GCC Code Coverage Report

			Exec	Total	Coverage
Directory:	./				
File:	storage/kvstore/filesystemstore/source/FileSystemStore.cpp	Lines:	217	321	67.6 %
Date:	2021-05-06 12:39:05	Branches:	98	254	38.6 %

```
Line Branch Exec  Source
   1                /* mbed Microcontroller Library
   2                 * Copyright (c) 2018 ARM Limited
   3                 *
   4                 * SPDX-License-Identifier: Apache-2.0
   5                 *
   6                 * Licensed under the Apache License, Version 2.0 (the "License");
   7                 * you may not use this file except in compliance with the License.
   8                 * You may obtain a copy of the License at
   9                 *
  10                 *     http://www.apache.org/licenses/LICENSE-2.0
  11                 *
  12                 * Unless required by applicable law or agreed to in writing, software
  13                 * distributed under the License is distributed on an "AS IS" BASIS,
  14                 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  15                 * See the License for the specific language governing permissions and
  16                 * limitations under the License.
  17                 */
  18
  19                #include "filesystemstore/FileSystemStore.h"
  20                #include "kv_config/kv_config.h"
  21                #include "filesystem/Dir.h"
  22                #include "filesystem/File.h"
  23                #include "blockdevice/BlockDevice.h"
  24                #include "mbed_error.h"
  25                #include <string.h>
  26                #include <stdio.h>
  27                #include <stdlib.h>
  28
  29                #include "mbed-trace/mbed_trace.h"
  30                #define TRACE_GROUP "FSST"
  31
  32                #define FSST_REVISION 1
```

```
33                  #define FSST_MAGIC 0x46535354 // "FSST" hex 'magic' signature
34
35                  #define FSST_DEFAULT_FOLDER_PATH "kvstore" //default FileSystemStore folder path on fs
36
37                  // Only write once flag is supported, other two are kept in storage but ignored
38                  static const uint32_t supported_flags = mbed::KVStore::WRITE_ONCE_FLAG | mbed::KVStore::REQUIRE_CONFIDENTIALITY_FLAG |
39                                                          mbed::KVStore::REQUIRE_REPLAY_PROTECTION_FLAG;
40
41                  using namespace mbed;
42
43                  namespace {
44
45                  // incremental set handle
46                  typedef struct {
47                      char *key;
48                      uint32_t create_flags;
49                      size_t data_size;
50                      File *file_handle;
51                  } inc_set_handle_t;
52
53                  // iterator handle
54                  typedef struct {
55                      void *dir_handle;
56                      char *prefix;
57                  } key_iterator_handle_t;
58
59                  } // anonymous namespace
60
61                  // Local Functions
62                  static char *string_ndup(const char *src, size_t size);
63
64
65                  // Class Functions
66              4   FileSystemStore::FileSystemStore(FileSystem *fs) : _fs(fs),
67                      _is_initialized(false), _cfg_fs_path(NULL), _cfg_fs_path_size(0),
68              4       _full_path_key(NULL), _cur_inc_data_size(0), _cur_inc_set_handle(NULL)
69                  {
70
71              4   }
72
73            106   int FileSystemStore::init()
```

```
74
75      106 {   int status = MBED_SUCCESS;
76
77      106     _mutex.lock();
78    ✓✗ 106     const char *temp_path = get_filesystemstore_folder_path();
79    ✗✓ 106     if (temp_path == NULL) {
80                   _cfg_fs_path_size = strlen(FSST_DEFAULT_FOLDER_PATH);
81                   _cfg_fs_path = string_ndup(FSST_DEFAULT_FOLDER_PATH, _cfg_fs_path_size);
82              } else {
83      106         _cfg_fs_path_size = strlen(temp_path);
84    ✓✗ 106         _cfg_fs_path = string_ndup(temp_path, _cfg_fs_path_size);
85              }
86
87    ✓✗ 106     _full_path_key = new char[_cfg_fs_path_size + KVStore::MAX_KEY_SIZE + 1];
88      106     memset(_full_path_key, 0, (_cfg_fs_path_size + KVStore::MAX_KEY_SIZE + 1));
89      106     strncpy(_full_path_key, _cfg_fs_path, _cfg_fs_path_size);
90      106     _full_path_key[_cfg_fs_path_size] = '/';
91      106     _cur_inc_data_size = 0;
92      106     _cur_inc_set_handle = NULL;
93    ✓✗ 212     Dir kv_dir;
94
95  ✓✗✓✓ 106     if (kv_dir.open(_fs, _cfg_fs_path) != 0) {
96                   tr_info("KV Dir: %s, doesnt exist - creating new.. ", _cfg_fs_path); //TBD verify ERRNO NOEXIST
97  ✓✗✗✓   4         if (_fs->mkdir(_cfg_fs_path,/* which flags ? */0777) != 0) {
98                       tr_error("KV Dir: %s, mkdir failed.. ", _cfg_fs_path); //TBD verify ERRNO NOEXIST
99                       status = MBED_ERROR_FAILED_OPERATION;
100                      goto exit_point;
101                  }
102             } else {
103                 tr_info("KV Dir: %s, exists(verified) - now closing it", _cfg_fs_path);
104   ✓✗ 102        if (kv_dir.close() != 0) {
105                      tr_error("KV Dir: %s, dir_close failed", _cfg_fs_path); //TBD verify ERRNO NOEXIST
106                  }
107             }
108
109     106     _is_initialized = true;
110     106 exit_point:
111
112     106     _mutex.unlock();
113
114     212     return status;
115
```

```
116                 }
117
118         105     int FileSystemStore::deinit()
119                 {
120         105         _mutex.lock();
121         105         _is_initialized = false;
122     ✓✗  105         delete[] _cfg_fs_path;
123     ✓✗  105         delete[] _full_path_key;
124         105         _mutex.unlock();
125         105         return MBED_SUCCESS;
126
127                 }
128
129                 int FileSystemStore::reset()
130                 {
131                     int status = MBED_SUCCESS;
132                     Dir kv_dir;
133                     struct dirent dir_ent;
134
135                     _mutex.lock();
136                     if (false == _is_initialized) {
137                         status = MBED_ERROR_NOT_READY;
138                         goto exit_point;
139                     }
140
141                     kv_dir.open(_fs, _cfg_fs_path);
142
143                     while (kv_dir.read(&dir_ent) != 0) {
144                         if (dir_ent.d_type != DT_REG) {
145                             continue;
146                         }
147                         // Build File's full path name and delete it (even if write-onced)
148                         _build_full_path_key(dir_ent.d_name);
149                         _fs->remove(_full_path_key);
150                     }
151
152                     kv_dir.close();
153
154                 exit_point:
155                     _mutex.unlock();
156                     return status;
157                 }
```

```
158                                                                                 
159          103  int FileSystemStore::set(const char *key, const void *buffer, size_t size, uint32_t create_flags)
160                {
161          103      int status = MBED_SUCCESS;
162                   set_handle_t handle;
163                
164    ✗✓    103      if (false == _is_initialized) {
165                        status = MBED_ERROR_NOT_READY;
166                        goto exit_point;
167                    }
168                
169    ✓✗✗✓  103      if ((!is_valid_key(key)) || ((buffer == NULL) && (size > 0))) {
       ✗✗✗✓
170                        status = MBED_ERROR_INVALID_ARGUMENT;
171                        goto exit_point;
172                    }
173                
174    ✓✗    103      status = set_start(&handle, key, size, create_flags);
175    ✗✓    103      if (status != MBED_SUCCESS) {
176                        tr_error("FSST Set set_start Failed: %d", status);
177                        goto exit_point;
178                    }
179                
180    ✓✗    103      status = set_add_data(handle, buffer, size);
181    ✗✓    103      if (status != MBED_SUCCESS) {
182                        tr_error("FSST Set set_add_data Failed: %d", status);
183                        set_finalize(handle);
184                        goto exit_point;
185                    }
186                
187    ✓✗    103      status = set_finalize(handle);
188    ✓✗    103      if (status != MBED_SUCCESS) {
189                        tr_error("FSST Set set_finalize Failed: %d", status);
190                        goto exit_point;
191                    }
192                
193          206  exit_point:
194                
195          103      return status;
196                }
197                
```

```
198 199  101  int FileSystemStore::get(const char *key, void *buffer, size_t buffer_size, size_t *actual_size, size_t offset)
199      101  {
200      101      int status = MBED_SUCCESS;
201
202   ✓✗ 202      File kv_file;
203      101      size_t kv_file_size = 0;
204      101      size_t value_actual_size = 0;
205
206      101      _mutex.lock();
207
208   ✗✓ 101      if (false == _is_initialized) {
209                    status = MBED_ERROR_NOT_READY;
210                    goto exit_point;
211                }
212
213                key_metadata_t key_metadata;
214
215 ✓✗✗✓ 101      if ((status = _verify_key_file(key, &key_metadata, &kv_file)) != MBED_SUCCESS) {
216                    tr_debug("File Verification failed, status: %d", status);
217                    goto exit_point;
218                }
219
220   ✓✗ 101      kv_file_size = kv_file.size() - key_metadata.metadata_size;
221                // Actual size is the minimum of buffer_size and remainder of data in file (file's data size - offset)
222      101      value_actual_size = buffer_size;
223   ✗✓ 101      if (offset > kv_file_size) {
224                    status = MBED_ERROR_INVALID_SIZE;
225                    goto exit_point;
226   ✓✗ 101      } else if ((kv_file_size - offset) < buffer_size) {
227      101          value_actual_size = kv_file_size - offset;
228                }
229
230 ✗✓✗✗ 101      if ((buffer == NULL) && (value_actual_size > 0)) {
231                    status = MBED_ERROR_INVALID_DATA_DETECTED;
232                    goto exit_point;
233                }
234
235   ✓✗ 101      if (actual_size != NULL) {
236      101          *actual_size = value_actual_size;
237                }
238
239   ✓✗ 101      kv_file.seek(key_metadata.metadata_size + offset, SEEK_SET);
```

```
240                     // Read remainder of data
241   ✓✗ 101            kv_file.read(buffer, value_actual_size);
242
243      101        exit_point:
244   ✗✓✗✗ 101            if ((status == MBED_SUCCESS) ||
245                             (status == MBED_ERROR_INVALID_DATA_DETECTED)) {
246   ✓✗ 101                kv_file.close();
247                     }
248      101            _mutex.unlock();
249
250      202            return status;
251                 }
252
253                 int FileSystemStore::get_info(const char *key, info_t *info)
254                 {
255                     int status = MBED_SUCCESS;
256                     File kv_file;
257
258                     _mutex.lock();
259
260                     if (false == _is_initialized) {
261                         status = MBED_ERROR_NOT_READY;
262                         goto exit_point;
263                     }
264
265                     key_metadata_t key_metadata;
266
267                     if ((status = _verify_key_file(key, &key_metadata, &kv_file)) != MBED_SUCCESS) {
268                         tr_debug("File Verification failed, status: %d", status);
269                         goto exit_point;
270                     }
271
272                     if (info != NULL) {
273                         info->size = kv_file.size() - key_metadata.metadata_size;
274                         info->flags = key_metadata.user_flags;
275                     }
276
277                 exit_point:
278                     if ((status == MBED_SUCCESS) ||
279                             (status == MBED_ERROR_INVALID_DATA_DETECTED)) {
280                         kv_file.close();
281                     }
```

```
282                     _mutex.unlock();
283
284                     return status;
285                 }
286
287         100     int FileSystemStore::remove(const char *key)
288                 {
289     ✓✗  200         File kv_file;
290                     key_metadata_t key_metadata;
291
292         100         _mutex.lock();
293
294         100         int status = MBED_SUCCESS;
295
296     ✗✓  100         if (false == _is_initialized) {
297                         status = MBED_ERROR_NOT_READY;
298                         goto exit_point;
299                     }
300
301                     /* If File Exists and is Valid, then check its Write Once Flag to verify its disabled before removing */
302                     /* If File exists and is not valid, or is Valid and not Write-Onced then remove it */
303   ✓✗✓✗  100         if ((status = _verify_key_file(key, &key_metadata, &kv_file)) == MBED_SUCCESS) {
304     ✗✓  100             if (key_metadata.user_flags & KVStore::WRITE_ONCE_FLAG) {
305                             kv_file.close();
306                             tr_error("File: %s, Exists but write protected", _full_path_key);
307                             status = MBED_ERROR_WRITE_PROTECTED;
308                             goto exit_point;
309                         }
310                     } else if ((status == MBED_ERROR_ITEM_NOT_FOUND) ||
311                                (status == MBED_ERROR_INVALID_ARGUMENT)) {
312                         goto exit_point;
313                     }
314     ✓✗  100         kv_file.close();
315
316   ✓✗✓✗  100         if (0 != _fs->remove(_full_path_key)) {
317                         status =  MBED_ERROR_FAILED_OPERATION;
318                     }
319
320         200     exit_point:
321         100         _mutex.unlock();
322         200         return status;
```

```
323                 }
324
325                 // Incremental set API
326         103     int FileSystemStore::set_start(set_handle_t *handle, const char *key, size_t final_data_size, uint32_t create_flags)
327                 {
328         103         int status = MBED_SUCCESS;
329         103         inc_set_handle_t *set_handle = NULL;
330                     File *kv_file;
331                     key_metadata_t key_metadata;
332         103         int key_len = 0;
333
334     ✗✓  103         if (create_flags & ~supported_flags) {
335                         return MBED_ERROR_INVALID_ARGUMENT;
336                     }
337
338                     // Only a single key file can be incrementaly editted at a time
339         103         _mutex.lock();
340
341   ✓✗✓✗  103         kv_file = new File;
342
343     ✗✓  103         if (handle == NULL) {
344                         status = MBED_ERROR_INVALID_ARGUMENT;
345                         goto exit_point;
346                     }
347
348                     /* If File Exists and is Valid, then check its Write Once Flag to verify its disabled before setting */
349                     /* If File exists and is not valid, or is Valid and not Write-Onced then erase it */
350     ✓✗  103         status = _verify_key_file(key, &key_metadata, kv_file);
351
352     ✗✓  103         if (status == MBED_ERROR_INVALID_ARGUMENT) {
353                         tr_error("File Verification failed, status: %d", status);
354                         goto exit_point;
355                     }
356
357     ✗✓  103         if (status == MBED_SUCCESS) {
358                         if (key_metadata.user_flags & KVStore::WRITE_ONCE_FLAG) {
359                             kv_file->close();
360                             status = MBED_ERROR_WRITE_PROTECTED;
361                             goto exit_point;
362                         }
363                     }
364
```

```
365                     /* For Success (not write_once) and for corrupted data close file before recreating it as a new file */
366   ✗✓  103           if (status != MBED_ERROR_ITEM_NOT_FOUND) {
367                         kv_file->close();
368                     }
369
370   ✓✗✗✓ 103          if ((status = kv_file->open(_fs, _full_path_key, O_WRONLY | O_CREAT | O_TRUNC)) != MBED_SUCCESS) {
371                         tr_info("set_start failed to open: %s, for writing, err: %d", _full_path_key, status);
372                         status = MBED_ERROR_FAILED_OPERATION ;
373                         goto exit_point;
374                     }
375             103     _cur_inc_data_size = 0;
376
377   ✓✗        103     set_handle = new inc_set_handle_t;
378             103     set_handle->create_flags = create_flags;
379             103     set_handle->data_size = final_data_size;
380             103     set_handle->file_handle = kv_file;
381             103     key_len = strlen(key);
382   ✓✗        103     set_handle->key = string_ndup(key, key_len);
383             103     *handle = (set_handle_t)set_handle;
384             103     _cur_inc_set_handle = *handle;
385
386             103     key_metadata.magic = FSST_MAGIC;
387             103     key_metadata.metadata_size = sizeof(key_metadata_t);
388             103     key_metadata.revision = FSST_REVISION;
389             103     key_metadata.user_flags = create_flags;
390   ✓✗        103     kv_file->write(&key_metadata, sizeof(key_metadata_t));
391
392             103 exit_point:
393   ✗✓        103     if (status != MBED_SUCCESS) {
394                         delete kv_file;
395                         _mutex.unlock();
396                     }
397             103     return status;
398                 }
399
400             103 int FileSystemStore::set_add_data(set_handle_t handle, const void *value_data, size_t data_size)
401                 {
402             103     int status = MBED_SUCCESS;
403             103     size_t added_data = 0;
404             103     inc_set_handle_t *set_handle = (inc_set_handle_t *)handle;
405                     File *kv_file;
```

```
406
407 ✗✓✗✗ ✓✗✗✓ 103     if (((value_data == NULL) && (data_size > 0)) || (handle == NULL) || (handle != _cur_inc_set_handle)) {
408                         status = MBED_ERROR_INVALID_ARGUMENT;
409                         goto exit_point;
410                     }
411
412                     // Single key incrementally edited, can be edited from multiple threads - lock to protect
413                 103 _inc_data_add_mutex.lock();
414               ✗✓ 103 if ((_cur_inc_data_size + data_size) > set_handle->data_size) {
415                         tr_warning("Added Data(%d) will exceed set_start final size(%d) - not adding data to file: %s",
416                                    _cur_inc_data_size + data_size, set_handle->data_size, _full_path_key);
417                         status = MBED_ERROR_INVALID_SIZE;
418                         goto exit_point;
419                     }
420
421                 103 kv_file = set_handle->file_handle;
422
423                 103 added_data = kv_file->write(value_data, data_size);
424               ✗✓ 103 if (added_data != data_size) {
425                         status = MBED_ERROR_FAILED_OPERATION ;
426                     }
427                 103 _cur_inc_data_size += added_data;
428
429                 103 exit_point:
430               ✓✗ 103     if (status != MBED_ERROR_INVALID_ARGUMENT) {
431                 103         _inc_data_add_mutex.unlock();
432                     }
433
434                 103     return status;
435                 }
436
437                 103 int FileSystemStore::set_finalize(set_handle_t handle)
438                 {
439                 103     int status = MBED_SUCCESS;
440                 103     inc_set_handle_t *set_handle = NULL;
441
442 ✓✗✗✓            103     if ((handle == NULL) || (handle != _cur_inc_set_handle)) {
443                         status =  MBED_ERROR_INVALID_ARGUMENT;
444                         goto exit_point;
445                     }
446
```

```
447          103      set_handle = (inc_set_handle_t *)handle;
448
449     ✗✓   103      if (set_handle->key == NULL) {
450                          status = MBED_ERROR_INVALID_DATA_DETECTED;
451                  } else {
452     ✗✓   103          if (_cur_inc_data_size != set_handle->data_size) {
453                              tr_error("Accumulated Data (%d) size doesn't match set_start final size (%d) - file: %s", _cur_inc_data_size,
454                                       set_handle->data_size, _full_path_key);
455                              status = MBED_ERROR_INVALID_SIZE;
456                              _fs->remove(_full_path_key);
457                          }
458     ✓✗   103          delete[] set_handle->key;
459                  }
460
461          103      set_handle->file_handle->close();
462     ✓✗   103      delete set_handle->file_handle;
463          103      delete set_handle;
464          103      _cur_inc_data_size = 0;
465          103      _cur_inc_set_handle = NULL;
466
467          103  exit_point:
468     ✓✗   103      if (status != MBED_ERROR_INVALID_ARGUMENT) {
469          103          _mutex.unlock();
470                  }
471
472          103      return status;
473                  }
474
475            1  int FileSystemStore::iterator_open(iterator_t *it, const char *prefix)
476                  {
477            1      int status = MBED_SUCCESS;
478            1      Dir *kv_dir = NULL;
479            1      key_iterator_handle_t *key_it = NULL;
480
481     ✗✓     1      if (it == NULL) {
482                          return MBED_ERROR_INVALID_ARGUMENT;
483                  }
484
485            1      _mutex.lock();
486     ✗✓     1      if (false == _is_initialized) {
487                          status = MBED_ERROR_NOT_READY;
488                          goto exit_point;
```

```
489          }
490    1     key_it = new key_iterator_handle_t;
491    1     key_it->dir_handle = NULL;
492    1     key_it->prefix = NULL;
493 ✓✗ 1     if (prefix != NULL) {
494    1         key_it->prefix = string_ndup(prefix, KVStore::MAX_KEY_SIZE);
495          }
496
497 ✓✗ 1     kv_dir = new Dir;
498 ✗✓ 1     if (kv_dir->open(_fs, _cfg_fs_path) != 0) {
499              tr_error("KV Dir: %s, doesnt exist", _cfg_fs_path); //TBD verify ERRNO NOEXIST
500              delete kv_dir;
501              if (key_it->prefix != NULL) {
502                  delete[] key_it->prefix;
503              }
504              delete key_it;
505              status = MBED_ERROR_ITEM_NOT_FOUND;
506              goto exit_point;
507          }
508
509    1     key_it->dir_handle = kv_dir;
510
511    1     *it = (iterator_t)key_it;
512
513    1 exit_point:
514    1     _mutex.unlock();
515
516    1     return status;
517      }
518
519    3 int FileSystemStore::iterator_next(iterator_t it, char *key, size_t key_size)
520      {
521          Dir *kv_dir;
522          struct dirent kv_dir_ent;
523    3     int status = MBED_ERROR_ITEM_NOT_FOUND;
524          key_iterator_handle_t *key_it;
525    3     size_t key_name_size = KVStore::MAX_KEY_SIZE;
526 ✓✗ 3     if (key_size < key_name_size) {
527    3         key_name_size = key_size;
528          }
529
```

```
530          3      _mutex.lock();
531    ✗✓    3      if (false == _is_initialized) {
532                       status = MBED_ERROR_NOT_READY;
533                       goto exit_point;
534                   }
535
536          3      key_it = (key_iterator_handle_t *)it;
537
538    ✓✗✗✓  3      if ((key_it->prefix != NULL) && (key_name_size < strlen(key_it->prefix))) {
539                       status = MBED_ERROR_INVALID_SIZE;
540                       goto exit_point;
541                   }
542
543          3      kv_dir = (Dir *)key_it->dir_handle;
544
545    ✓✗✓✓  7      while (kv_dir->read(&kv_dir_ent) != 0) {
546      ✓✓  4          if (kv_dir_ent.d_type != DT_REG) {
547          2              continue;
548                       }
549
550    ✓✗✓✗  4          if ((key_it->prefix == NULL) ||
551          2                  (strncmp(kv_dir_ent.d_name, key_it->prefix, strlen(key_it->prefix)) == 0)) {
552      ✗✓  2              if (key_name_size < strlen(kv_dir_ent.d_name)) {
553                             status = MBED_ERROR_INVALID_SIZE;
554                             break;
555                         }
556          2              strncpy(key, kv_dir_ent.d_name, key_name_size);
557          2              key[key_name_size - 1] = '\0';
558          2              status = MBED_SUCCESS;
559          2              break;
560                       }
561                   }
562
563          1  exit_point:
564          3      _mutex.unlock();
565          3      return status;
566              }
567
568          1  int FileSystemStore::iterator_close(iterator_t it)
569              {
570          1      int status = MBED_SUCCESS;
571          1      key_iterator_handle_t *key_it = (key_iterator_handle_t *)it;
```

```
572
573   1      _mutex.lock();
574 ✗✓  1      if (key_it == NULL) {
575            status = MBED_ERROR_INVALID_ARGUMENT;
576            goto exit_point;
577          }
578
579 ✓✗  1      if (key_it->prefix != NULL) {
580 ✓✗  1          delete[] key_it->prefix;
581          }
582
583 ✓✗  1      if (key_it->dir_handle != NULL) {
584    1          ((Dir *)(key_it->dir_handle))->close();
585 ✓✗  1          delete ((Dir *)(key_it->dir_handle));
586          }
587    1     delete key_it;
588
589    1  exit_point:
590    1      _mutex.unlock();
591    1      return status;
592        }
593
594  304   int FileSystemStore::_verify_key_file(const char *key, key_metadata_t *key_metadata, File *kv_file)
595        {
596  304       int status = MBED_SUCCESS;
597
598 ✗✓ 304       if (!is_valid_key(key)) {
599            status = MBED_ERROR_INVALID_ARGUMENT;
600            goto exit_point;
601          }
602
603  304       _build_full_path_key(key);
604
605 ✓✓ 304       if (0 != kv_file->open(_fs, _full_path_key, O_RDONLY)) {
606  103           status = MBED_ERROR_ITEM_NOT_FOUND;
607  103           goto exit_point;
608          }
609
610          //Read Metadata
611  201       kv_file->read(key_metadata, sizeof(key_metadata_t));
612
```

613	✓✗✓✗	402	`    if ((key_metadata->magic != FSST_MAGIC) ||`
614		201	`            (key_metadata->revision > FSST_REVISION)) {`
615			`        status = MBED_ERROR_INVALID_DATA_DETECTED;`
616			`        goto exit_point;`
617			`    }`
618			
619		505	`exit_point:`
620		304	`    return status;`
621			`}`
622			
623		304	`int FileSystemStore::_build_full_path_key(const char *key_src)`
624			`{`
625		304	`    strncpy(&_full_path_key[_cfg_fs_path_size + 1/* for path's \ */], key_src, KVStore::MAX_KEY_SIZE);`
626		304	`    _full_path_key[(_cfg_fs_path_size + KVStore::MAX_KEY_SIZE)] = '\0';`
627		304	`    return 0;`
628			`}`
629			
630			`// Local Functions`
631		210	`static char *string_ndup(const char *src, size_t size)`
632			`{`
633		210	`    char *string_copy = new char[size + 1];`
634		210	`    strncpy(string_copy, src, size);`
635		210	`    string_copy[size] = '\0';`
636		210	`    return string_copy;`
637			`}`