the files are given .txt or .raw extensions.

After creating the RDF, RHFS needs to be made aware of it using the following command from a Linux terminal:

```
  $ rhfsctl –a db.rdf
```

The `rhfsctl` tool is a Ryft program and is pre-loaded on the Ryft ONE system. It is described in more detail in *Appendix A – Record Definition Information*.

In the example above, after RHFS has been made aware of db.rdf, the db.rdf file can be deleted by the user if they wish. However, it may be useful to save the file in a known location for future reference (or to simplify future editing, should you need to update the definition). More details on the RDF requirements can be found in *Appendix A – Record Definition Information*.

Step 2: Create and Store Employee Database Files

Create and store the employee database files (there can be more than one) in the RHFS file system. In this example, there are two files. The first is `local_employees.db`:

```
<employee>
    <First_Name> Bob </First_Name>
    <Last_Name> Smith </Last_Name>
    <Address> 1111 Bobs Rd </Address>
    <City> Frederick </City>
    <State> MD </State>
    <Zip> 21701 </Zip>
</employee>
<employee>
    <First_Name> Bob </First_Name>
    <Last_Name> Jones </Last_Name>
    <Address> 2222 Bobs Rd </Address>
    <City> Rockville </City>
    <State> MD </State>
    <Zip> 20874 </Zip>
</employee>
<employee>
    <First_Name> John </First_Name>
    <Last_Name> Smith </Last_Name>
    <Address> 1111 Johns Rd </Address>
    <City> Charlestown </City>
    <State> WV </State>
    <Zip> 25414 </Zip>
</employee>
```

The second database file is `remote_employees.db`:

```
<employee>
    <First_Name> Mike </First_Name>
    <Last_Name> Smith </Last_Name>
    <Address> 1111 Mikes Rd </Address>
    <City> Las Vegas </City>
    <State> NV </State>
    <Zip> 89101 </Zip>
</employee>
```

Step 3: Create Query Program

Create the query program to invoke the appropriate Ryft API calls: `employee_query.c`

```
#include <libryftone.h>

int percent_callback(uint8_t percent_complete)
{
      printf("Percent Complete: %d\n",(int) percent_complete);
      return 0;
}

int main( int argc, char* argv[] )
{
     rol_result_t employee_list;
     rol_result_t employees_in_maryland;
     int requested_number_of_processing_nodes;

     const char* employee_databases[] =
     {
          "local_employees.db",
          "remote_employees.db"
     };

     if (argc != 2)
     {
          printf("ERROR: please provide a number of processing node resources
to execute on\n");
     }
     else
     {
          requested_number_of_processing_nodes = atoi( argv[ 1 ] );

          rol_process_files(&employee_list, employee_databases, 2);
          rol_search_exact(
                           &employees_in_maryland, &employee_list,
                           "( record.State EQUALS \"MD\" ) ", 0, NULL);
          rol_set_data_results_file (
               &employees_in_maryland, "employees_in_md.db");
          rol_execute_algorithm(requested_number_of_processing_nodes,
                                (void*)&percent_callback);
     }

    return 0;
}
```

Step 4: Compile and Link File To Ryft ONE Library

Once the source file is created, compile and link the file to the Ryft ONE Library. For example:

```
gcc -o employee_query employee_query.c –lryftone
```

Step 5: Execute the query.

Now, run the query. For example:

```
./employee_query 4
```

In this case the program is requested to be executed using four processing nodes within the analytics system. If four nodes are not available at the time of execution, the program is placed into a waiting queue and executed once four processing nodes become available.

Upon completion of the program's execution, the result file `employees_in_md.db` is generated and will reside in the RHFS mount point, which is /ryftone by default. The RHFS mount point may be modified in the Ryft ONE Settings Database. Here is an example:

```
<employee>
    <First_Name> Bob </First_Name>
    <Last_Name> Smith </Last_Name>
    <Address> 1111 Bobs Rd </Address>
    <City> Frederick </City>
    <State> MD </State>
    <Zip> 21701 </Zip>
</employee>
<employee>
    <First_Name> Bob </First_Name>
    <Last_Name> Jones </Last_Name>
    <Address> 2222 Bobs Rd </Address>
    <City> Rockville </City>
    <State> MD </State>
    <Zip> 20874 </Zip>
</employee>
```

6 Appendix A – Record Definition Information

In order to optimize the operational performance of the Ryft ONE acceleration hardware, the Ryft Hybrid File System requires contextual information (where available) describing the source content to be processed. This allows the file system to intelligently segment (or "chunk") the source content into optimized processing segments and store them in the manner most beneficial to the processing hardware.

These pieces of contextual information are referred to as Record Definition Files (RDF). These files provide the necessary information to create chunk files with complete records encapsulated. RDF files are created externally using a standard text editor (e.g. vi) and then submitted into RHFS and maintained using the **rhfsctl** command.

All files to be processed must have an associated RDF entry, otherwise attempts to write to the file system will be rejected with an "Invalid Argument" error code. As files are written to the file system, a check is made against the **file_glob** directive in each RDF and when a match is found the constraints are used to optimize the distribution of data on the storage array. Once submitted, an RDF can only be deleted, not modified.

RHFS comes preloaded with two RDF definitions for supporting raw binary and unstructured text files, with file extensions of **.raw** and **.txt**, respectively, which are detailed below.

6.1 Examples of Configuration Files

This section details different options for the construction of RDF files, in increasing complexity.

6.1.1 Minimal / Raw Binary Data Example RDF

The following example details the minimum requirements for an RDF. It is one of the system-supplied Record Definition Files and is used for storage raw data of any type without any implied knowledge as to the content of the data. For example, raw memory or network captures, randomization, etc.

Raw Binary Data Example RDF

```
# Raw Binary Format RDF ①
#
chunk_size_mb    = 64; ②
file_glob        = "*.raw"; ③
```

1. The first two lines are comments. Comments run from the detection of a # character or the // pattern to the end of the line, or can use the /* … */ syntax from the C programming language. Comment lines are options and can be placed at will throughout the RDF. Whitespace can be used throughout the RDF definition as necessary.
2. The third line declares the maximum size directive **chunk_size_mb**, specified in megabytes, of the chunks used to split the stored source data in the storage. Valid values for this are actively maintained in the Ryft ONE Settings Database, but currently are the integer values 1, 64, 256 or 1024. Attempts to use invalid types will cause the RDF to be rejected. Note that each of the

allocation units stored in the file system is independent of the chunk size. The allocation units themselves will grow to, but not exceed, 64 megabytes.

3. The final line uses the **file_glob** directive to declare the filename wildcard match used to determine which RDF applies to which file in the file system. In this case, any file in the file system whose name ends in "**.raw**" (e.g. /ryftone/data/foo.raw) will be processed using this RDF. Only one RDF with a particular glob can be active in the file system at a single time (i.e. two RDF's cannot be active with extension ".foo" concurrently.)

These two directives, **chunk_size_mb** and **file_glob**, are the minimum directives required for a valid RDF to be accepted. All statements in the RDF file end with a single semicolon. There is no limit on the number of RDFs that can be supported by the file system. Of note, in the case of files created using this extension, no content-based splitting will be applied to the file; only when a file size grows to exceed the **chunk_size_mb** field will the current allocation be truncated and a new back-end storage allocation be created and populated.

6.1.2 Unstructured Text Example RDF

Where the prior case dealt with raw binary data, the following RDF provides the means to instruct the RHFS to intelligently split the associated file based on a single ASCII space character.

Unstructured Text Example RDF

```
# Unstructured Text RDF
#
chunk_size_mb    = 64;
file_glob        = "*.raw";
record_end       = " ";
```

The only difference between this RDF and the previous example is the addition of the **record_end** directive, which provides a string to be used by the file system for determining the end of a "record."

In this case, since no other information was provided, a "record" is considered to be a word (i.e. consecutive alpha-numeric characters delimited by a single space character.) By supplying this, the file system ensures that a word is not potentially split across a storage allocation boundary.

The maximum size of a **record_end** directive is 32 bytes, and can be either be a single word without whitespace (e.g., address) or a quoted string which contains whitespace (e.g., "local address"). If a size larger than 32 is attempted, the `rhfsctl` utility will reject it. For more information about the `rhfsctl` utility, see the section *Installing an RDF and Associated Files*.

6.1.3 Tagged File Example RDF

Below an RDF is defined wherein more information about the content is provided to both allow the file system to better allocate the storage but also to provide context information to the Ryft ONE analytics engine for further computation. In this example, we assume that the content provided is in a textual XML-like format with an extension of .tag whereby the Ryft ONE is to be instructed to search for records within the document, delimited by their start and end tags (i.e. strings of text), wherein subfields are also declared for processing by the Ryft ONE.

Tagged File Example RDF

```
# Tagged File RDF
#
chunk_size_mb     = 64;
file_glob         = "*.tag";
record_start      = "<Applicant>"; ①
record_end        = "</Applicant>";
field_type        = "tagged"; ②

tags = { ③
  first_name      = ( "<first_name>", "</first_name>" ), ④
  last_name       = ( "<last_name>",  "</last_name>"  ),
  address         = ( "<address>",    "</address>"    ),
  first_name      = ( "<zip>",        "</zip>"        )
};
```

1. The **record_start** directive is used to inform analytics primitives which text delimits the start of a record for searching, but is ignored by the file system. A **record_end** directive is required for a **record_start** directive to be processed, and is subject to the same context restrictions as **record_end**. If both **record_start** and **record_end** are defined, both tags must start with the same character and end with the same character. In this case, both tags start with a less-than character and end with a greater-than character. Further, the starting and ending tag characters cannot appear elsewhere in either tag. For instance, a start tag of "<Invalid<Tag>" is invalid as the less than character appears at both the start of the tag and again later in the tag.
2. **field_type** is used to instruct what additional content amplification will be provided next in the RDF. Valid values are **tagged**, **delimited**, and **columnal**.
3. The **tags** directive is used to provide a list of the field delimiter information for potential use by the analytics engine. This directive can only be applied when the **field_type** directive is set to "tagged". This directive defines a list of (zero or more) pairs, all of which are enclosed in braces.
4. Inside the braces are the tagged field definitions themselves. They are defined with a string name (no spaces), an equals sign, then a parenthesized list of exactly two quoted strings, which represent the start and the end field tags respectively. They are subject to the same limitations as the **record_start** and **record_end** tags regarding size and content.

6.2 RDF Syntax Reference

RDF files are parsed using the `libconfig` package which is pre-installed on the Ryft ONE system. RDF files are subject to the `libconfig` parsing rules.

(See http://www.hyperrealm.com/libconfig/libconfig_manual.html for information and naming conventions on the definition of the settings and grouping constructs and their syntax.)

This is a table of the valid directives and their valid uses:

NOTE: Syntactically correct, yet unknown, directives are silently ignored.

Valid RDF Directives

Directive	Type	Required	Details
chunk_size_mb	Integer Setting	y	Must be 1, 64, 256, or 1024.
file_glob	Quoted String Setting	y	A string indicating the pathname match for the file extension being specified. For example, "*.xml"
record_start	Quoted String Setting	n	**record_end** must be present, 32 chars max.
record_end	Quoted String Setting	n	32 chars max
field_type	Quoted String Setting	n	One of: "tagged", "delimited", or "columnal"
tags	Group of Settings of Lists containing Quoted String Pairs	n	Use only when **field_type** is tagged. An example is given in the section Installing an RDF and Associated Files.

6.3 RDF Maintenance

RDF files are maintained in the system through the use of the `rhfsctl` command. This command provides four command line switches for administrating RDF files within RHFS:

Switch	Argument	Details
-a	Source RDF Filename	Reads an RDF file, submits it to RHFS, and upon syntax and content validation is added to the system. It will now be reported using the -R option below.
-R	(none)	Reports a list of all of the current RDF files by index and the associated file extension (taken from the RDF itself when submitted.)
-r	index of RDF	Prints the RDF source associated with the index number provided.
-d	Index of RDF	Removes the RDF entry from the system with the index number provided.

If, upon deletion of the RDF, there still exist files in the file system that were created using the just-deleted RDF index, then generated listings will contain the comment "(pending deletion)". Once all of the files that used that RDF have been deleted from the system, the RDF will be automatically purged as well.

7 Appendix B – RDF Usage Example

This appendix serves to illustrate how Record Definition Files are used with the Ryft ONE Library. For RDF formatting rules, see *Appendix A – Record Definition Information*.

7.1 Creating an RDF

The purpose of creating an RDF is to define how input data is formatted so that it can be properly analyzed. First, let's take a look at an example input file (with a fabricated file extension of "**.jbs**"), **passengers.jbs**

```
<passenger>
      <name>  Hannibal Smith        </name>
      <dob>   10-01-1928            </dob>
      <phone> 011-310-555-1212      </phone>
</passenger>
<passenger>
      <name>  DR. Thomas Magnum     </name>
      <dob>   01-29-1945            </dob>
      <phone> 011-310-555-2323      </phone>
</passenger>
<passenger>
      <name>  Steve McGarett        </name>
      <dob>   12-30-1920            </dob>
      <phone> 011-310-555-3434      </phone>
</passenger>
<passenger>
      <name>  Michael Knight        </name>
      <dob>   08-17-1952            </dob>
      <phone> 011-310-555-4545      </phone>
</passenger>
<passenger>
      <name>  Stringfellow Hawke    </name>
      <dob>   08-15-1944            </dob>
      <phone> 011-310-555-5656      </phone>
</passenger>
<passenger>
      <name>  Sonny Crockett        </name>
      <dob>   12-14-1949            </dob>
      <phone> 011-310-555-6767      </phone>
</passenger>
<passenger>
      <name>  MR. T Magnum          </name>
      <dob>   01-29-1946            </dob>
      <phone> 011-310-555-7878      </phone>
</passenger>
```

This input file is composed of "passenger" records, each with three fields: "name," "dob," and "phone."

Now, let's create a Record Definition File for this type of input file:

jbs.rdf

```
# Tagged File RDF
#
chunk_size_mb     = 64;
file_glob         = "*.jbs";
record_start      = "<passenger>";
record_end        = "</passenger>";
field_type        = "tagged";

tags = {
  name            = ( "<name>",  "</name>" ),
  dob             = ( "<dob>",   "</dob>"  ),
  phone           = ( "<phone>", "</phone>")
};
```

7.2 Installing an RDF and Associated Files

Now that we have created the **jbs.rdf** Record Definition File for **passengers.jbs** and any other input file with the associated "**.jbs**" extension, we need to install the RDF in order to load **.jbs** files into the Ryft Hybrid File System.

This can be done by using the following command:

```
  $ rhfsctl –a jbs.rdf
```

In order to display a list of Record Definition Files currently recognized by the RHFS, use the following command:

```
  $ rhfsctl -R
```

This should now display the following:

```
   1: *.raw             (system)
   2: *.txt             (system)
   3: *.jbs
```

Now you will be able to copy input files to the RHFS mount:

```
  $ cp passengers.jbs /ryftone
```

At this point, it is now possible to create a User Program to analyze any input files with a **.jbs** extension.

7.3 Example .jbs Record-Based Search

Here is an example user program to search for “310-555-5656”:

7.3.1 passengers.jbs Example User Program

```
#include <libryftone.h>
#include <stdio.h>

const char* input_files[] =
{
    "passengers.jbs"
};

rol_result_t passenger_list;
rol_result_t search_results;

void run_test(int num_nodes)
{
    rol_process_files(&passenger_list, input_files, 1);

    rol_search_exact(&search_results, &passenger_list, "(RECORD CONTAINS
\"310-555-5656\")", 0, "\n-\n");

    rol_set_index_results_file(&search_results, "jbs_index_results.txt");
    rol_set_data_results_file(&search_results, "jbs_data_results.jbs");

    rol_execute_algorithm(num_nodes, NULL);

}

int main(int argc, char* argv[])
{
    int requested_number_of_processing_nodes;

    if (argc != 2) // Don't run the program because # of nodes not requested
    {
        printf("ERROR: please provide a number of processing node resources
to execute on\n");
    }
    else // Run the program and request the number of nodes the user requests
    {
        requested_number_of_processing_nodes = atoi(argv[1]);
        run_test(requested_number_of_processing_nodes);
    }
    return 0;
}
```

7.3.2 passengers.jbs Example Output

Both data results and index results will be produced by this user program:

Data Results, stored in /ryftone/jbs_data_results.jbs

```
<passenger>
      <name>  Stringfellow Hawke    </name>
      <dob>   08-15-1944            </dob>
      <phone> 011-310-555-5656      </phone>
</passenger>
-
```

Index Results, stored in /ryftone/jbs_index_results.txt

```
/ryftone/passengers.jbs,522,130
```

7.4 Potential Errors Encountered

Below is a table of common errors that may be encountered while loading an RDF and associated files into the Ryft Hybrid File System:

Error Message	Cause
Adding RDF from jbs.rdf FAILED: RDF already exists with glob *.jbs	An RDF file with file_glob set to "*.jbs" has already been loaded. Use a new extension or remove the old RDF first (rhfsctl -d <RDF ID>).
Cannot access jbs.rdf. Aborting!	Tried to add an RDF file that does not exist in the current directory.
Adding RDF from jbs.rdf FAILED: syntax error at line 7	A syntax error was found in your RDF. For details, see RDF formatting rules in *Appendix A – Record Definition Information*.
Adding RDF from jbs.rdf FAILED: chunk_size_mb is not an acceptable value.	An unacceptable value for chunk_size_mb was given. The acceptable values are: 1, 64, 256 or 1024. Attempts to use invalid sizes will cause the RDF to be rejected.
mv: cannot create regular file '/ryftone/passengers.jlb': Invalid argument	Tried to move or copy a file with an extension that has not been defined by an RDF yet.

8 Appendix C – Searching Raw Text Examples

Suppose that you extracted a list of passengers that had flown through LAX airport on Sunday, October 5, 2014, and stored that information in a file called **passengers.txt**:

```
Hannibal Smith, 10-01-1928, 310-555-1212
DR. Thomas Magnum, 01-29-1945, 310-555-2323
Steve McGarett, 12-30-1920, 310-555-3434
Michael Knight, 08-17-1952, 310-555-4545
Stringfellow Hawke, 08-15-1944, 310-555-5656
Sonny Crockett, 12-14-1949, 310-555-6767
```

The following examples are based on this information file.

8.1 Example 1

If you were interested in finding all passengers who listed their birthday as “12-14-1949”, but weren’t sure if the formatting used ‘-‘ or ‘/’ to delimit the date fields, you could create the following program using a search fuzziness parameter of 2 to allow either ‘-‘ or ‘/’ to be recognized, and return 16 characters on each side of the match.

8.1.1 Example 1 User Program

```
#include <libryftone.h>

// Example callback function for printing percentage complete
// messages to stdout during execution.
int percent_callback(uint8_t percent_complete)
{
      printf("Percent Complete: %d\n",(int) percent_complete);
      return 0;
}

int main( int argc, char* argv[] )
{
     rol_result_t passenger_data;
     rol_result_t search_results;
     int requested_number_of_processing_nodes;

     const char* input_files[] =
     {
          "passengers.txt"
     };

     if (argc != 2)
     {
          printf("ERROR: please provide a number of processing"
                      "Node resources to execute on\n");
     }
     else
```

```
     {
          requested_number_of_processing_nodes = atoi( argv[ 1 ] );

          rol_process_files(&passenger_data, input_files, 1);
          rol_search_fuzzy(
                              &search_results,
                              &passenger_data,
                              "(RAW_TEXT CONTAINS \"12/14/1949\")",
                              16,
                              2,
                              NULL);

          rol_set_data_results_file (
                              &search_results, "results.txt");
          rol_execute_algorithm(requested_number_of_processing_nodes,
                                (void*)&percent_callback);
     }

    return 0;
}
```

8.1.2 Example 1 Output

This user program will create a data results file called “results.txt” located relative to the RHFS mount point. Data results files will return the matched text including the number of characters requested on each side of the match. The output file contains:

```
Sonny Crockett, 12-14-1949, 310-555-6767
```

NOTE: No indication of the resulting Hamming distance was provided. In this example, the Hamming distance was 2, since two ‘-‘ characters were found in place of the requested ‘/’ characters. If the distance of the match is important to the result or to subsequent analysis, then the `rol_set_index_results_file` call can be added to the source code to provide an index results output file, which includes Hamming distance information. For more information, see the section *Search*.

8.2 Example 2

If you were interested in finding all passengers who had a phone number of “310-555-2323”, but weren’t sure if the phone number was entered with a country code or not, and wanted to return 31 characters on each side of the match, then you could create the following program using an exact match to effectively ignore leading country code information.

8.2.1 Example 2 User Program

```
#include <libryftone.h>

int main( int argc, char* argv[] )
{
     rol_result_t passenger_data;
```

```
     rol_result_t search_results;
     int requested_number_of_processing_nodes;

     const char* input_files[] =
     {
          "passengers.txt"
     };

     if (argc != 2)
     {
          printf("ERROR: please provide a number of processing"
                      "Node resources to execute on\n");
     }
     else
     {
          requested_number_of_processing_nodes = atoi( argv[ 1 ] );

          rol_process_files(&passenger_data, input_files, 1);
          rol_search_exact(
                              &search_results,
                              &passenger_data,
                              "(RAW_TEXT CONTAINS \"310-555-2323\")",
                              31,
                              "\n-\n");

          rol_set_data_results_file (
                              &search_results, "results.txt");
          // NOTE: No callback defined in this example, so we set
          // the second parameter of rol_execute_algorithm to NULL
          rol_execute_algorithm(requested_number_of_processing_nodes, NULL);
     }

    return 0;
}
```

8.2.2 Example 2 Output

The user program above created a data results file called “results.txt” located relative to the RHFS mount point, containing the following results:

```
DR. Thomas Magnum, 01-29-1945, 310-555-2323
Steve McGarett, 12-30-1920, 31
-
```

8.3 Example 3

If you wanted to obtain Ryft ONE statistics for the previous example, you could modify your program to use the `rol_get_statistics()` function. In the following example, the user requests the total number of matches found while doing the search operation.

8.3.1 Example 3 User Program

```
#include <libryftone.h>
#include <inttypes.h>
#include <stdio.h>

int main( int argc, char* argv[] )
{
     rol_result_t passenger_data;
     rol_result_t search_results;
     int requested_number_of_processing_nodes;
     uint64_t search_num_matches;

     const char* input_files[] =
     {
          "passengers.txt"
     };

     if (argc != 2)
     {
          printf("ERROR: please provide a number of processing"
                      "Node resources to execute on\n");
     }
     else
     {
          requested_number_of_processing_nodes = atoi( argv[ 1 ] );

          rol_process_files(&passenger_data, input_files, 1);
          rol_search_exact(
                              &search_results,
                              &passenger_data,
                              "(RAW_TEXT CONTAINS \"310-555-2323\")",
                              31,
                              "\n\n");

          rol_set_data_results_file (
                              &search_results, "results.txt");

          rol_execute_algorithm(requested_number_of_processing_nodes, NULL);
          rol_get_statistics(&search_results,
                              "TOTAL_NUMBER_OF_MATCHES",
                              &search_num_matches);
          printf("Total Number of Matches Found: %"PRIu64"\n",\
                               search_num_matches);

     }
    return 0;
}
```

8.3.2 Example 3 Output

The output file will be exactly the same as in Example 2. The difference is that in addition to the generated result file, the following message will be printed to stdout:

```
Total Number of Matches Found: 1
```

8.4 Example 4

If you were interested in finding all passengers with the name “Mr. Thomas Magnum” and returning 26 characters of information on either side of the match, but you thought that he might be trying to evade a mob of fans by disguising himself as a “DR. Thomas Magnum” you could create the following program using a fuzziness parameter of 2.

8.4.1 Example 4 User Program

```
#include <libryftone.h>

int percent_callback(uint8_t percent_complete)
{
      printf("Fuzzy search is %d%% complete!\n", (int) percent_complete);
      return 0;
}

int main( int argc, char* argv[] )
{
     rol_result_t passenger_data;
     rol_result_t search_results;
     int requested_number_of_processing_nodes;

     const char* input_files[] =
     {
          "passengers.txt"
     };

     if (argc != 2)
     {
          printf("ERROR: please provide a number of processing"
                       "Node resources to execute on\n");
     }
     else
     {
          requested_number_of_processing_nodes = atoi( argv[ 1 ] );

          rol_process_files(&passenger_data, input_files, 1);
          rol_search_fuzzy(
                              &search_results,
                              &passenger_data,
                               "(RAW_TEXT CONTAINS \"Mr. Thomas Magnum\")",
                              26,
                              2,
                              "\n");
```

```
          rol_set_data_results_file (
                              &search_results, "results.txt");
          rol_execute_algorithm(requested_number_of_processing_nodes,
                              (void*) &percent_callback);
     }
    return 0;
}
```

8.4.2 Example 4 Output

The user program above created a data results file called “results.txt” located relative to the RHFS mount point, containing the following results:

```
 10-01-1928, 310-555-1212
DR. Thomas Magnum, 01-29-1945, 310-555-2323
```

9 Appendix D – Term Frequency Analysis of Raw Text Examples

Suppose now that you have a similar list of passengers as seen in earlier examples, but that there are two people named Sonny Crockett that flew through LAX on Sunday October 5th, 2014:

```
Hannibal Smith
DR. Thomas Magnum
Steve McGarett
Michael Knight
Stringfellow Hawke
Sonny Crockett
Sonny Crockett
```

9.1 Example 1

You could create the following program which uses the case-insensitive term frequency primitive to determine if there are any duplicates or passengers with the same name:

9.1.1 Example 1 User Program

```
#include <libryftone.h>
#include <stdio.h>
const char* input_files[] =
{
    "passengers.txt"

};
int display_percentage_complete(uint8_t percent_complete)
{
      printf("Execution is %d percent complete.\n", (int) percent_complete);
      return 0;
}
rol_result_t passenger_data;
rol_result_t term_frequency_results;
void term_frequency_example(int num_nodes)
{
    rol_process_files(&passenger_data, input_files, 1);
    rol_rawtext_based_term_frequency_i(&term_frequency_results,
                                       &passenger_data);
    rol_set_data_results_file(&term_frequency_results,
                              "term_frequency_results.db");
    rol_execute_algorithm(num_nodes,
                          (void*) &display_percentage_complete);
}
int main(int argc, char* argv[])
{
    int requested_number_of_processing_nodes;
    if (argc != 2) // Don't run the program because # of nodes not requested
    {
        printf("ERROR: please provide a number of processing node resources"
               "to execute on\n");
```

```
    }
    else // Run the program and request the number of nodes the user requests
    {
        requested_number_of_processing_nodes = atoi(argv[1]);

        term_frequency_example(requested_number_of_processing_nodes);

    }
    return 0;
}
```

9.1.2 Example 1 Output

Note that the output here assumes the case in-sensitive term frequency function call, as utilized in the example above. For case-insensitive term frequency results, all terms are always provided in lowercase in the output:

```
     hannibal   1
     smith   1
     dr.   1
     thomas   1
     magnum   1
     steve   1
     mcgarett 1
     michael   1
     knight  1
     stringfellow   1
     hawke   1
     sonny   2
     crockett   2
```

If, on the other hand, a case sensitive term frequency function call had been used, the output would look like:

```
     Hannibal   1
     Smith   1
     DR.   1
     Thomas   1
     Magnum   1
     Steve   1
     McGarett 1
     Michael   1
     Knight  1
     Stringfellow   1
     Hawke   1
     Sonny   2
     Crockett   2
```

10 Appendix E – Error Checking Examples

The Ryft ONE Library supports error checking through the return values of the available functions, as well as, through the `rol_get_error_string()` function. This section illustrates the usage of both.

NOTE: A return value of 0 represents a successful function call. Any non-zero value represents an error. Error codes for a variety of error cases have been defined and are listed in *Appendix H – Error Codes*.

10.1 Example 1

In this example, the user attempts to do an exact search on records in "passengers.brs" for the text "310". The return values of `rol_process_files()`, `rol_set_data_results_file()`, and `rol_execute_algorithm()` are checked to verify that no errors occurred. If an error occurs, the error string returned by calling the `rol_get_error_string()` function is printed to standard output. In this scenario, the user has set the data results output file to be "data_results.db". Since the input file is a ".txt" file, and since it is required that the data results output file have the same extension as the input file, this will naturally lead to an error.

10.1.1 Example 1 User Program

```
#include <libryftone.h>
#include <stdio.h>

int main( int argc, char* argv[] )
{
	rol_result_t passenger_data;
	rol_result_t search_results;
	int requested_number_of_processing_nodes;
	int ret_val = 0;

	char* input_files[] =
	{
		"passengers.brs"
	};

	if (argc < 2)
	{
		printf("%s:  Usage: %s <number_of_processing_nodes>\n", argv[0],
argv[0]);
		return 0;
	}
	else
	{
		requested_number_of_processing_nodes = atoi( argv[ 1 ] );

		ret_val = rol_process_files(&passenger_data, (const char
**)input_files, 1);
		if (ret_val != 0)
		{
			printf("%s:\n", rol_get_error_string());
		}
```

```
            else
            {
                  rol_search_exact(&search_results, &passenger_data, "(RECORD
CONTAINS \"310\")", 0, 0);
                  rol_set_index_results_file (&search_results,
"index_results.txt");
                  ret_val = rol_set_data_results_file (&search_results,
"data_results.db");
            }
      }
      if (ret_val != 0)
      {
            printf("%s:\n", rol_get_error_string());
      }
      else
      {
            ret_val =
rol_execute_algorithm(requested_number_of_processing_nodes, NULL);
      }

      if (ret_val != 0)
      {
            printf("%s:\n", rol_get_error_string());
      }
      return 0;
}
```

10.1.2 Example 1 Output

Running this user program will result in the following error printed to standard output:

```
Error From CCC: prog_9021:
Compiler: Compiling program:
--->ProcessFile node (ID = 852):    /ryftone/passengers.brs
|------>Search node (ID = 853, parent = 852):
        (RECORD CONTAINS "310") (surroundingWidth=0)
           input =
           output_index = ,
           output_results =
|---------->WriteResultsFile node (ID = 854, parent = (853):
               index = /ryftone/index_results.txt
               result =
|---------->WriteResultsFile node (ID = 855, parent = (853):
               index =
               result = /ryftone/data_results.db

Compiler: Checking Tree For Errors ...     no errors found
Compiler: Checking Tree For Logic Errors ...     no logic errors found
Compiler: Checking Index File Extensions ...     successful.
Compiler: Checking Result File Extensions ...     failed:
Node Search (ID = 853): expected extension .brs but found .db
```

```
Received an error from the CCC!
:
```

10.2 Example 2

The following example prints error codes along with the error text in case of an error. This time, the user has misspelled the word “CONTAINS” in the `rol_search_exact()` call.

10.2.1 Example 2 User Program

```
#include <libryftone.h>
#include <stdio.h>

int main( int argc, char* argv[] )
{
      rol_result_t passenger_data;
      rol_result_t search_results;
      int requested_number_of_processing_nodes;
      int ret_val = 0;

      char* input_files[] =
      {
            "passengers.brs"
      };

      if (argc < 2)
      {
            printf("%s:  Usage: %s <number_of_processing_nodes>\n", argv[0],
argv[0]);
            return 0;
      }
      else
      {
            requested_number_of_processing_nodes = atoi( argv[ 1 ] );

            ret_val = rol_process_files(&passenger_data, (const char
**)input_files, 1);
            if (ret_val != 0)
            {
                  printf("%s:\n", rol_get_error_string());
                      printf("Error code: %d\n", ret_val);
            }
            else
            {
                  rol_search_exact(&search_results, &passenger_data, "(RECORD
CONTINS \"310\")", 0, 0);
                  rol_set_index_results_file (&search_results,
"index_results.txt");
                  ret_val = rol_set_data_results_file (&search_results,
"data_results.brs");
            }
      }
```

```
      if (ret_val != 0)
      {
            printf("%s:\n", rol_get_error_string());
                printf("Error code: %d\n", ret_val);
      }
      else
      {
            ret_val =
rol_execute_algorithm(requested_number_of_processing_nodes, NULL);
      }

      if (ret_val != 0)
      {
            printf("%s:\n", rol_get_error_string());
                printf("Error code: %d\n", ret_val);
      }
      return 0;
}
```

10.2.2 Example 2 Output

The following output is printed to standard output. Note that the error code displayed is "-9", which is used to indicate that a function call prior to this one has resulted in an error and that as a result, this call has been ignored.

```
Error From CCC: prog_5007:
Compiler: Compiling program:
--->ProcessFile node (ID = 868):    /ryftone/passengers.brs
|------>Search node (ID = 869, parent = 868):
        (RECORD CONTINS "310") (surroundingWidth=0)
           input =
           output_index = ,
           output_results =
|---------->WriteResultsFile node (ID = 870, parent = (869):
               index = /ryftone/index_results.txt
               result =
|---------->WriteResultsFile node (ID = 871, parent = (869):
               index =
               result = /ryftone/data_results.brs

Compiler: Checking Tree For Errors ...     errors found:
Search node query errors: (ID = 869):

    Input string = (RECORD CONTINS "310")
    Error text = syntax error, unexpected any unquoted string

Received an error from the CCC!
:
Error code: -9
```

11 Appendix F – File Extension Requirements

In order to assure proper functionality, the user must adhere to the following requirements when building a user program:

- Any file used as an input to an operation must have an associated Record Definition File (RDF) that describes the file, thereby registering the extension with the analytics system. The system has built-in RDFs for .txt (which will include files without extensions as well) and .raw files.
- A raw text search will accept any registered file extensions.
- All other searches require that:
    - If multiple input files are given, they must all have the same extension
    - If an index output file is specified, it must have either a .txt extension or no extension
    - If a result output file is specified, it must have the same extension as the input files
- Term Frequency operations require that:
    - If multiple input files are given, they must all have the same extension
    - If a result file is specified, it must have either a .txt extension or no extension

12 Appendix G - Ryft ONE Settings Database

The Ryft ONE Settings Database stores the settings for all Ryft ONE software components. The settings database file should not be manipulated directly, instead the Ryft-provided `sdbutil` (Settings Database Utility) should be used to view and/or change setting values. The following describes the basic usage of `sdbutil`.

The various settings stored in the database are grouped into sections. To view all of the section names use the "sections" command:

```
$ sdbutil sections
Ryft ONE Settings Database Utility

Number of sections: 5
------------------------------------------------------------------------
--

  logging
  notification_actions
  notification_events
  RHFS
  CCC
```

To view the values in a section use the "display" command and specify the section name:

```
$ sdbutil display CCC
Ryft ONE Settings Database Utility

Section: CCC (2 settings)
------------------------------------------------------------------------
--

         Name: TCP_PORT
         Type: INTEGER
        Value: 7000
   Validation: ^([1-9][0-9]{0,3}|[1-5][0-9]{4}|6[0-4][0-9]{3}|65[0-4][0-
9]{2}|655[0-2][0-9]|6553[0-5])$
  Description: The TCP port used to pass messages between the front-end and
the CCC.

         Name: TEMP_DIR
         Type: STRING
        Value: /tmp/ryftone/
   Validation: ^/[^\0]+
  Description: The location where temporary files generated by CCC are
stored.
```

To change the value of a setting, use the “update” command and specify the section name, the setting name, and a new value. Note the setting values are validated using the regular expression shown above, on lines labeled “Validation”.

```
$ sdbutil update CCC TCP_PORT 7001
Ryft ONE Settings Database Utility

Section: CCC
Setting: TCP_PORT
New value: 7001
Updated successfully
```

13 Appendix H – Error Codes

Name of Error Code	Value	Cause of Error
ERROR_CODE_INVALID_MESSAGE	-5	Received an invalid / unknown message from CCC Manager (potential API version issue)
ERROR_CODE_SEND_FAILURE	-7	An issue occurred when attempting to send a message to the CCC.
ERROR_CODE_TIMEOUT	-8	The library did not receive an acknowledgment or keep-alive in a given time and has timed out.
ERROR_CODE_ERROR_FROM_CCC	-9	User received an error message from the CCC.
ERROR_CODE_ERROR_FROM_PREVIOUS	-11	An error already occurred in a previous library function call, so this error is returned and the operation request is ignored.
ERROR_CODE_UNKNOWN_ERROR	-25	An unknown error has occurred.
ERROR_CODE_ABNORMAL_TERMINATION	-22	The CCC sent a termination message response with an abnormal status.
ERROR_CODE_INVALID_NUMBER_OF_FILES	-27	Fewer files provided than specified in rol_process_files call.
ERROR_CODE_UNKNOWN_MESG	-30	An unknown message type was received by the CCC.
ERROR_CODE_INVALID_API	-31	API version mismatch between CCC and libryftone.
ERROR_CODE_INVALID_NODES	-32	Invalid number of processing nodes requested.
ERROR_CODE_MISSING_PROCESS_FILES	-37	Process files command was not called first.
ERROR_CODE_ERROR_PROCESSING_MESSAGE	-40	The CCC encountered an error while processing.
ERROR_CODE_UNINITIALIZED_RESULT	-41	Program attempts to use an uninitialized result variable.
ERROR_CODE_EMPTY_PROGRAM	-42	No operations were requested before the execute function was called.