Ryft One Library API User's Guide

Document ID: TBD

Document Version: 1.8.2

Revision Date: 15 January 2015

Revision History

Date	Reason for Change	Version
20150115	Added rol_get_statistics function	1.8.2
20150114	Added key_field_name parameter to rol_term_frequency function	1.8.1
20141223	Fixed examples and updated function parameters	1.8.0
20141203	Minor updates and corrections, addition of error codes to Appendix C	1.7.0
20141119	Finished updating examples and library details, added [EMPTY!] error codes section to appendix	1.5.0
20141114	Minor updates to reflect changes to the library	1.3.0
20141107	Minor updates to reflect changes to the library	1.2.0
20141007	Fixed minor errors in coding examples; Added API description for function rol_execute_algorithm	1.1.0
20141006	First Draft	1.0.0

Table of Contents

List of Figures

No table of figures entries found.

List of Tables

No table of figures entries found.

Glossary

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

Word / Term	Definition
ROL	Ryft One Library
ROL API	Ryft One Library API
TFS	The File System
UP	User Programs

1 Scope

This document is intended to explain the usage of the Analytics system Ryft One Library API. 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 One Library API ("ROL API") is the primary interface between an end user and the analytics system. The ROL API 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 ("UP") 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 API 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 are provided.

3 Concept of Operation

The analytics system and ROL API 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 primitives must be located in the TFS file system. The TFS file system is a proprietary Linux file system located at an appropriate mount point, and can be accessed using any available Linux protocol (such as scp/tftp/etc) or programmatically via standard file operations (such as C library fopen/fread/fwrite).

Once data has been loaded onto the TFS file system UPs can be executed to analyze, modify, or reduce the data using primitive operations. Each UP processes a set of input files, and produces a single output file containing the resultant data.

4 Available Functions

The ROL API provides a basic set of primitive functions:

- Open File
- Write To File
- Search
- Sort
- Term Frequency
- Return Error String
- Statistics

Each of these primitives is discussed in the following sections.

4.1 Ryft ONE Library Data Type

While many parameters used by the ROL API 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 ONE Library. As an example, if using the library's Search function, the data to be searched will be of type rol_result_t and the result of that search will also be of type rol_result_t. The rol_result_t type cannot be used outside of the ROL API.

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 C.

4.2 Open File

In order to operate on a set of data, the corresponding set of data files must first be opened. Use the `rol_process_files` primitive to open a set of data files. Use this syntax for the `rol_process_files` primitive:

```
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 filenames to be processed.
- ***number_of_files*** is an integer specifying the number of items 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 files in the set.

NOTE: Only one `rol_process_files` primitive call can be made in a UP, and it **must** be the first API call made in the UP.

4.3 Write To File

Once a set of data files is processed, the results must be written to a file on the TFS file system. Use the `rol_set_index_results_file` and `rol_set_data_results_file` functions to store the 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 UP.
- ***index_file_location*** and ***data_file_location*** are the names of the files on the TFS file system to which the index data and results are written, respectively.

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.

One, or both files, can be produced and written to disk. If a file is not required, set that filename to NULL.

Running a sort or term frequency operation results in a different results format than the search operation:

- A sort operation will produce a single data file containing the sorted data records in the same format as the input.
- A term frequency operation will produce a single output file containing the frequency of each term in the input.

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.

Use the `rol_search` family of primitives to search the 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);

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);
```

- ***result*** is a representation of the results from the search.
- ***input_data*** is a representation of the data to be searched.
- ***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 the description of the 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.

`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.

`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 by 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.

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 the entire record
RECORD.<field_name>	The input is a series of records. Search only the field called <field_name>

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

relational_operator Values	Description
EQUALS	The input must match expression exactly, 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"
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. 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 th"?"s string"0D

Multiple relational_expressions can be combined using the logical operators AND, OR, and XOR. These logical operators behave in the same manner as they do in the C language. 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 Sort

The `rol_sort` function family sorts a set of input based on a key field. The input data must be organized as a set of records, with each record having a key field. The output is organized in the same format as the input. Use this syntax with the `rol_sort` primitive family:

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

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

- **result** is a representation of the sorted data.
- ***input_data*** contains the data to be sorted.
- ***key_field_name*** specifies the name of the field to be used as the sort key.

`rol_sort_ascending` will sort *input_data* in ascending order, using the field specified by *key_field_name*. `rol_sort_descending` will sort *input_data* in descending order, using the field specified by *key_field_name*.

4.6 Term Frequency

The `rol_term_frequency` function computes the term frequency on a set of input data. The output is a list of term/frequency pairs, with one pair for each unique term in the input. Use this syntax with the `rol_term_frequency` primitive:

```
rol_result_t rol_term_frequency(
     rol_result_t*          result,
     rol_result_t*          input_data,
     const char*            input_data_format
     const char*            key_field_name);
```

- ***result*** is a representation of the term frequency results.
- ***input_data*** represents the data to be analyzed.
- ***input_data_format*** specifies how the data should be interpreted.
- ***key_field_name*** specifies the name of the field to be used as the term frequency key.

Possible values for *input_data_format* are:

input_data_format Values	Description	Output
RAW_TEXT	The input is a sequence of raw bytes with no implicit formatting or grouping	The output is a list of terms and their respective frequency within the input
RECORD	The input is a series of records. Search the entire record	The output is a list of terms and their respective frequency **within each record** of the input. Each record is analyzed separately
RECORD.<field_name>	The input is a series of records. Search only the field called <field_name>	The output is a list of terms and their respective frequency **within the specified field** of each record of the input. Only the specified field of the record is analyzed, and all other fields are ignored. Each record is analyzed separately

4.7 Execute Algorithm

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

```
rol_result_t rol_execute_algorithm(
     const uinst16_t        nodes_to_process);
```

- ***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.

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 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 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 starving scenarios for the initially queued program.

4.8 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 UP 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. More detailed error information can also be seen in `/var/log/syslog` or `/var/log/ryft/ryftone.log` if logging to these files has been enabled by the user.

4.9 Statistics

Statistics can be obtained for Search, Sort, and Term Frequency operations using the rol_get_statistics function. Use this syntax to request statistics for a given operation:

```
void rol_get_statistics(
     rol_result_t*           result,
     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 the data type of the requested statistic as other statistics with different data types are added.

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.

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 Sort
EXECUTION_DURATION	**uint64_t**	Amount of time taken to complete the specified operation	Milliseconds	Search Term Frequency Sort
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

The first step in creating a UP is to create the C source file that contains the ROL API primitive operation calls. For example, consider the following example which searches a set of XML-like databases that contain a list of employees, finds all of the employees that live in MD, and then sorts them by last name.

Step 1: Create a record definition file that specifies the format of the data stored in the employee databases:

```
# Employee database record definition

# the file extension associated with this type of
# file and the type of record
extension  = ".db";
type       = variable;

# 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 = 1024;

# the start and end of the record can be identified
# using these "tags"
record = ( ( "<employee>" ), ( "</employee >" ) );

fields = (
  ( "First_Name",  "<First_Name>", "</First_Name>" )
  ( "Last_Name",   "<Last_Name>",  "</Last_Name >" )
  ( "Address",     "<Address>",    "</Address>" )
  ( "City",        "<City>",       "</City>" )
  ( "State",       "<State>",      "</State>" )
  ( "Zip",         "<Zip>",        "</Zip>" )
);
```

Step 2: Create and store the employee database files (there can be more than one) on the analytics system 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 the query program: employee_query.cc

```
#include <libryftone.h>

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

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

     if (argc != 2)
     {
          printf("%s: 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);
          rol_sort_ascending(
                             &sorted_employees_in_maryland,
                             &employees_in_maryland, "Last_Name");
          rol_set_data_results_file (
               &sorted_employees_in_maryland, "sorted_employees_in_md.db");
          rol_execute_algorithm(requested_number_of_processing_nodes);
     }

    return 0;
}
```

Step 4: Once the query sort file is created, compile and link the file to the PL. For example:

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

Step 5: Execute 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 resulting file is generated:
sorted_employees_in_md.db

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

6 Appendix A – Searching Raw Text Examples

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

```
Hannibal Smith, 10-01-1928,011-310-555-1212
DR. Thomas Magnum, 01-29-1945,310-555-2323
Steve McGarett, 12-30-1920,310-555-3434
Michael Knight, 08-17-1952,011-,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.

6.1 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 '-' to be replaced by '/', and return 16 characters on each side of the match:

```
#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_fuzzy(
                              &search_results,
```

```
                              &passenger_data,
                              "(RAW_TEXT CONTAINS \"12-14-1949\")",
                              16,
                              2 );

          rol_set_data_results_file (
                              &search_results, "results.db");
          rol_execute_algorithm(requested_number_of_processing_nodes);
     }

    return 0;
}
```

6.2 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 21 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:

```
#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\")",
                              21);

          rol_set_data_results_file (
```

```
                              &search_results, "results.db");
          rol_execute_algorithm(requested_number_of_processing_nodes);
     }

    return 0;
}
```

6.3 Example 3

If you were interested in finding all passengers with the name "Mr. Thomas Magnum" and returning 25 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 a "Dr. Thomas Magnum" you could create the following program using a fuzziness parameter of 2:

```
#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_fuzzy(
                              &search_results,
                              &passenger_data,
                               "(RAW_TEXT CONTAINS \"Mr. Thomas Magnum\")",
                              25,
                              2 );

          rol_set_data_results_file (
                              &search_results, "results.db");
          rol_execute_algorithm(requested_number_of_processing_nodes);
     }

    return 0;
}
```

6.4 Example 4

The example below illustrates the usage of the rol_get_statistics function. In this example, the user requests the execution duration of the Sort operation once the execution is complete:

```
#include <libryftone.h>

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

char test_results[10000];
const char* employee_databases[] =
{
    "local_employees.db",
    "remote_employees.db",
    "contractors.db"
};

rol_result_t employee_list;
rol_result_t employees_in_maryland;
rol_result_t sorted_employees_in_maryland;

int main(int argc, char* argv[])
{
    int requested_number_of_processing_nodes;
    uint64_t sort_exec_time;
    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]);
        rol_process_files(&employee_list, employee_databases, 3);
        rol_search_exact(&employees_in_maryland, &employee_list,
                         "(RAW_TEXT CONTAINS  \"MD\")", 5);
        rol_sort_ascending(&sorted_employees_in_maryland,
                           &employees_in_maryland, "Last_Name");
        rol_set_index_results_file(&sorted_employees_in_maryland,
                                   "index_results.txt");
        rol_execute_algorithm(requested_number_of_processing_nodes);
        rol_get_statistics(&sorted_employees_in_maryland,
                           "EXECUTION_DURATION",
                           &sort_exec_time);
        printf("Execution Duration of Sort Operation %"PRIu64"\n",\
               sort_exec_time);
    }
    return 0;
}
```

7 Appendix B – Future Release Functionality

The following features are currently under consideration for inclusion into future releases:

- Case insensitive searching
- Month/date transposition
    - Note that month/date transposition can be handled in the current API for some use cases by utilizing a fuzzy match with a relatively high distance, since the maximum distance for encodings of "MM/DD/YYYY" and "DD/MM/YYYY" is four.

This list is subject to change over time as new functionality and extensions are identified.

8 Appendix C – Error Codes

Name of Error Code	Value	Cause of Error
ERROR_CODE_SEND_FAILURE	-7	Attempted to send a message to the CCC Manager, failed
ERROR_CODE_TIMEOUT	-8	Semaphore or message queue receive timed out
ERROR_CODE_INVALID_MESSAGE	-5	Received an invalid / unknown message from CCC Manager (potential API version issue)
ERROR_CODE_ERROR_FROM_CCC	-9	Received an error message from the CCC Manager
ERROR_CODE_ERROR_FROM_PREVIOUS	-11	Previous RYFT One Library call resulted in an error, skipping this function call and returning this error code for remaining calls
ERROR_CODE_ERROR_CODE_UNEXPECTED	-15	Received an unexpected message when waiting for a response message
ERROR_CODE_UNKNOWN_ERROR	-25	An unknown error has occurred
ERROR_CODE_NORMAL_TERMINATION	25	Received normal termination from CCC Manager - not really an error
ERROR_CODE_ABNORMAL_TERMINATION	-22	Received abnormal termination from the CCC Manager
ERROR_CODE_INVALID_NUMBER_OF_FILES	-27	User gave fewer files than they specified in number_of_files parameter