GCC Code Coverage Report

			Exec	Total	Coverage
Directory:	./				
File:	storage/blockdevice/source/MBRBlockDevice.cpp	Lines:	176	204	86.3 %
Date:	2021-05-06 12:39:05	Branches:	83	136	61.0 %

```
   Line Branch    Exec  Source
      1                 /* mbed Microcontroller Library
      2                  * Copyright (c) 2017 ARM Limited
      3                  * SPDX-License-Identifier: Apache-2.0
      4                  *
      5                  * Licensed under the Apache License, Version 2.0 (the "License");
      6                  * you may not use this file except in compliance with the License.
      7                  * You may obtain a copy of the License at
      8                  *
      9                  *     http://www.apache.org/licenses/LICENSE-2.0
     10                  *
     11                  * Unless required by applicable law or agreed to in writing, software
     12                  * distributed under the License is distributed on an "AS IS" BASIS,
     13                  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
     14                  * See the License for the specific language governing permissions and
     15                  * limitations under the License.
     16                  */
     17
     18                 #include "blockdevice/MBRBlockDevice.h"
     19                 #include "platform/mbed_atomic.h"
     20                 #include "platform/mbed_toolchain.h"
     21                 #include "platform/mbed_assert.h"
     22                 #include <algorithm>
     23                 #include <string.h>
     24
     25                 namespace mbed {
     26
     27                 // On disk structures, all entries are little endian
     28                 MBED_PACKED(struct) mbr_entry {
     29                     uint8_t status;
     30                     uint8_t chs_start[3];
```

```
31          uint8_t type;
32          uint8_t chs_stop[3];
33          uint32_t lba_offset;
34          uint32_t lba_size;
35      };
36
37      MBED_PACKED(struct) mbr_table {
38          struct mbr_entry entries[4];
39          uint8_t signature[2];
40      };
41
42      // Little-endian conversion, should compile to noop
43      // if system is little-endian
44   36 static inline uint32_t tole32(uint32_t a)
45      {
46          union {
47              uint32_t u32;
48              uint8_t u8[4];
49          } w;
50
51   36     w.u8[0] = a >>  0;
52   36     w.u8[1] = a >>  8;
53   36     w.u8[2] = a >> 16;
54   36     w.u8[3] = a >> 24;
55
56   36     return w.u32;
57      }
58
59   28 static inline uint32_t fromle32(uint32_t a)
60      {
61   28     return tole32(a);
62      }
63
64    8 static void tochs(uint32_t lba, uint8_t chs[3])
65      {
66    8     uint32_t sector = std::min<uint32_t>(lba, 0xfffffd) + 1;
67    8     chs[0] = (sector >> 6) & 0xff;
68    8     chs[1] = ((sector >> 0) & 0x3f) | ((sector >> 16) & 0xc0);
69    8     chs[2] = (sector >> 14) & 0xff;
70    8 }
```

```
71
72
73          // Partition after address are turned into absolute
74          // addresses, assumes bd is initialized
75      4   static int partition_absolute(
76              BlockDevice *bd, int part, uint8_t type,
77              bd_size_t offset, bd_size_t size)
78          {
79              // Allocate smallest buffer necessary to write MBR
80  ✓✗  4       uint32_t buffer_size = std::max<uint32_t>(bd->get_program_size(), sizeof(struct mbr_table));
81
82              // Prevent alignment issues
83  ✗✓  4       if (buffer_size % bd->get_program_size() != 0) {
84                  buffer_size += bd->get_program_size() - (buffer_size % bd->get_program_size());
85              }
86
87      4       uint8_t *buffer = new uint8_t[buffer_size];
88
89              // Check for existing MBR
90      4       int err = bd->read(buffer, 512 - buffer_size, buffer_size);
91  ✗✓  4       if (err) {
92                  delete[] buffer;
93                  return err;
94              }
95
96      4       uint32_t table_start_offset = buffer_size - sizeof(struct mbr_table);
97      4       struct mbr_table *table = reinterpret_cast<struct mbr_table *>(
98      4                                     &buffer[table_start_offset]);
99  ✓✗✗✓ 4       if (table->signature[0] != 0x55 || table->signature[1] != 0xaa) {
100                 // Setup default values for MBR
101                 table->signature[0] = 0x55;
102                 table->signature[1] = 0xaa;
103                 memset(table->entries, 0, sizeof(table->entries));
104             }
105
106             // For Windows-formatted SD card, it is not partitioned (no MBR), but its PBR has the
107             // same boot signature (0xaa55) as MBR. We would easily mis-recognize this SD card has valid
108             // partitions if we only check partition type. We add check by only accepting 0x00 (inactive)
109             // /0x80 (active) for valid partition status.
110 ✓✓  20      for (int i = 1; i <= 4; i++) {
```

```
111  ✗✓✗✗  16          if (table->entries[i - 1].status != 0x00 &&
112                             table->entries[i - 1].status != 0x80) {
113                         memset(table->entries, 0, sizeof(table->entries));
114                         break;
115                     }
116                 }
117
118                 // Setup new partition
119  ✓✗✗✓   4      MBED_ASSERT(part >= 1 && part <= 4);
120         4      table->entries[part - 1].status = 0x00; // inactive (not bootable)
121         4      table->entries[part - 1].type = type;
122
123                 // lba dimensions
124    ✗✓   4      MBED_ASSERT(bd->is_valid_erase(offset, size));
125    ✓✗   4      uint32_t sector = std::max<uint32_t>(bd->get_erase_size(), 512);
126         4      uint32_t lba_offset = offset / sector;
127         4      uint32_t lba_size = size / sector;
128         4      table->entries[part - 1].lba_offset = tole32(lba_offset);
129         4      table->entries[part - 1].lba_size = tole32(lba_size);
130
131                 // chs dimensions
132         4      tochs(lba_offset,            table->entries[part - 1].chs_start);
133         4      tochs(lba_offset + lba_size - 1, table->entries[part - 1].chs_stop);
134
135                 // Check that we don't overlap other entries
136    ✓✓   20     for (int i = 1; i <= 4; i++) {
137  ✓✓✓✓   16         if (i != part && table->entries[i - 1].type != 0x00) {
138          6             uint32_t neighbor_lba_offset = fromle32(table->entries[i - 1].lba_offset);
139          6             uint32_t neighbor_lba_size = fromle32(table->entries[i - 1].lba_size);
140  ✗✓✗✗   6             MBED_ASSERT(
141                             (lba_offset >= neighbor_lba_offset + neighbor_lba_size) ||
142                             (lba_offset + lba_size <= neighbor_lba_offset));
143                         (void)neighbor_lba_offset;
144                         (void)neighbor_lba_size;
145                     }
146                 }
147
148                 // As the erase operation may do nothing, erase remainder of the buffer, to eradicate
149                 // any remaining programmed data (such as previously programmed file systems).
150    ✓✗   4      if (table_start_offset > 0) {
```

```
151        4          memset(buffer, 0xFF, table_start_offset);
152                   }
153  ✗✓    4          if (table_start_offset + sizeof(struct mbr_table) < buffer_size) {
154                       memset(buffer + table_start_offset + sizeof(struct mbr_table), 0xFF,
155                              buffer_size - (table_start_offset + sizeof(struct mbr_table)));
156                   }
157
158                   // Write out MBR
159        4          err = bd->erase(0, bd->get_erase_size());
160  ✗✓    4          if (err) {
161                       delete[] buffer;
162                       return err;
163                   }
164
165        4          err = bd->program(buffer, 512 - buffer_size, buffer_size);
166  ✓✗    4          delete[] buffer;
167        4          return err;
168              }
169
170        1     int MBRBlockDevice::partition(BlockDevice *bd, int part, uint8_t type, bd_addr_t start)
171              {
172        1          int err = bd->init();
173  ✗✓    1          if (err) {
174                       return err;
175                   }
176
177                   // Calculate dimensions
178  ✓✗    1          bd_size_t size = ((int64_t)start < 0) ? -start : start;
179        1          bd_size_t offset = bd->size();
180
181  ✗✓    1          if (size < 512) {
182                       size += std::max<uint32_t>(bd->get_erase_size(), 512);
183                   }
184
185        1          offset -= size;
186
187        1          err = partition_absolute(bd, part, type, offset, size);
188  ✗✓    1          if (err) {
189                       return err;
190                   }
```

```
        
        1      err = bd->deinit();
  ✗✓    1      if (err) {
                   return err;
               }

        1      return 0;
           }

        3  int MBRBlockDevice::partition(BlockDevice *bd, int part, uint8_t type,
                                         bd_addr_t start, bd_addr_t stop)
           {
        3      int err = bd->init();
  ✗✓    3      if (err) {
                   return err;
               }

               // Calculate dimensions
  ✗✓    3      bd_size_t offset = ((int64_t)start < 0) ? -start : start;
  ✗✓    3      bd_size_t size = ((int64_t)stop < 0) ? -stop : stop;

  ✓✓    3      if (offset < 512) {
  ✓✗    1          offset += std::max<uint32_t>(bd->get_erase_size(), 512);
               }

        3      size -= offset;

        3      err = partition_absolute(bd, part, type, offset, size);
  ✗✓    3      if (err) {
                   return err;
               }

        3      err = bd->deinit();
  ✗✓    3      if (err) {
                   return err;
               }

        3      return 0;
           }

```

```
231          11  MBRBlockDevice::MBRBlockDevice(BlockDevice *bd, int part)
232          11      : _bd(bd), _offset(0), _size(0), _type(0), _part(part), _init_ref_count(0), _is_initialized(false)
233              {
234  ✓✗✗✓    11      MBED_ASSERT(_part >= 1 && _part <= 4);
235          11  }
236
237          11  int MBRBlockDevice::init()
238              {
239                  uint32_t buffer_size;
240          11      uint8_t *buffer = 0;
241                  struct mbr_table *table;
242                  bd_size_t sector;
243                  int err;
244
245          11      uint32_t val = core_util_atomic_incr_u32(&_init_ref_count, 1);
246
247    ✗✓    11      if (val != 1) {
248                      return BD_ERROR_OK;
249                  }
250
251          11      err = _bd->init();
252    ✗✓    11      if (err) {
253                      goto fail;
254                  }
255
256                  // Allocate smallest buffer necessary to write MBR
257    ✓✗    11      buffer_size = std::max<uint32_t>(_bd->get_read_size(), sizeof(struct mbr_table));
258          11      buffer = new uint8_t[buffer_size];
259
260          11      err = _bd->read(buffer, 512 - buffer_size, buffer_size);
261    ✗✓    11      if (err) {
262                      goto fail;
263                  }
264
265                  // Check for valid table
266          11      table = reinterpret_cast<struct mbr_table *>(&buffer[buffer_size - sizeof(struct mbr_table)]);
267  ✓✓✗✓    11      if (table->signature[0] != 0x55 || table->signature[1] != 0xaa) {
268           1          err = BD_ERROR_INVALID_MBR;
269           1          goto fail;
270                  }
```

```
271                                
272                                     // Check for valid partition status
273                                     // Same reason as in partition_absolute regarding Windows-formatted SD card
274  ✓✓✓✗      11                     if (table->entries[_part - 1].status != 0x00 &&
275                1                         table->entries[_part - 1].status != 0x80) {
276                1                     err = BD_ERROR_INVALID_PARTITION;
277                1                     goto fail;
278                                     }
279                                
280                                     // Check for valid entry
281                                     // 0x00 = no entry
282                                     // 0x05, 0x0f = extended partitions, currently not supported
283  ✓✗✓✓      18                     if ((table->entries[_part - 1].type == 0x00 ||
284    ✗✓      17                         table->entries[_part - 1].type == 0x05 ||
285                8                         table->entries[_part - 1].type == 0x0f)) {
286                1                     err = BD_ERROR_INVALID_PARTITION;
287                1                     goto fail;
288                                     }
289                                
290                                     // Get partition attributes
291     ✓✗         8                 sector = std::max<uint32_t>(_bd->get_erase_size(), 512);
292                8                 _type = table->entries[_part - 1].type;
293                8                 _offset = fromle32(table->entries[_part - 1].lba_offset) * sector;
294                8                 _size   = fromle32(table->entries[_part - 1].lba_size)   * sector;
295                                
296                                     // Check that block addresses are valid
297     ✗✓         8                 if (!_bd->is_valid_erase(_offset, _size)) {
298                                         err = BD_ERROR_INVALID_PARTITION;
299                                         goto fail;
300                                     }
301                                
302                8                 _is_initialized = true;
303     ✓✗         8                 delete[] buffer;
304                8                 return BD_ERROR_OK;
305                                
306                3             fail:
307     ✓✗         3                 delete[] buffer;
308                3                 _is_initialized = false;
309                3                 _init_ref_count = 0;
310                3                 return err;
```

```
311                 }
312
313         9       int MBRBlockDevice::deinit()
314                 {
315     ✓✓  9           if (!_is_initialized) {
316         1               return BD_ERROR_OK;
317                     }
318
319         8           uint32_t val = core_util_atomic_decr_u32(&_init_ref_count, 1);
320
321     ✗✓  8           if (val) {
322                         return BD_ERROR_OK;
323                     }
324
325         8           _is_initialized = false;
326         8           return _bd->deinit();
327                 }
328
329         1       int MBRBlockDevice::sync()
330                 {
331     ✓✗  1           if (!_is_initialized) {
332         1               return BD_ERROR_DEVICE_ERROR;
333                     }
334
335                     return _bd->sync();
336                 }
337
338         3       int MBRBlockDevice::read(void *b, bd_addr_t addr, bd_size_t size)
339                 {
340     ✓✓  3           if (!_is_initialized) {
341         1               return BD_ERROR_DEVICE_ERROR;
342                     }
343
344     ✓✓  2           if (!is_valid_read(addr, size)) {
345         1               return BD_ERROR_DEVICE_ERROR;
346                     }
347
348         1           return _bd->read(b, addr + _offset, size);
349                 }
350
```

```
351        4  int MBRBlockDevice::program(const void *b, bd_addr_t addr, bd_size_t size)
352           {
353   ✓✓   4      if (!_is_initialized) {
354        1          return BD_ERROR_DEVICE_ERROR;
355           }
356
357   ✓✓   3      if (!is_valid_program(addr, size)) {
358        2          return BD_ERROR_DEVICE_ERROR;
359           }
360
361        1      return _bd->program(b, addr + _offset, size);
362           }
363
364        3  int MBRBlockDevice::erase(bd_addr_t addr, bd_size_t size)
365           {
366   ✓✓   3      if (!_is_initialized) {
367        1          return BD_ERROR_DEVICE_ERROR;
368           }
369
370   ✓✓   2      if (!is_valid_erase(addr, size)) {
371        1          return BD_ERROR_DEVICE_ERROR;
372           }
373
374        1      return _bd->erase(addr + _offset, size);
375           }
376
377        6  bd_size_t MBRBlockDevice::get_read_size() const
378           {
379   ✓✓   6      if (!_is_initialized) {
380        1          return 0;
381           }
382
383        5      return _bd->get_read_size();
384           }
385
386        8  bd_size_t MBRBlockDevice::get_program_size() const
387           {
388   ✓✓   8      if (!_is_initialized) {
389        1          return 0;
390           }
```

```
391
392      7      return _bd->get_program_size();
393             }
394
395      2      bd_size_t MBRBlockDevice::get_erase_size() const
396             {
397   ✓✓ 2          if (!_is_initialized) {
398      1          return 0;
399             }
400
401      1      return _bd->get_erase_size();
402             }
403
404      6      bd_size_t MBRBlockDevice::get_erase_size(bd_addr_t addr) const
405             {
406   ✓✓ 6          if (!_is_initialized) {
407      1          return 0;
408             }
409
410      5      return _bd->get_erase_size(_offset + addr);
411             }
412
413      2      int MBRBlockDevice::get_erase_value() const
414             {
415   ✓✓ 2          if (!_is_initialized) {
416      1          return BD_ERROR_DEVICE_ERROR;
417             }
418
419      1      return _bd->get_erase_value();
420             }
421
422      5      bd_size_t MBRBlockDevice::size() const
423             {
424      5      return _size;
425             }
426
427      1      bd_size_t MBRBlockDevice::get_partition_start() const
428             {
429      1      return _offset;
430             }
```

```
431                                
432          1  bd_size_t MBRBlockDevice::get_partition_stop() const
433             {
434          1      return _offset + _size;
435             }
436                                
437          1  uint8_t MBRBlockDevice::get_partition_type() const
438             {
439          1      return _type;
440             }
441                                
442          1  int MBRBlockDevice::get_partition_number() const
443             {
444          1      return _part;
445             }
446                                
447          1  const char *MBRBlockDevice::get_type() const
448             {
449    ✓✗    1      if (_bd != NULL) {
450          1          return _bd->get_type();
451             }
452                                
453                    return NULL;
454             }
455                                
456             } // namespace mbed
```

Generated by: GCOVR (Version 4.2)