tdrozdovsky Fixed warnings History

1 contributor

346 lines (286 sloc) | 8.24 KB ...

```
1  // SPDX-License-Identifier: BSD-2-Clause
2  /*
3   * Copyright (c) 2014, STMicroelectronics International N.V.
4   * All rights reserved.
5   *
6   * Redistribution and use in source and binary forms, with or without
7   * modification, are permitted provided that the following conditions are met:
8   *
9   * 1. Redistributions of source code must retain the above copyright notice,
10  * this list of conditions and the following disclaimer.
11  *
12  * 2. Redistributions in binary form must reproduce the above copyright notice,
13  * this list of conditions and the following disclaimer in the documentation
14  * and/or other materials provided with the distribution.
15  *
16  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
17  * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
18  * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
19  * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
20  * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
21  * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
22  * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
23  * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
24  * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
25  * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
26  * POSSIBILITY OF SUCH DAMAGE.
27  */
28 #include <stdlib.h>
29 #include <string.h>
```

```
30
31  #include <tee_api.h>
32  #include <utee_syscalls.h>
33  #include <user_ta_header.h>
34  #include "tee_user_mem.h"
35  //#include "tee_api_private.h"
36  #include "utee_types.h"
37
38  static const void *tee_api_instance_data;
39
40  /* System API - Internal Client API */
41
42  void __utee_from_param(struct utee_params *up, uint32_t param_types,
43                         const TEE_Param params[TEE_NUM_PARAMS])
44  {
45          size_t n;
46
47          up->types = param_types;
48          for (n = 0; n < TEE_NUM_PARAMS; n++) {
49                  switch (TEE_PARAM_TYPE_GET(param_types, n)) {
50                  case TEE_PARAM_TYPE_VALUE_INPUT:
51                  case TEE_PARAM_TYPE_VALUE_OUTPUT:
52                  case TEE_PARAM_TYPE_VALUE_INOUT:
53                          up->vals[n * 2] = params[n].value.a;
54                          up->vals[n * 2 + 1] = params[n].value.b;
55                          break;
56                  case TEE_PARAM_TYPE_MEMREF_INPUT:
57                  case TEE_PARAM_TYPE_MEMREF_OUTPUT:
58                  case TEE_PARAM_TYPE_MEMREF_INOUT:
59                          up->vals[n * 2] = (uintptr_t)params[n].memref.buffer;
60                          up->vals[n * 2 + 1] = params[n].memref.size;
61                          break;
62                  default:
63                          up->vals[n * 2] = 0;
64                          up->vals[n * 2 + 1] = 0;
65                          break;
66                  }
67          }
68  }
69
70  void __utee_to_param(TEE_Param params[TEE_NUM_PARAMS],
71                         uint32_t *param_types, const struct utee_params *up)
72  {
73          size_t n;
74          uint32_t types = up->types;
75
76          for (n = 0; n < TEE_NUM_PARAMS; n++) {
77                  uintptr_t a = up->vals[n * 2];
78                  uintptr_t b = up->vals[n * 2 + 1];
```

```
79
80                  switch (TEE_PARAM_TYPE_GET(types, n)) {
81                  case TEE_PARAM_TYPE_VALUE_INPUT:
82                  case TEE_PARAM_TYPE_VALUE_OUTPUT:
83                  case TEE_PARAM_TYPE_VALUE_INOUT:
84                          params[n].value.a = a;
85                          params[n].value.b = b;
86                          break;
87                  case TEE_PARAM_TYPE_MEMREF_INPUT:
88                  case TEE_PARAM_TYPE_MEMREF_OUTPUT:
89                  case TEE_PARAM_TYPE_MEMREF_INOUT:
90                          params[n].memref.buffer = (void *)a;
91                          params[n].memref.size = b;
92                          break;
93                  default:
94                          break;
95                  }
96          }
97
98          if (param_types)
99                  *param_types = types;
100 }
101
102 TEE_Result TEE_OpenTASession(const TEE_UUID *destination,
103                              uint32_t cancellationRequestTimeout,
104                              uint32_t paramTypes,
105                              TEE_Param params[TEE_NUM_PARAMS],
106                              TEE_TASessionHandle *session,
107                              uint32_t *returnOrigin)
108 {
109         TEE_Result res;
110         struct utee_params up;
111         uint32_t s;
112
113         __utee_from_param(&up, paramTypes, params);
114         res = utee_open_ta_session(destination, cancellationRequestTimeout,
115                                    &up, &s, returnOrigin);
116         __utee_to_param(params, NULL, &up);
117         /*
118          * Specification says that *session must hold TEE_HANDLE_NULL is
119          * TEE_SUCCESS isn't returned. Set it here explicitly in case
120          * the syscall fails before out parameters has been updated.
121          */
122         if (res != TEE_SUCCESS)
123                 s = TEE_HANDLE_NULL;
124
125         *session = (TEE_TASessionHandle)(uintptr_t)s;
126         return res;
127 }
```

```
128
129 void TEE_CloseTASession(TEE_TASessionHandle session)
130 {
131         if (session != TEE_HANDLE_NULL) {
132                 TEE_Result res = utee_close_ta_session((uintptr_t)session);
133
134                 if (res != TEE_SUCCESS)
135                         TEE_Panic(res);
136         }
137 }
138
139 TEE_Result TEE_InvokeTACommand(TEE_TASessionHandle session,
140                                uint32_t cancellationRequestTimeout,
141                                uint32_t commandID, uint32_t paramTypes,
142                                TEE_Param params[TEE_NUM_PARAMS],
143                                uint32_t *returnOrigin)
144 {
145         TEE_Result res;
146         uint32_t ret_origin;
147         struct utee_params up;
148
149         __utee_from_param(&up, paramTypes, params);
150         res = utee_invoke_ta_command((uintptr_t)session,
151                                      cancellationRequestTimeout,
152                                      commandID, &up, &ret_origin);
153         __utee_to_param(params, NULL, &up);
154
155         if (returnOrigin != NULL)
156                 *returnOrigin = ret_origin;
157
158         if (ret_origin == TEE_ORIGIN_TRUSTED_APP)
159                 return res;
160
161         if (res != TEE_SUCCESS &&
162             res != TEE_ERROR_OUT_OF_MEMORY &&
163             res != TEE_ERROR_TARGET_DEAD)
164                 TEE_Panic(res);
165
166         return res;
167 }
168
169 /* System API - Cancellations */
170
171 bool TEE_GetCancellationFlag(void)
172 {
173         uint32_t c;
174         TEE_Result res = utee_get_cancellation_flag(&c);
175
176         if (res != TEE_SUCCESS)
```

```
177                 c = 0;
178         return !!c;
179 }
180
181 bool TEE_UnmaskCancellation(void)
182 {
183         uint32_t old_mask;
184         TEE_Result res = utee_unmask_cancellation(&old_mask);
185
186         if (res != TEE_SUCCESS)
187                 TEE_Panic(res);
188         return !!old_mask;
189 }
190
191 bool TEE_MaskCancellation(void)
192 {
193         uint32_t old_mask;
194         TEE_Result res = utee_mask_cancellation(&old_mask);
195
196         if (res != TEE_SUCCESS)
197                 TEE_Panic(res);
198         return !!old_mask;
199 }
200
201 /* System API - Memory Management */
202
203 TEE_Result TEE_CheckMemoryAccessRights(uint32_t accessFlags, void *buffer,
204                                        uint32_t size)
205 {
206         TEE_Result res;
207
208         if (size == 0)
209                 return TEE_SUCCESS;
210
211         /* Check access rights against memory mapping */
212         res = utee_check_access_rights(accessFlags, buffer, size);
213         if (res != TEE_SUCCESS)
214                 goto out;
215
216         /*
217         * Check access rights against input parameters
218         * Previous legacy code was removed and will need to be restored
219         */
220
221         res = TEE_SUCCESS;
222 out:
223         return res;
224 }
225
```

```
226 void TEE_SetInstanceData(const void *instanceData)
227 {
228         tee_api_instance_data = instanceData;
229 }
230
231 const void *TEE_GetInstanceData(void)
232 {
233         return tee_api_instance_data;
234 }
235
236 void *TEE_MemMove(void *dest, const void *src, uint32_t size)
237 {
238         return memmove(dest, src, size);
239 }
240
241 int32_t TEE_MemCompare(const void *buffer1, const void *buffer2, uint32_t size)
242 {
243         return memcmp(buffer1, buffer2, size);
244 }
245
246 void *TEE_MemFill(void *buff, uint32_t x, uint32_t size)
247 {
248         return memset(buff, x, size);
249 }
250
251 /* Date & Time API */
252
253 void TEE_GetSystemTime(TEE_Time *time)
254 {
255         TEE_Result res = utee_get_time(UTEE_TIME_CAT_SYSTEM, time);
256
257         if (res != TEE_SUCCESS)
258                 TEE_Panic(res);
259 }
260
261 TEE_Result TEE_Wait(uint32_t timeout)
262 {
263         TEE_Result res = utee_wait(timeout);
264
265         if (res != TEE_SUCCESS && res != TEE_ERROR_CANCEL)
266                 TEE_Panic(res);
267
268         return res;
269 }
270
271 TEE_Result TEE_GetTAPersistentTime(TEE_Time *time)
272 {
273         TEE_Result res;
274
```

```
275         res = utee_get_time(UTEE_TIME_CAT_TA_PERSISTENT, time);
276
277         if (res != TEE_SUCCESS && res != TEE_ERROR_OVERFLOW) {
278                 time->seconds = 0;
279                 time->millis = 0;
280         }
281
282         if (res != TEE_SUCCESS &&
283             res != TEE_ERROR_TIME_NOT_SET &&
284             res != TEE_ERROR_TIME_NEEDS_RESET &&
285             res != TEE_ERROR_OVERFLOW &&
286             res != TEE_ERROR_OUT_OF_MEMORY)
287                 TEE_Panic(res);
288
289         return res;
290 }
291
292 TEE_Result TEE_SetTAPersistentTime(const TEE_Time *time)
293 {
294         TEE_Result res;
295
296         res = utee_set_ta_time(time);
297
298         if (res != TEE_SUCCESS &&
299             res != TEE_ERROR_OUT_OF_MEMORY &&
300             res != TEE_ERROR_STORAGE_NO_SPACE)
301                 TEE_Panic(res);
302
303         return res;
304 }
305
306 void TEE_GetREETime(TEE_Time *time)
307 {
308         TEE_Result res = utee_get_time(UTEE_TIME_CAT_REE, time);
309
310         if (res != TEE_SUCCESS)
311                 TEE_Panic(res);
312 }
313
314 void *TEE_Malloc(uint32_t len, uint32_t hint)
315 {
316         return tee_user_mem_alloc(len, hint);
317 }
318
319 void *TEE_Realloc(const void *buffer, uint32_t newSize)
320 {
321         /*
322          * GP TEE Internal API specifies newSize as 'uint32_t'.
323          * use unsigned 'size_t' type. it is at least 32bit!
```

```
         */
        return tee_user_mem_realloc((void *)buffer, (size_t) newSize);
}

void TEE_Free(void *buffer)
{
  tee_user_mem_free(buffer);
}

/* Cache maintenance support (TA requires the CACHE_MAINTENANCE property) */
TEE_Result TEE_CacheClean(char *buf, size_t len)
{
        return utee_cache_operation(buf, len, TEE_CACHECLEAN);
}
TEE_Result TEE_CacheFlush(char *buf, size_t len)
{
        return utee_cache_operation(buf, len, TEE_CACHEFLUSH);
}

TEE_Result TEE_CacheInvalidate(char *buf, size_t len)
{
        return utee_cache_operation(buf, len, TEE_CACHEINVALIDATE);
}
```