Optee core

/\*

\* Copyright (c) 2015, Linaro Limited

\*

\* This software is licensed under the terms of the GNU General Public

\* License version 2, as published by the Free Software Foundation, and

\* may be copied, distributed, and modified under those terms.

\*

\* This program is distributed in the hope that it will be useful,

\* but WITHOUT ANY WARRANTY; without even the implied warranty of

\* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

\* GNU General Public License for more details.

\*

\*/

#define pr\_fmt(fmt) KBUILD\_MODNAME ": " fmt

#include <linux/arm-smccc.h>

#include <linux/errno.h>

#include <linux/io.h>

#include <linux/module.h>

#include <linux/of.h>

#include <linux/of\_platform.h>

#include <linux/platform\_device.h>

#include <linux/slab.h>

#include <linux/string.h>

#include <linux/tee\_drv.h>

#include <linux/types.h>

#include <linux/uaccess.h>

#include "optee\_private.h"

#include "optee\_smc.h"

#include "shm\_pool.h"

#define DRIVER\_NAME "optee"

#define OPTEE\_SHM\_NUM\_PRIV\_PAGES CONFIG\_OPTEE\_SHM\_NUM\_PRIV\_PAGES

/\*\*

\* optee\_from\_msg\_param() - convert from OPTEE\_MSG parameters to

\* struct tee\_param

\* @params: subsystem internal parameter representation

\* @num\_params: number of elements in the parameter arrays

\* @msg\_params: OPTEE\_MSG parameters

\* Returns 0 on success or <0 on failure

\*/

int optee\_from\_msg\_param(struct tee\_param \*params, size\_t num\_params,

const struct optee\_msg\_param \*msg\_params)

{

int rc;

size\_t n;

struct tee\_shm \*shm;

phys\_addr\_t pa;

for (n = 0; n < num\_params; n++) {

struct tee\_param \*p = params + n;

const struct optee\_msg\_param \*mp = msg\_params + n;

u32 attr = mp->attr & OPTEE\_MSG\_ATTR\_TYPE\_MASK;

switch (attr) {

case OPTEE\_MSG\_ATTR\_TYPE\_NONE:

p->attr = TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_NONE;

memset(&p->u, 0, sizeof(p->u));

break;

case OPTEE\_MSG\_ATTR\_TYPE\_VALUE\_INPUT:

case OPTEE\_MSG\_ATTR\_TYPE\_VALUE\_OUTPUT:

case OPTEE\_MSG\_ATTR\_TYPE\_VALUE\_INOUT:

p->attr = TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_VALUE\_INPUT +

attr - OPTEE\_MSG\_ATTR\_TYPE\_VALUE\_INPUT;

p->u.value.a = mp->u.value.a;

p->u.value.b = mp->u.value.b;

p->u.value.c = mp->u.value.c;

break;

case OPTEE\_MSG\_ATTR\_TYPE\_TMEM\_INPUT:

case OPTEE\_MSG\_ATTR\_TYPE\_TMEM\_OUTPUT:

case OPTEE\_MSG\_ATTR\_TYPE\_TMEM\_INOUT:

p->attr = TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_MEMREF\_INPUT +

attr - OPTEE\_MSG\_ATTR\_TYPE\_TMEM\_INPUT;

p->u.memref.size = mp->u.tmem.size;

shm = (struct tee\_shm \*)(unsigned long)

mp->u.tmem.shm\_ref;

if (!shm) {

p->u.memref.shm\_offs = 0;

p->u.memref.shm = NULL;

break;

}

rc = tee\_shm\_get\_pa(shm, 0, &pa);

if (rc)

return rc;

p->u.memref.shm\_offs = mp->u.tmem.buf\_ptr - pa;

p->u.memref.shm = shm;

/\* Check that the memref is covered by the shm object \*/

if (p->u.memref.size) {

size\_t o = p->u.memref.shm\_offs +

p->u.memref.size - 1;

rc = tee\_shm\_get\_pa(shm, o, NULL);

if (rc)

return rc;

}

break;

case OPTEE\_MSG\_ATTR\_TYPE\_RMEM\_INPUT:

case OPTEE\_MSG\_ATTR\_TYPE\_RMEM\_OUTPUT:

case OPTEE\_MSG\_ATTR\_TYPE\_RMEM\_INOUT:

p->attr = TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_MEMREF\_INPUT +

attr - OPTEE\_MSG\_ATTR\_TYPE\_RMEM\_INPUT;

p->u.memref.size = mp->u.rmem.size;

shm = (struct tee\_shm \*)(unsigned long)

mp->u.rmem.shm\_ref;

if (!shm) {

p->u.memref.shm\_offs = 0;

p->u.memref.shm = NULL;

break;

}

p->u.memref.shm\_offs = mp->u.rmem.offs;

p->u.memref.shm = shm;

break;

default:

return -EINVAL;

}

}

return 0;

}

static int to\_msg\_param\_tmp\_mem(struct optee\_msg\_param \*mp,

const struct tee\_param \*p)

{

int rc;

phys\_addr\_t pa;

mp->attr = OPTEE\_MSG\_ATTR\_TYPE\_TMEM\_INPUT + p->attr -

TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_MEMREF\_INPUT;

mp->u.tmem.shm\_ref = (unsigned long)p->u.memref.shm;

mp->u.tmem.size = p->u.memref.size;

if (!p->u.memref.shm) {

mp->u.tmem.buf\_ptr = 0;/\*

\* Copyright (c) 2015, Linaro Limited

\*

\* This software is licensed under the terms of the GNU General Public

\* License version 2, as published by the Free Software Foundation, and

\* may be copied, distributed, and modified under those terms.

\*

\* This program is distributed in the hope that it will be useful,

\* but WITHOUT ANY WARRANTY; without even the implied warranty of

\* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

\* GNU General Public License for more details.

\*

\*/

#define pr\_fmt(fmt) KBUILD\_MODNAME ": " fmt

#include <linux/arm-smccc.h>

#include <linux/errno.h>

#include <linux/io.h>

#include <linux/module.h>

#include <linux/of.h>

#include <linux/of\_platform.h>

#include <linux/platform\_device.h>

#include <linux/slab.h>

#include <linux/string.h>

#include <linux/tee\_drv.h>

#include <linux/types.h>

#include <linux/uaccess.h>

#include "optee\_private.h"

#include "optee\_smc.h"

#include "shm\_pool.h"

#define DRIVER\_NAME "optee"

#define OPTEE\_SHM\_NUM\_PRIV\_PAGES CONFIG\_OPTEE\_SHM\_NUM\_PRIV\_PAGES

/\*\*

\* optee\_from\_msg\_param() - convert from OPTEE\_MSG parameters to

\* struct tee\_param

\* @params: subsystem internal parameter representation

\* @num\_params: number of elements in the parameter arrays

\* @msg\_params: OPTEE\_MSG parameters

\* Returns 0 on success or <0 on failure

\*/

int optee\_from\_msg\_param(struct tee\_param \*params, size\_t num\_params,

const struct optee\_msg\_param \*msg\_params)

{

int rc;

size\_t n;

struct tee\_shm \*shm;

phys\_addr\_t pa;

for (n = 0; n < num\_params; n++) {

struct tee\_param \*p = params + n;

const struct optee\_msg\_param \*mp = msg\_params + n;

u32 attr = mp->attr & OPTEE\_MSG\_ATTR\_TYPE\_MASK;

switch (attr) {

case OPTEE\_MSG\_ATTR\_TYPE\_NONE:

p->attr = TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_NONE;

memset(&p->u, 0, sizeof(p->u));

break;

case OPTEE\_MSG\_ATTR\_TYPE\_VALUE\_INPUT:

case OPTEE\_MSG\_ATTR\_TYPE\_VALUE\_OUTPUT:

case OPTEE\_MSG\_ATTR\_TYPE\_VALUE\_INOUT:

p->attr = TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_VALUE\_INPUT +

attr - OPTEE\_MSG\_ATTR\_TYPE\_VALUE\_INPUT;

p->u.value.a = mp->u.value.a;

p->u.value.b = mp->u.value.b;

p->u.value.c = mp->u.value.c;

break;

case OPTEE\_MSG\_ATTR\_TYPE\_TMEM\_INPUT:

case OPTEE\_MSG\_ATTR\_TYPE\_TMEM\_OUTPUT:

case OPTEE\_MSG\_ATTR\_TYPE\_TMEM\_INOUT:

p->attr = TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_MEMREF\_INPUT +

attr - OPTEE\_MSG\_ATTR\_TYPE\_TMEM\_INPUT;

p->u.memref.size = mp->u.tmem.size;

shm = (struct tee\_shm \*)(unsigned long)

mp->u.tmem.shm\_ref;

if (!shm) {

p->u.memref.shm\_offs = 0;

p->u.memref.shm = NULL;

break;

}

rc = tee\_shm\_get\_pa(shm, 0, &pa);

if (rc)

return rc;

p->u.memref.shm\_offs = mp->u.tmem.buf\_ptr - pa;

p->u.memref.shm = shm;

/\* Check that the memref is covered by the shm object \*/

if (p->u.memref.size) {

size\_t o = p->u.memref.shm\_offs +

p->u.memref.size - 1;

rc = tee\_shm\_get\_pa(shm, o, NULL);

if (rc)

return rc;

}

break;

case OPTEE\_MSG\_ATTR\_TYPE\_RMEM\_INPUT:

case OPTEE\_MSG\_ATTR\_TYPE\_RMEM\_OUTPUT:

case OPTEE\_MSG\_ATTR\_TYPE\_RMEM\_INOUT:

p->attr = TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_MEMREF\_INPUT +

attr - OPTEE\_MSG\_ATTR\_TYPE\_RMEM\_INPUT;

p->u.memref.size = mp->u.rmem.size;

shm = (struct tee\_shm \*)(unsigned long)

mp->u.rmem.shm\_ref;

if (!shm) {

p->u.memref.shm\_offs = 0;

p->u.memref.shm = NULL;

break;

}

p->u.memref.shm\_offs = mp->u.rmem.offs;

p->u.memref.shm = shm;

break;

default:

return -EINVAL;

}

}

return 0;

}

static int to\_msg\_param\_tmp\_mem(struct optee\_msg\_param \*mp,

const struct tee\_param \*p)

{

int rc;

phys\_addr\_t pa;

mp->attr = OPTEE\_MSG\_ATTR\_TYPE\_TMEM\_INPUT + p->attr -

TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_MEMREF\_INPUT;

mp->u.tmem.shm\_ref = (unsigned long)p->u.memref.shm;

mp->u.tmem.size = p->u.memref.size;

if (!p->u.memref.shm) {

mp->u.tmem.buf\_ptr = 0;

return 0;

}

rc = tee\_shm\_get\_pa(p->u.memref.shm, p->u.memref.shm\_offs, &pa);

if (rc)

return rc;

mp->u.tmem.buf\_ptr = pa;

mp->attr |= OPTEE\_MSG\_ATTR\_CACHE\_PREDEFINED <<

OPTEE\_MSG\_ATTR\_CACHE\_SHIFT;

return 0;

}

static int to\_msg\_param\_reg\_mem(struct optee\_msg\_param \*mp,

const struct tee\_param \*p)

{

mp->attr = OPTEE\_MSG\_ATTR\_TYPE\_RMEM\_INPUT + p->attr -

TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_MEMREF\_INPUT;

mp->u.rmem.shm\_ref = (unsigned long)p->u.memref.shm;

mp->u.rmem.size = p->u.memref.size;

mp->u.rmem.offs = p->u.memref.shm\_offs;

return 0;

}

/\*\*

\* optee\_to\_msg\_param() - convert from struct tee\_params to OPTEE\_MSG parameters

\* @msg\_params: OPTEE\_MSG parameters

\* @num\_params: number of elements in the parameter arrays

\* @params: subsystem itnernal parameter representation

\* Returns 0 on success or <0 on failure

\*/

int optee\_to\_msg\_param(struct optee\_msg\_param \*msg\_params, size\_t num\_params,

const struct tee\_param \*params)

{

int rc;

size\_t n;

for (n = 0; n < num\_params; n++) {

const struct tee\_param \*p = params + n;

struct optee\_msg\_param \*mp = msg\_params + n;

switch (p->attr) {

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_NONE:

mp->attr = TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_NONE;

memset(&mp->u, 0, sizeof(mp->u));

break;

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_VALUE\_INPUT:

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_VALUE\_OUTPUT:

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_VALUE\_INOUT:

mp->attr = OPTEE\_MSG\_ATTR\_TYPE\_VALUE\_INPUT + p->attr -

TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_VALUE\_INPUT;

mp->u.value.a = p->u.value.a;

mp->u.value.b = p->u.value.b;

mp->u.value.c = p->u.value.c;

break;

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_MEMREF\_INPUT:

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_MEMREF\_OUTPUT:

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_MEMREF\_INOUT:

if (tee\_shm\_is\_registered(p->u.memref.shm))

rc = to\_msg\_param\_reg\_mem(mp, p);

else

rc = to\_msg\_param\_tmp\_mem(mp, p);

if (rc)

return rc;

break;

default:

return -EINVAL;

}

}

return 0;

}

static void optee\_get\_version(struct tee\_device \*teedev,

struct tee\_ioctl\_version\_data \*vers)

{

struct tee\_ioctl\_version\_data v = {

.impl\_id = TEE\_IMPL\_ID\_OPTEE,

.impl\_caps = TEE\_OPTEE\_CAP\_TZ,

.gen\_caps = TEE\_GEN\_CAP\_GP,

};

struct optee \*optee = tee\_get\_drvdata(teedev);

if (optee->sec\_caps & OPTEE\_SMC\_SEC\_CAP\_DYNAMIC\_SHM)

v.gen\_caps |= TEE\_GEN\_CAP\_REG\_MEM;

\*vers = v;

}

static int optee\_open(struct tee\_context \*ctx)

{

struct optee\_context\_data \*ctxdata;

struct tee\_device \*teedev = ctx->teedev;

struct optee \*optee = tee\_get\_drvdata(teedev);

ctxdata = kzalloc(sizeof(\*ctxdata), GFP\_KERNEL);

if (!ctxdata)

return -ENOMEM;

if (teedev == optee->supp\_teedev) {

bool busy = true;

mutex\_lock(&optee->supp.mutex);

if (!optee->supp.ctx) {

busy = false;

optee->supp.ctx = ctx;

}

mutex\_unlock(&optee->supp.mutex);

if (busy) {

kfree(ctxdata);

return -EBUSY;

}

}

mutex\_init(&ctxdata->mutex);

INIT\_LIST\_HEAD(&ctxdata->sess\_list);

ctx->data = ctxdata;

return 0;

}

static void optee\_release(struct tee\_context \*ctx)

{

struct optee\_context\_data \*ctxdata = ctx->data;

struct tee\_device \*teedev = ctx->teedev;

struct optee \*optee = tee\_get\_drvdata(teedev);

struct tee\_shm \*shm;

struct optee\_msg\_arg \*arg = NULL;

phys\_addr\_t parg;

struct optee\_session \*sess;

struct optee\_session \*sess\_tmp;

if (!ctxdata)

return;

shm = tee\_shm\_alloc(ctx, sizeof(struct optee\_msg\_arg), TEE\_SHM\_MAPPED);

if (!IS\_ERR(shm)) {

arg = tee\_shm\_get\_va(shm, 0);

/\*

\* If va2pa fails for some reason, we can't call into

\* secure world, only free the memory. Secure OS will leak

\* sessions and finally refuse more sessions, but we will

\* at least let normal world reclaim its memory.

\*/

if (!IS\_ERR(arg))

if (tee\_shm\_va2pa(shm, arg, &parg))

arg = NULL; /\* prevent usage of parg below \*/

}

list\_for\_each\_entry\_safe(sess, sess\_tmp, &ctxdata->sess\_list,

list\_node) {

list\_del(&sess->list\_node);

if (!IS\_ERR\_OR\_NULL(arg)) {

memset(arg, 0, sizeof(\*arg));

arg->cmd = OPTEE\_MSG\_CMD\_CLOSE\_SESSION;

arg->session = sess->session\_id;

optee\_do\_call\_with\_arg(ctx, parg);

}

kfree(sess);

}

kfree(ctxdata);

if (!IS\_ERR(shm))

tee\_shm\_free(shm);

ctx->data = NULL;

if (teedev == optee->supp\_teedev)

optee\_supp\_release(&optee->supp);

}

static const struct tee\_driver\_ops optee\_ops = {

.get\_version = optee\_get\_version,

.open = optee\_open,

.release = optee\_release,

.open\_session = optee\_open\_session,

.close\_session = optee\_close\_session,

.invoke\_func = optee\_invoke\_func,

.cancel\_req = optee\_cancel\_req,

.shm\_register = optee\_shm\_register,

.shm\_unregister = optee\_shm\_unregister,

};

static const struct tee\_desc optee\_desc = {

.name = DRIVER\_NAME "-clnt",

.ops = &optee\_ops,

.owner = THIS\_MODULE,

};

static const struct tee\_driver\_ops optee\_supp\_ops = {

.get\_version = optee\_get\_version,

.open = optee\_open,

.release = optee\_release,

.supp\_recv = optee\_supp\_recv,

.supp\_send = optee\_supp\_send,

.shm\_register = optee\_shm\_register\_supp,

.shm\_unregister = optee\_shm\_unregister\_supp,

};

static const struct tee\_desc optee\_supp\_desc = {

.name = DRIVER\_NAME "-supp",

.ops = &optee\_supp\_ops,

.owner = THIS\_MODULE,

.flags = TEE\_DESC\_PRIVILEGED,

};

static bool optee\_msg\_api\_uid\_is\_optee\_api(optee\_invoke\_fn \*invoke\_fn)

{

struct arm\_smccc\_res res;

invoke\_fn(OPTEE\_SMC\_CALLS\_UID, 0, 0, 0, 0, 0, 0, 0, &res);

if (res.a0 == OPTEE\_MSG\_UID\_0 && res.a1 == OPTEE\_MSG\_UID\_1 &&

res.a2 == OPTEE\_MSG\_UID\_2 && res.a3 == OPTEE\_MSG\_UID\_3)

return true;

return false;

}

static void optee\_msg\_get\_os\_revision(optee\_invoke\_fn \*invoke\_fn)

{

union {

struct arm\_smccc\_res smccc;

struct optee\_smc\_call\_get\_os\_revision\_result result;

} res = {

.result = {

.build\_id = 0

}

};

invoke\_fn(OPTEE\_SMC\_CALL\_GET\_OS\_REVISION, 0, 0, 0, 0, 0, 0, 0,

&res.smccc);

if (res.result.build\_id)

pr\_info("revision %lu.%lu (%08lx)", res.result.major,

res.result.minor, res.result.build\_id);

else

pr\_info("revision %lu.%lu", res.result.major, res.result.minor);

}

static bool optee\_msg\_api\_revision\_is\_compatible(optee\_invoke\_fn \*invoke\_fn)

{

union {

struct arm\_smccc\_res smccc;

struct optee\_smc\_calls\_revision\_result result;

} res;

invoke\_fn(OPTEE\_SMC\_CALLS\_REVISION, 0, 0, 0, 0, 0, 0, 0, &res.smccc);

if (res.result.major == OPTEE\_MSG\_REVISION\_MAJOR &&

(int)res.result.minor >= OPTEE\_MSG\_REVISION\_MINOR)

return true;

return false;

}

static bool optee\_msg\_exchange\_capabilities(optee\_invoke\_fn \*invoke\_fn,

u32 \*sec\_caps)

{

union {

struct arm\_smccc\_res smccc;

struct optee\_smc\_exchange\_capabilities\_result result;

} res;

u32 a1 = 0;

/\*

\* TODO This isn't enough to tell if it's UP system (from kernel

\* point of view) or not, is\_smp() returns the the information

\* needed, but can't be called directly from here.

\*/

if (!IS\_ENABLED(CONFIG\_SMP) || nr\_cpu\_ids == 1)

a1 |= OPTEE\_SMC\_NSEC\_CAP\_UNIPROCESSOR;

invoke\_fn(OPTEE\_SMC\_EXCHANGE\_CAPABILITIES, a1, 0, 0, 0, 0, 0, 0,

&res.smccc);

if (res.result.status != OPTEE\_SMC\_RETURN\_OK)

return false;

\*sec\_caps = res.result.capabilities;

return true;

}

static struct tee\_shm\_pool \*

optee\_config\_shm\_memremap(optee\_invoke\_fn \*invoke\_fn, void \*\*memremaped\_shm,

u32 sec\_caps)

{

union {

struct arm\_smccc\_res smccc;

struct optee\_smc\_get\_shm\_config\_result result;

} res;

unsigned long vaddr;

phys\_addr\_t paddr;

size\_t size;

phys\_addr\_t begin;

phys\_addr\_t end;

void \*va;

struct tee\_shm\_pool\_mgr \*priv\_mgr;

struct tee\_shm\_pool\_mgr \*dmabuf\_mgr;

void \*rc;

invoke\_fn(OPTEE\_SMC\_GET\_SHM\_CONFIG, 0, 0, 0, 0, 0, 0, 0, &res.smccc);

if (res.result.status != OPTEE\_SMC\_RETURN\_OK) {

pr\_info("shm service not available\n");

return ERR\_PTR(-ENOENT);

}

if (res.result.settings != OPTEE\_SMC\_SHM\_CACHED) {

pr\_err("only normal cached shared memory supported\n");

return ERR\_PTR(-EINVAL);

}

begin = roundup(res.result.start, PAGE\_SIZE);

end = rounddown(res.result.start + res.result.size, PAGE\_SIZE);

paddr = begin;

size = end - begin;

if (size < 2 \* OPTEE\_SHM\_NUM\_PRIV\_PAGES \* PAGE\_SIZE) {

pr\_err("too small shared memory area\n");

return ERR\_PTR(-EINVAL);

}

va = memremap(paddr, size, MEMREMAP\_WB);

if (!va) {

pr\_err("shared memory ioremap failed\n");

return ERR\_PTR(-EINVAL);

}

vaddr = (unsigned long)va;

/\*

\* If OP-TEE can work with unregistered SHM, we will use own pool

\* for private shm

\*/

if (sec\_caps & OPTEE\_SMC\_SEC\_CAP\_DYNAMIC\_SHM) {

rc = optee\_shm\_pool\_alloc\_pages();

if (IS\_ERR(rc))

goto err\_memunmap;

priv\_mgr = rc;

} else {

const size\_t sz = OPTEE\_SHM\_NUM\_PRIV\_PAGES \* PAGE\_SIZE;

rc = tee\_shm\_pool\_mgr\_alloc\_res\_mem(vaddr, paddr, sz,

3 /\* 8 bytes aligned \*/);

if (IS\_ERR(rc))

goto err\_memunmap;

priv\_mgr = rc;

vaddr += sz;

paddr += sz;

size -= sz;

}

rc = tee\_shm\_pool\_mgr\_alloc\_res\_mem(vaddr, paddr, size, PAGE\_SHIFT);

if (IS\_ERR(rc))

goto err\_free\_priv\_mgr;

dmabuf\_mgr = rc;

rc = tee\_shm\_pool\_alloc(priv\_mgr, dmabuf\_mgr);

if (IS\_ERR(rc))

goto err\_free\_dmabuf\_mgr;

\*memremaped\_shm = va;

return rc;

err\_free\_dmabuf\_mgr:

tee\_shm\_pool\_mgr\_destroy(dmabuf\_mgr);

err\_free\_priv\_mgr:

tee\_shm\_pool\_mgr\_destroy(priv\_mgr);

err\_memunmap:

memunmap(va);

return rc;

}

/\* Simple wrapper functions to be able to use a function pointer \*/

static void optee\_smccc\_smc(unsigned long a0, unsigned long a1,

unsigned long a2, unsigned long a3,

unsigned long a4, unsigned long a5,

unsigned long a6, unsigned long a7,

struct arm\_smccc\_res \*res)

{

arm\_smccc\_smc(a0, a1, a2, a3, a4, a5, a6, a7, res);

}

static void optee\_smccc\_hvc(unsigned long a0, unsigned long a1,

unsigned long a2, unsigned long a3,

unsigned long a4, unsigned long a5,

unsigned long a6, unsigned long a7,

struct arm\_smccc\_res \*res)

{

arm\_smccc\_hvc(a0, a1, a2, a3, a4, a5, a6, a7, res);

}

static optee\_invoke\_fn \*get\_invoke\_func(struct device\_node \*np)

{

const char \*method;

pr\_info("probing for conduit method from DT.\n");

if (of\_property\_read\_string(np, "method", &method)) {

pr\_warn("missing \"method\" property\n");

return ERR\_PTR(-ENXIO);

}

if (!strcmp("hvc", method))

return optee\_smccc\_hvc;

else if (!strcmp("smc", method))

return optee\_smccc\_smc;

pr\_warn("invalid \"method\" property: %s\n", method);

return ERR\_PTR(-EINVAL);

}

static struct optee \*optee\_probe(struct device\_node \*np)

{

optee\_invoke\_fn \*invoke\_fn;

struct tee\_shm\_pool \*pool;

struct optee \*optee = NULL;

void \*memremaped\_shm = NULL;

struct tee\_device \*teedev;

u32 sec\_caps;

int rc;

invoke\_fn = get\_invoke\_func(np);

if (IS\_ERR(invoke\_fn))

return (void \*)invoke\_fn;

if (!optee\_msg\_api\_uid\_is\_optee\_api(invoke\_fn)) {

pr\_warn("api uid mismatch\n");

return ERR\_PTR(-EINVAL);

}

optee\_msg\_get\_os\_revision(invoke\_fn);

if (!optee\_msg\_api\_revision\_is\_compatible(invoke\_fn)) {

pr\_warn("api revision mismatch\n");

return ERR\_PTR(-EINVAL);

}

if (!optee\_msg\_exchange\_capabilities(invoke\_fn, &sec\_caps)) {

pr\_warn("capabilities mismatch\n");

return ERR\_PTR(-EINVAL);

}

/\*

\* We have no other option for shared memory, if secure world

\* doesn't have any reserved memory we can use we can't continue.

\*/

if (!(sec\_caps & OPTEE\_SMC\_SEC\_CAP\_HAVE\_RESERVED\_SHM))

return ERR\_PTR(-EINVAL);

pool = optee\_config\_shm\_memremap(invoke\_fn, &memremaped\_shm, sec\_caps);

if (IS\_ERR(pool))

return (void \*)pool;

optee = kzalloc(sizeof(\*optee), GFP\_KERNEL);

if (!optee) {

rc = -ENOMEM;

goto err;

}

optee->invoke\_fn = invoke\_fn;

optee->sec\_caps = sec\_caps;

teedev = tee\_device\_alloc(&optee\_desc, NULL, pool, optee);

if (IS\_ERR(teedev)) {

rc = PTR\_ERR(teedev);

goto err;

}

optee->teedev = teedev;

teedev = tee\_device\_alloc(&optee\_supp\_desc, NULL, pool, optee);

if (IS\_ERR(teedev)) {

rc = PTR\_ERR(teedev);

goto err;

}

optee->supp\_teedev = teedev;

rc = tee\_device\_register(optee->teedev);

if (rc)

goto err;

rc = tee\_device\_register(optee->supp\_teedev);

if (rc)

goto err;

mutex\_init(&optee->call\_queue.mutex);

INIT\_LIST\_HEAD(&optee->call\_queue.waiters);

optee\_wait\_queue\_init(&optee->wait\_queue);

optee\_supp\_init(&optee->supp);

optee->memremaped\_shm = memremaped\_shm;

optee->pool = pool;

optee\_enable\_shm\_cache(optee);

pr\_info("initialized driver\n");

return optee;

err:

if (optee) {

/\*

\* tee\_device\_unregister() is safe to call even if the

\* devices hasn't been registered with

\* tee\_device\_register() yet.

\*/

tee\_device\_unregister(optee->supp\_teedev);

tee\_device\_unregister(optee->teedev);

kfree(optee);

}

if (pool)

tee\_shm\_pool\_free(pool);

if (memremaped\_shm)

memunmap(memremaped\_shm);

return ERR\_PTR(rc);

}

static void optee\_remove(struct optee \*optee)

{

/\*

\* Ask OP-TEE to free all cached shared memory objects to decrease

\* reference counters and also avoid wild pointers in secure world

\* into the old shared memory range.

\*/

optee\_disable\_shm\_cache(optee);

/\*

\* The two devices has to be unregistered before we can free the

\* other resources.

\*/

tee\_device\_unregister(optee->supp\_teedev);

tee\_device\_unregister(optee->teedev);

tee\_shm\_pool\_free(optee->pool);

if (optee->memremaped\_shm)

memunmap(optee->memremaped\_shm);

optee\_wait\_queue\_exit(&optee->wait\_queue);

optee\_supp\_uninit(&optee->supp);

mutex\_destroy(&optee->call\_queue.mutex);

kfree(optee);

}

static const struct of\_device\_id optee\_match[] = {

{ .compatible = "linaro,optee-tz" },

{},

};

static struct optee \*optee\_svc;

static int \_\_init optee\_driver\_init(void)

{

struct device\_node \*fw\_np;

struct device\_node \*np;

struct optee \*optee;

/\* Node is supposed to be below /firmware \*/

fw\_np = of\_find\_node\_by\_name(NULL, "firmware");

if (!fw\_np)

return -ENODEV;

np = of\_find\_matching\_node(fw\_np, optee\_match);

if (!np)

return -ENODEV;

optee = optee\_probe(np);

of\_node\_put(np);

if (IS\_ERR(optee))

return PTR\_ERR(optee);

optee\_svc = optee;

return 0;

}

module\_init(optee\_driver\_init);

static void \_\_exit optee\_driver\_exit(void)

{

struct optee \*optee = optee\_svc;

optee\_svc = NULL;

if (optee)

optee\_remove(optee);

}

module\_exit(optee\_driver\_exit);

MODULE\_AUTHOR("Linaro");

MODULE\_DESCRIPTION("OP-TEE driver");

MODULE\_SUPPORTED\_DEVICE("");

MODULE\_VERSION("1.0");

MODULE\_LICENSE("GPL v2");

return 0;

}

rc = tee\_shm\_get\_pa(p->u.memref.shm, p->u.memref.shm\_offs, &pa);

if (rc)

return rc;

mp->u.tmem.buf\_ptr = pa;

mp->attr |= OPTEE\_MSG\_ATTR\_CACHE\_PREDEFINED <<

OPTEE\_MSG\_ATTR\_CACHE\_SHIFT;

return 0;

}

static int to\_msg\_param\_reg\_mem(struct optee\_msg\_param \*mp,

const struct tee\_param \*p)

{

mp->attr = OPTEE\_MSG\_ATTR\_TYPE\_RMEM\_INPUT + p->attr -

TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_MEMREF\_INPUT;

mp->u.rmem.shm\_ref = (unsigned long)p->u.memref.shm;

mp->u.rmem.size = p->u.memref.size;

mp->u.rmem.offs = p->u.memref.shm\_offs;

return 0;

}

/\*\*

\* optee\_to\_msg\_param() - convert from struct tee\_params to OPTEE\_MSG parameters

\* @msg\_params: OPTEE\_MSG parameters

\* @num\_params: number of elements in the parameter arrays

\* @params: subsystem itnernal parameter representation

\* Returns 0 on success or <0 on failure

\*/

int optee\_to\_msg\_param(struct optee\_msg\_param \*msg\_params, size\_t num\_params,

const struct tee\_param \*params)

{

int rc;

size\_t n;

for (n = 0; n < num\_params; n++) {

const struct tee\_param \*p = params + n;

struct optee\_msg\_param \*mp = msg\_params + n;

switch (p->attr) {

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_NONE:

mp->attr = TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_NONE;

memset(&mp->u, 0, sizeof(mp->u));

break;

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_VALUE\_INPUT:

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_VALUE\_OUTPUT:

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_VALUE\_INOUT:

mp->attr = OPTEE\_MSG\_ATTR\_TYPE\_VALUE\_INPUT + p->attr -

TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_VALUE\_INPUT;

mp->u.value.a = p->u.value.a;

mp->u.value.b = p->u.value.b;

mp->u.value.c = p->u.value.c;

break;

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_MEMREF\_INPUT:

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_MEMREF\_OUTPUT:

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_MEMREF\_INOUT:

if (tee\_shm\_is\_registered(p->u.memref.shm))

rc = to\_msg\_param\_reg\_mem(mp, p);

else

rc = to\_msg\_param\_tmp\_mem(mp, p);

if (rc)

return rc;

break;

default:

return -EINVAL;

}

}

return 0;

}

static void optee\_get\_version(struct tee\_device \*teedev,

struct tee\_ioctl\_version\_data \*vers)

{

struct tee\_ioctl\_version\_data v = {

.impl\_id = TEE\_IMPL\_ID\_OPTEE,

.impl\_caps = TEE\_OPTEE\_CAP\_TZ,

.gen\_caps = TEE\_GEN\_CAP\_GP,

};

struct optee \*optee = tee\_get\_drvdata(teedev);

if (optee->sec\_caps & OPTEE\_SMC\_SEC\_CAP\_DYNAMIC\_SHM)

v.gen\_caps |= TEE\_GEN\_CAP\_REG\_MEM;

\*vers = v;

}

static int optee\_open(struct tee\_context \*ctx)

{

struct optee\_context\_data \*ctxdata;

struct tee\_device \*teedev = ctx->teedev;

struct optee \*optee = tee\_get\_drvdata(teedev);

ctxdata = kzalloc(sizeof(\*ctxdata), GFP\_KERNEL);

if (!ctxdata)

return -ENOMEM;

if (teedev == optee->supp\_teedev) {

bool busy = true;

mutex\_lock(&optee->supp.mutex);

if (!optee->supp.ctx) {

busy = false;

optee->supp.ctx = ctx;

}

mutex\_unlock(&optee->supp.mutex);

if (busy) {

kfree(ctxdata);

return -EBUSY;

}

}

mutex\_init(&ctxdata->mutex);

INIT\_LIST\_HEAD(&ctxdata->sess\_list);

ctx->data = ctxdata;

return 0;

}

static void optee\_release(struct tee\_context \*ctx)

{

struct optee\_context\_data \*ctxdata = ctx->data;

struct tee\_device \*teedev = ctx->teedev;

struct optee \*optee = tee\_get\_drvdata(teedev);

struct tee\_shm \*shm;

struct optee\_msg\_arg \*arg = NULL;

phys\_addr\_t parg;

struct optee\_session \*sess;

struct optee\_session \*sess\_tmp;

if (!ctxdata)

return;

shm = tee\_shm\_alloc(ctx, sizeof(struct optee\_msg\_arg), TEE\_SHM\_MAPPED);

if (!IS\_ERR(shm)) {

arg = tee\_shm\_get\_va(shm, 0);

/\*

\* If va2pa fails for some reason, we can't call into

\* secure world, only free the memory. Secure OS will leak

\* sessions and finally refuse more sessions, but we will

\* at least let normal world reclaim its memory.

\*/

if (!IS\_ERR(arg))

if (tee\_shm\_va2pa(shm, arg, &parg))

arg = NULL; /\* prevent usage of parg below \*/

}

list\_for\_each\_entry\_safe(sess, sess\_tmp, &ctxdata->sess\_list,

list\_node) {

list\_del(&sess->list\_node);

if (!IS\_ERR\_OR\_NULL(arg)) {

memset(arg, 0, sizeof(\*arg));

arg->cmd = OPTEE\_MSG\_CMD\_CLOSE\_SESSION;

arg->session = sess->session\_id;

optee\_do\_call\_with\_arg(ctx, parg);

}

kfree(sess);

}

kfree(ctxdata);

if (!IS\_ERR(shm))

tee\_shm\_free(shm);

ctx->data = NULL;

if (teedev == optee->supp\_teedev)

optee\_supp\_release(&optee->supp);

}

static const struct tee\_driver\_ops optee\_ops = {

.get\_version = optee\_get\_version,

.open = optee\_open,

.release = optee\_release,

.open\_session = optee\_open\_session,

.close\_session = optee\_close\_session,

.invoke\_func = optee\_invoke\_func,

.cancel\_req = optee\_cancel\_req,

.shm\_register = optee\_shm\_register,

.shm\_unregister = optee\_shm\_unregister,

};

static const struct tee\_desc optee\_desc = {

.name = DRIVER\_NAME "-clnt",

.ops = &optee\_ops,

.owner = THIS\_MODULE,

};

static const struct tee\_driver\_ops optee\_supp\_ops = {

.get\_version = optee\_get\_version,

.open = optee\_open,

.release = optee\_release,

.supp\_recv = optee\_supp\_recv,

.supp\_send = optee\_supp\_send,

.shm\_register = optee\_shm\_register\_supp,

.shm\_unregister = optee\_shm\_unregister\_supp,

};

static const struct tee\_desc optee\_supp\_desc = {

.name = DRIVER\_NAME "-supp",

.ops = &optee\_supp\_ops,

.owner = THIS\_MODULE,

.flags = TEE\_DESC\_PRIVILEGED,

};

static bool optee\_msg\_api\_uid\_is\_optee\_api(optee\_invoke\_fn \*invoke\_fn)

{

struct arm\_smccc\_res res;

invoke\_fn(OPTEE\_SMC\_CALLS\_UID, 0, 0, 0, 0, 0, 0, 0, &res);

if (res.a0 == OPTEE\_MSG\_UID\_0 && res.a1 == OPTEE\_MSG\_UID\_1 &&

res.a2 == OPTEE\_MSG\_UID\_2 && res.a3 == OPTEE\_MSG\_UID\_3)

return true;

return false;

}

static void optee\_msg\_get\_os\_revision(optee\_invoke\_fn \*invoke\_fn)

{

union {

struct arm\_smccc\_res smccc;

struct optee\_smc\_call\_get\_os\_revision\_result result;

} res = {

.result = {

.build\_id = 0

}

};

invoke\_fn(OPTEE\_SMC\_CALL\_GET\_OS\_REVISION, 0, 0, 0, 0, 0, 0, 0,

&res.smccc);

if (res.result.build\_id)

pr\_info("revision %lu.%lu (%08lx)", res.result.major,

res.result.minor, res.result.build\_id);

else

pr\_info("revision %lu.%lu", res.result.major, res.result.minor);

}

static bool optee\_msg\_api\_revision\_is\_compatible(optee\_invoke\_fn \*invoke\_fn)

{

union {

struct arm\_smccc\_res smccc;

struct optee\_smc\_calls\_revision\_result result;

} res;

invoke\_fn(OPTEE\_SMC\_CALLS\_REVISION, 0, 0, 0, 0, 0, 0, 0, &res.smccc);

if (res.result.major == OPTEE\_MSG\_REVISION\_MAJOR &&

(int)res.result.minor >= OPTEE\_MSG\_REVISION\_MINOR)

return true;

return false;

}

static bool optee\_msg\_exchange\_capabilities(optee\_invoke\_fn \*invoke\_fn,

u32 \*sec\_caps)

{

union {

struct arm\_smccc\_res smccc;

struct optee\_smc\_exchange\_capabilities\_result result;

} res;

u32 a1 = 0;

/\*

\* TODO This isn't enough to tell if it's UP system (from kernel

\* point of view) or not, is\_smp() returns the the information

\* needed, but can't be called directly from here.

\*/

if (!IS\_ENABLED(CONFIG\_SMP) || nr\_cpu\_ids == 1)

a1 |= OPTEE\_SMC\_NSEC\_CAP\_UNIPROCESSOR;

invoke\_fn(OPTEE\_SMC\_EXCHANGE\_CAPABILITIES, a1, 0, 0, 0, 0, 0, 0,

&res.smccc);

if (res.result.status != OPTEE\_SMC\_RETURN\_OK)

return false;

\*sec\_caps = res.result.capabilities;

return true;

}

static struct tee\_shm\_pool \*

optee\_config\_shm\_memremap(optee\_invoke\_fn \*invoke\_fn, void \*\*memremaped\_shm,

u32 sec\_caps)

{

union {

struct arm\_smccc\_res smccc;

struct optee\_smc\_get\_shm\_config\_result result;

} res;

unsigned long vaddr;

phys\_addr\_t paddr;

size\_t size;

phys\_addr\_t begin;

phys\_addr\_t end;

void \*va;

struct tee\_shm\_pool\_mgr \*priv\_mgr;

struct tee\_shm\_pool\_mgr \*dmabuf\_mgr;

void \*rc;

invoke\_fn(OPTEE\_SMC\_GET\_SHM\_CONFIG, 0, 0, 0, 0, 0, 0, 0, &res.smccc);

if (res.result.status != OPTEE\_SMC\_RETURN\_OK) {

pr\_info("shm service not available\n");

return ERR\_PTR(-ENOENT);

}

if (res.result.settings != OPTEE\_SMC\_SHM\_CACHED) {

pr\_err("only normal cached shared memory supported\n");

return ERR\_PTR(-EINVAL);

}

begin = roundup(res.result.start, PAGE\_SIZE);

end = rounddown(res.result.start + res.result.size, PAGE\_SIZE);

paddr = begin;

size = end - begin;

if (size < 2 \* OPTEE\_SHM\_NUM\_PRIV\_PAGES \* PAGE\_SIZE) {

pr\_err("too small shared memory area\n");

return ERR\_PTR(-EINVAL);

}

va = memremap(paddr, size, MEMREMAP\_WB);

if (!va) {

pr\_err("shared memory ioremap failed\n");

return ERR\_PTR(-EINVAL);

}

vaddr = (unsigned long)va;

/\*

\* If OP-TEE can work with unregistered SHM, we will use own pool

\* for private shm

\*/

if (sec\_caps & OPTEE\_SMC\_SEC\_CAP\_DYNAMIC\_SHM) {

rc = optee\_shm\_pool\_alloc\_pages();

if (IS\_ERR(rc))

goto err\_memunmap;

priv\_mgr = rc;

} else {

const size\_t sz = OPTEE\_SHM\_NUM\_PRIV\_PAGES \* PAGE\_SIZE;

rc = tee\_shm\_pool\_mgr\_alloc\_res\_mem(vaddr, paddr, sz,

3 /\* 8 bytes aligned \*/);

if (IS\_ERR(rc))

goto err\_memunmap;

priv\_mgr = rc;

vaddr += sz;

paddr += sz;

size -= sz;

}

rc = tee\_shm\_pool\_mgr\_alloc\_res\_mem(vaddr, paddr, size, PAGE\_SHIFT);

if (IS\_ERR(rc))

goto err\_free\_priv\_mgr;

dmabuf\_mgr = rc;

rc = tee\_shm\_pool\_alloc(priv\_mgr, dmabuf\_mgr);

if (IS\_ERR(rc))

goto err\_free\_dmabuf\_mgr;

\*memremaped\_shm = va;

return rc;

err\_free\_dmabuf\_mgr:

tee\_shm\_pool\_mgr\_destroy(dmabuf\_mgr);

err\_free\_priv\_mgr:

tee\_shm\_pool\_mgr\_destroy(priv\_mgr);

err\_memunmap:

memunmap(va);

return rc;

}

/\* Simple wrapper functions to be able to use a function pointer \*/

static void optee\_smccc\_smc(unsigned long a0, unsigned long a1,

unsigned long a2, unsigned long a3,

unsigned long a4, unsigned long a5,

unsigned long a6, unsigned long a7,

struct arm\_smccc\_res \*res)

{

arm\_smccc\_smc(a0, a1, a2, a3, a4, a5, a6, a7, res);

}

static void optee\_smccc\_hvc(unsigned long a0, unsigned long a1,

unsigned long a2, unsigned long a3,

unsigned long a4, unsigned long a5,

unsigned long a6, unsigned long a7,

struct arm\_smccc\_res \*res)

{

arm\_smccc\_hvc(a0, a1, a2, a3, a4, a5, a6, a7, res);

}

static optee\_invoke\_fn \*get\_invoke\_func(struct device\_node \*np)

{

const char \*method;

pr\_info("probing for conduit method from DT.\n");

if (of\_property\_read\_string(np, "method", &method)) {

pr\_warn("missing \"method\" property\n");

return ERR\_PTR(-ENXIO);

}

if (!strcmp("hvc", method))

return optee\_smccc\_hvc;

else if (!strcmp("smc", method))

return optee\_smccc\_smc;

pr\_warn("invalid \"method\" property: %s\n", method);

return ERR\_PTR(-EINVAL);

}

static struct optee \*optee\_probe(struct device\_node \*np)

{

optee\_invoke\_fn \*invoke\_fn;

struct tee\_shm\_pool \*pool;

struct optee \*optee = NULL;

void \*memremaped\_shm = NULL;

struct tee\_device \*teedev;

u32 sec\_caps;

int rc;

invoke\_fn = get\_invoke\_func(np);

if (IS\_ERR(invoke\_fn))

return (void \*)invoke\_fn;

if (!optee\_msg\_api\_uid\_is\_optee\_api(invoke\_fn)) {

pr\_warn("api uid mismatch\n");

return ERR\_PTR(-EINVAL);

}

optee\_msg\_get\_os\_revision(invoke\_fn);

if (!optee\_msg\_api\_revision\_is\_compatible(invoke\_fn)) {

pr\_warn("api revision mismatch\n");

return ERR\_PTR(-EINVAL);

}

if (!optee\_msg\_exchange\_capabilities(invoke\_fn, &sec\_caps)) {

pr\_warn("capabilities mismatch\n");

return ERR\_PTR(-EINVAL);

}

/\*

\* We have no other option for shared memory, if secure world

\* doesn't have any reserved memory we can use we can't continue.

\*/

if (!(sec\_caps & OPTEE\_SMC\_SEC\_CAP\_HAVE\_RESERVED\_SHM))

return ERR\_PTR(-EINVAL);

pool = optee\_config\_shm\_memremap(invoke\_fn, &memremaped\_shm, sec\_caps);

if (IS\_ERR(pool))

return (void \*)pool;

optee = kzalloc(sizeof(\*optee), GFP\_KERNEL);

if (!optee) {

rc = -ENOMEM;

goto err;

}

optee->invoke\_fn = invoke\_fn;

optee->sec\_caps = sec\_caps;

teedev = tee\_device\_alloc(&optee\_desc, NULL, pool, optee);

if (IS\_ERR(teedev)) {

rc = PTR\_ERR(teedev);

goto err;

}

optee->teedev = teedev;

teedev = tee\_device\_alloc(&optee\_supp\_desc, NULL, pool, optee);

if (IS\_ERR(teedev)) {

rc = PTR\_ERR(teedev);

goto err;

}

optee->supp\_teedev = teedev;

rc = tee\_device\_register(optee->teedev);

if (rc)

goto err;

rc = tee\_device\_register(optee->supp\_teedev);

if (rc)

goto err;

mutex\_init(&optee->call\_queue.mutex);

INIT\_LIST\_HEAD(&optee->call\_queue.waiters);

optee\_wait\_queue\_init(&optee->wait\_queue);

optee\_supp\_init(&optee->supp);

optee->memremaped\_shm = memremaped\_shm;

optee->pool = pool;

optee\_enable\_shm\_cache(optee);

pr\_info("initialized driver\n");

return optee;

err:

if (optee) {

/\*

\* tee\_device\_unregister() is safe to call even if the

\* devices hasn't been registered with

\* tee\_device\_register() yet.

\*/

tee\_device\_unregister(optee->supp\_teedev);

tee\_device\_unregister(optee->teedev);

kfree(optee);

}

if (pool)

tee\_shm\_pool\_free(pool);

if (memremaped\_shm)

memunmap(memremaped\_shm);

return ERR\_PTR(rc);

}

static void optee\_remove(struct optee \*optee)

{

/\*

\* Ask OP-TEE to free all cached shared memory objects to decrease

\* reference counters and also avoid wild pointers in secure world

\* into the old shared memory range.

\*/

optee\_disable\_shm\_cache(optee);

/\*

\* The two devices has to be unregistered before we can free the

\* other resources.

\*/

tee\_device\_unregister(optee->supp\_teedev);

tee\_device\_unregister(optee->teedev);

tee\_shm\_pool\_free(optee->pool);

if (optee->memremaped\_shm)

memunmap(optee->memremaped\_shm);

optee\_wait\_queue\_exit(&optee->wait\_queue);

optee\_supp\_uninit(&optee->supp);

mutex\_destroy(&optee->call\_queue.mutex);

kfree(optee);

}

static const struct of\_device\_id optee\_match[] = {

{ .compatible = "linaro,optee-tz" },

{},

};

static struct optee \*optee\_svc;

static int \_\_init optee\_driver\_init(void)

{

struct device\_node \*fw\_np;

struct device\_node \*np;

struct optee \*optee;

/\* Node is supposed to be below /firmware \*/

fw\_np = of\_find\_node\_by\_name(NULL, "firmware");

if (!fw\_np)

return -ENODEV;

np = of\_find\_matching\_node(fw\_np, optee\_match);

if (!np)

return -ENODEV;

optee = optee\_probe(np);

of\_node\_put(np);

if (IS\_ERR(optee))

return PTR\_ERR(optee);

optee\_svc = optee;

return 0;

}

module\_init(optee\_driver\_init);

static void \_\_exit optee\_driver\_exit(void)

{

struct optee \*optee = optee\_svc;

optee\_svc = NULL;

if (optee)

optee\_remove(optee);

}

module\_exit(optee\_driver\_exit);

MODULE\_AUTHOR("Linaro");

MODULE\_DESCRIPTION("OP-TEE driver");

MODULE\_SUPPORTED\_DEVICE("");

MODULE\_VERSION("1.0");

MODULE\_LICENSE("GPL v2");

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

call.c

/\*

\* Copyright (c) 2015, Linaro Limited

\*

\* This software is licensed under the terms of the GNU General Public

\* License version 2, as published by the Free Software Foundation, and

\* may be copied, distributed, and modified under those terms.

\*

\* This program is distributed in the hope that it will be useful,

\* but WITHOUT ANY WARRANTY; without even the implied warranty of

\* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

\* GNU General Public License for more details.

\*

\*/

#include <linux/arm-smccc.h>

#include <linux/device.h>

#include <linux/err.h>

#include <linux/errno.h>

#include <linux/mm.h>

#include <linux/slab.h>

#include <linux/tee\_drv.h>

#include <linux/types.h>

#include <linux/uaccess.h>

#include "optee\_private.h"

#include "optee\_smc.h"

struct optee\_call\_waiter {

struct list\_head list\_node;

struct completion c;

};

static void optee\_cq\_wait\_init(struct optee\_call\_queue \*cq,

struct optee\_call\_waiter \*w)

{

/\*

\* We're preparing to make a call to secure world. In case we can't

\* allocate a thread in secure world we'll end up waiting in

\* optee\_cq\_wait\_for\_completion().

\*

\* Normally if there's no contention in secure world the call will

\* complete and we can cleanup directly with optee\_cq\_wait\_final().

\*/

mutex\_lock(&cq->mutex);

/\*

\* We add ourselves to the queue, but we don't wait. This

\* guarantees that we don't lose a completion if secure world

\* returns busy and another thread just exited and try to complete

\* someone.

\*/

init\_completion(&w->c);

list\_add\_tail(&w->list\_node, &cq->waiters);

mutex\_unlock(&cq->mutex);

}

static void optee\_cq\_wait\_for\_completion(struct optee\_call\_queue \*cq,

struct optee\_call\_waiter \*w)

{

wait\_for\_completion(&w->c);

mutex\_lock(&cq->mutex);

/\* Move to end of list to get out of the way for other waiters \*/

list\_del(&w->list\_node);

reinit\_completion(&w->c);

list\_add\_tail(&w->list\_node, &cq->waiters);

mutex\_unlock(&cq->mutex);

}

static void optee\_cq\_complete\_one(struct optee\_call\_queue \*cq)

{

struct optee\_call\_waiter \*w;

list\_for\_each\_entry(w, &cq->waiters, list\_node) {

if (!completion\_done(&w->c)) {

complete(&w->c);

break;

}

}

}

static void optee\_cq\_wait\_final(struct optee\_call\_queue \*cq,

struct optee\_call\_waiter \*w)

{

/\*

\* We're done with the call to secure world. The thread in secure

\* world that was used for this call is now available for some

\* other task to use.

\*/

mutex\_lock(&cq->mutex);

/\* Get out of the list \*/

list\_del(&w->list\_node);

/\* Wake up one eventual waiting task \*/

optee\_cq\_complete\_one(cq);

/\*

\* If we're completed we've got a completion from another task that

\* was just done with its call to secure world. Since yet another

\* thread now is available in secure world wake up another eventual

\* waiting task.

\*/

if (completion\_done(&w->c))

optee\_cq\_complete\_one(cq);

mutex\_unlock(&cq->mutex);

}

/\* Requires the filpstate mutex to be held \*/

static struct optee\_session \*find\_session(struct optee\_context\_data \*ctxdata,

u32 session\_id)

{

struct optee\_session \*sess;

list\_for\_each\_entry(sess, &ctxdata->sess\_list, list\_node)

if (sess->session\_id == session\_id)

return sess;

return NULL;

}

/\*\*

\* optee\_do\_call\_with\_arg() - Do an SMC to OP-TEE in secure world

\* @ctx: calling context

\* @parg: physical address of message to pass to secure world

\*

\* Does and SMC to OP-TEE in secure world and handles eventual resulting

\* Remote Procedure Calls (RPC) from OP-TEE.

\*

\* Returns return code from secure world, 0 is OK

\*/

u32 optee\_do\_call\_with\_arg(struct tee\_context \*ctx, phys\_addr\_t parg)

{

struct optee \*optee = tee\_get\_drvdata(ctx->teedev);

struct optee\_call\_waiter w;

struct optee\_rpc\_param param = { };

struct optee\_call\_ctx call\_ctx = { };

u32 ret;

param.a0 = OPTEE\_SMC\_CALL\_WITH\_ARG;

reg\_pair\_from\_64(&param.a1, &param.a2, parg);

/\* Initialize waiter \*/

optee\_cq\_wait\_init(&optee->call\_queue, &w);

while (true) {

struct arm\_smccc\_res res;

optee->invoke\_fn(param.a0, param.a1, param.a2, param.a3,

param.a4, param.a5, param.a6, param.a7,

&res);

if (res.a0 == OPTEE\_SMC\_RETURN\_ETHREAD\_LIMIT) {

/\*

\* Out of threads in secure world, wait for a thread

\* become available.

\*/

optee\_cq\_wait\_for\_completion(&optee->call\_queue, &w);

} else if (OPTEE\_SMC\_RETURN\_IS\_RPC(res.a0)) {

param.a0 = res.a0;

param.a1 = res.a1;

param.a2 = res.a2;

param.a3 = res.a3;

optee\_handle\_rpc(ctx, &param, &call\_ctx);

} else {

ret = res.a0;

break;

}

}

optee\_rpc\_finalize\_call(&call\_ctx);

/\*

\* We're done with our thread in secure world, if there's any

\* thread waiters wake up one.

\*/

optee\_cq\_wait\_final(&optee->call\_queue, &w);

return ret;

}

static struct tee\_shm \*get\_msg\_arg(struct tee\_context \*ctx, size\_t num\_params,

struct optee\_msg\_arg \*\*msg\_arg,

phys\_addr\_t \*msg\_parg)

{

int rc;

struct tee\_shm \*shm;

struct optee\_msg\_arg \*ma;/\*

\* Copyright (c) 2015, Linaro Limited

\*

\* This software is licensed under the terms of the GNU General Public

\* License version 2, as published by the Free Software Foundation, and

\* may be copied, distributed, and modified under those terms.

\*

\* This program is distributed in the hope that it will be useful,

\* but WITHOUT ANY WARRANTY; without even the implied warranty of

\* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

\* GNU General Public License for more details.

\*

\*/

#include <linux/arm-smccc.h>

#include <linux/device.h>

#include <linux/err.h>

#include <linux/errno.h>

#include <linux/mm.h>

#include <linux/slab.h>

#include <linux/tee\_drv.h>

#include <linux/types.h>

#include <linux/uaccess.h>

#include "optee\_private.h"

#include "optee\_smc.h"

struct optee\_call\_waiter {

struct list\_head list\_node;

struct completion c;

};

static void optee\_cq\_wait\_init(struct optee\_call\_queue \*cq,

struct optee\_call\_waiter \*w)

{

/\*

\* We're preparing to make a call to secure world. In case we can't

\* allocate a thread in secure world we'll end up waiting in

\* optee\_cq\_wait\_for\_completion().

\*

\* Normally if there's no contention in secure world the call will

\* complete and we can cleanup directly with optee\_cq\_wait\_final().

\*/

mutex\_lock(&cq->mutex);

/\*

\* We add ourselves to the queue, but we don't wait. This

\* guarantees that we don't lose a completion if secure world

\* returns busy and another thread just exited and try to complete

\* someone.

\*/

init\_completion(&w->c);

list\_add\_tail(&w->list\_node, &cq->waiters);

mutex\_unlock(&cq->mutex);

}

static void optee\_cq\_wait\_for\_completion(struct optee\_call\_queue \*cq,

struct optee\_call\_waiter \*w)

{

wait\_for\_completion(&w->c);

mutex\_lock(&cq->mutex);

/\* Move to end of list to get out of the way for other waiters \*/

list\_del(&w->list\_node);

reinit\_completion(&w->c);

list\_add\_tail(&w->list\_node, &cq->waiters);

mutex\_unlock(&cq->mutex);

}

static void optee\_cq\_complete\_one(struct optee\_call\_queue \*cq)

{

struct optee\_call\_waiter \*w;

list\_for\_each\_entry(w, &cq->waiters, list\_node) {

if (!completion\_done(&w->c)) {

complete(&w->c);

break;

}

}

}

static void optee\_cq\_wait\_final(struct optee\_call\_queue \*cq,

struct optee\_call\_waiter \*w)

{

/\*

\* We're done with the call to secure world. The thread in secure

\* world that was used for this call is now available for some

\* other task to use.

\*/

mutex\_lock(&cq->mutex);

/\* Get out of the list \*/

list\_del(&w->list\_node);

/\* Wake up one eventual waiting task \*/

optee\_cq\_complete\_one(cq);

/\*

\* If we're completed we've got a completion from another task that

\* was just done with its call to secure world. Since yet another

\* thread now is available in secure world wake up another eventual

\* waiting task.

\*/

if (completion\_done(&w->c))

optee\_cq\_complete\_one(cq);

mutex\_unlock(&cq->mutex);

}

/\* Requires the filpstate mutex to be held \*/

static struct optee\_session \*find\_session(struct optee\_context\_data \*ctxdata,

u32 session\_id)

{

struct optee\_session \*sess;

list\_for\_each\_entry(sess, &ctxdata->sess\_list, list\_node)

if (sess->session\_id == session\_id)

return sess;

return NULL;

}

/\*\*

\* optee\_do\_call\_with\_arg() - Do an SMC to OP-TEE in secure world

\* @ctx: calling context

\* @parg: physical address of message to pass to secure world

\*

\* Does and SMC to OP-TEE in secure world and handles eventual resulting

\* Remote Procedure Calls (RPC) from OP-TEE.

\*

\* Returns return code from secure world, 0 is OK

\*/

u32 optee\_do\_call\_with\_arg(struct tee\_context \*ctx, phys\_addr\_t parg)

{

struct optee \*optee = tee\_get\_drvdata(ctx->teedev);

struct optee\_call\_waiter w;

struct optee\_rpc\_param param = { };

struct optee\_call\_ctx call\_ctx = { };

u32 ret;

param.a0 = OPTEE\_SMC\_CALL\_WITH\_ARG;

reg\_pair\_from\_64(&param.a1, &param.a2, parg);

/\* Initialize waiter \*/

optee\_cq\_wait\_init(&optee->call\_queue, &w);

while (true) {

struct arm\_smccc\_res res;

optee->invoke\_fn(param.a0, param.a1, param.a2, param.a3,

param.a4, param.a5, param.a6, param.a7,

&res);

if (res.a0 == OPTEE\_SMC\_RETURN\_ETHREAD\_LIMIT) {

/\*

\* Out of threads in secure world, wait for a thread

\* become available.

\*/

optee\_cq\_wait\_for\_completion(&optee->call\_queue, &w);

} else if (OPTEE\_SMC\_RETURN\_IS\_RPC(res.a0)) {

param.a0 = res.a0;

param.a1 = res.a1;

param.a2 = res.a2;

param.a3 = res.a3;

optee\_handle\_rpc(ctx, &param, &call\_ctx);

} else {

ret = res.a0;

break;

}

}

optee\_rpc\_finalize\_call(&call\_ctx);

/\*

\* We're done with our thread in secure world, if there's any

\* thread waiters wake up one.

\*/

optee\_cq\_wait\_final(&optee->call\_queue, &w);

return ret;

}

static struct tee\_shm \*get\_msg\_arg(struct tee\_context \*ctx, size\_t num\_params,

struct optee\_msg\_arg \*\*msg\_arg,

phys\_addr\_t \*msg\_parg)

{

int rc;

struct tee\_shm \*shm;

struct optee\_msg\_arg \*ma;

shm = tee\_shm\_alloc(ctx, OPTEE\_MSG\_GET\_ARG\_SIZE(num\_params),

TEE\_SHM\_MAPPED);

if (IS\_ERR(shm))

return shm;

ma = tee\_shm\_get\_va(shm, 0);

if (IS\_ERR(ma)) {

rc = PTR\_ERR(ma);

goto out;

}

rc = tee\_shm\_get\_pa(shm, 0, msg\_parg);

if (rc)

goto out;

memset(ma, 0, OPTEE\_MSG\_GET\_ARG\_SIZE(num\_params));

ma->num\_params = num\_params;

\*msg\_arg = ma;

out:

if (rc) {

tee\_shm\_free(shm);

return ERR\_PTR(rc);

}

return shm;

}

int optee\_open\_session(struct tee\_context \*ctx,

struct tee\_ioctl\_open\_session\_arg \*arg,

struct tee\_param \*param)

{

struct optee\_context\_data \*ctxdata = ctx->data;

int rc;

struct tee\_shm \*shm;

struct optee\_msg\_arg \*msg\_arg;

phys\_addr\_t msg\_parg;

struct optee\_session \*sess = NULL;

/\* +2 for the meta parameters added below \*/

shm = get\_msg\_arg(ctx, arg->num\_params + 2, &msg\_arg, &msg\_parg);

if (IS\_ERR(shm))

return PTR\_ERR(shm);

msg\_arg->cmd = OPTEE\_MSG\_CMD\_OPEN\_SESSION;

msg\_arg->cancel\_id = arg->cancel\_id;

/\*

\* Initialize and add the meta parameters needed when opening a

\* session.

\*/

msg\_arg->params[0].attr = OPTEE\_MSG\_ATTR\_TYPE\_VALUE\_INPUT |

OPTEE\_MSG\_ATTR\_META;

msg\_arg->params[1].attr = OPTEE\_MSG\_ATTR\_TYPE\_VALUE\_INPUT |

OPTEE\_MSG\_ATTR\_META;

memcpy(&msg\_arg->params[0].u.value, arg->uuid, sizeof(arg->uuid));

memcpy(&msg\_arg->params[1].u.value, arg->uuid, sizeof(arg->clnt\_uuid));

msg\_arg->params[1].u.value.c = arg->clnt\_login;

rc = optee\_to\_msg\_param(msg\_arg->params + 2, arg->num\_params, param);

if (rc)

goto out;

sess = kzalloc(sizeof(\*sess), GFP\_KERNEL);

if (!sess) {

rc = -ENOMEM;

goto out;

}

if (optee\_do\_call\_with\_arg(ctx, msg\_parg)) {

msg\_arg->ret = TEEC\_ERROR\_COMMUNICATION;

msg\_arg->ret\_origin = TEEC\_ORIGIN\_COMMS;

}

if (msg\_arg->ret == TEEC\_SUCCESS) {

/\* A new session has been created, add it to the list. \*/

sess->session\_id = msg\_arg->session;

mutex\_lock(&ctxdata->mutex);

list\_add(&sess->list\_node, &ctxdata->sess\_list);

mutex\_unlock(&ctxdata->mutex);

} else {

kfree(sess);

}

if (optee\_from\_msg\_param(param, arg->num\_params, msg\_arg->params + 2)) {

arg->ret = TEEC\_ERROR\_COMMUNICATION;

arg->ret\_origin = TEEC\_ORIGIN\_COMMS;

/\* Close session again to avoid leakage \*/

optee\_close\_session(ctx, msg\_arg->session);

} else {

arg->session = msg\_arg->session;

arg->ret = msg\_arg->ret;

arg->ret\_origin = msg\_arg->ret\_origin;

}

out:

tee\_shm\_free(shm);

return rc;

}

int optee\_close\_session(struct tee\_context \*ctx, u32 session)

{

struct optee\_context\_data \*ctxdata = ctx->data;

struct tee\_shm \*shm;

struct optee\_msg\_arg \*msg\_arg;

phys\_addr\_t msg\_parg;

struct optee\_session \*sess;

/\* Check that the session is valid and remove it from the list \*/

mutex\_lock(&ctxdata->mutex);

sess = find\_session(ctxdata, session);

if (sess)

list\_del(&sess->list\_node);

mutex\_unlock(&ctxdata->mutex);

if (!sess)

return -EINVAL;

kfree(sess);

shm = get\_msg\_arg(ctx, 0, &msg\_arg, &msg\_parg);

if (IS\_ERR(shm))

return PTR\_ERR(shm);

msg\_arg->cmd = OPTEE\_MSG\_CMD\_CLOSE\_SESSION;

msg\_arg->session = session;

optee\_do\_call\_with\_arg(ctx, msg\_parg);

tee\_shm\_free(shm);

return 0;

}

int optee\_invoke\_func(struct tee\_context \*ctx, struct tee\_ioctl\_invoke\_arg \*arg,

struct tee\_param \*param)

{

struct optee\_context\_data \*ctxdata = ctx->data;

struct tee\_shm \*shm;

struct optee\_msg\_arg \*msg\_arg;

phys\_addr\_t msg\_parg;

struct optee\_session \*sess;

int rc;

/\* Check that the session is valid \*/

mutex\_lock(&ctxdata->mutex);

sess = find\_session(ctxdata, arg->session);

mutex\_unlock(&ctxdata->mutex);

if (!sess)

return -EINVAL;

shm = get\_msg\_arg(ctx, arg->num\_params, &msg\_arg, &msg\_parg);

if (IS\_ERR(shm))

return PTR\_ERR(shm);

msg\_arg->cmd = OPTEE\_MSG\_CMD\_INVOKE\_COMMAND;

msg\_arg->func = arg->func;

msg\_arg->session = arg->session;

msg\_arg->cancel\_id = arg->cancel\_id;

rc = optee\_to\_msg\_param(msg\_arg->params, arg->num\_params, param);

if (rc)

goto out;

if (optee\_do\_call\_with\_arg(ctx, msg\_parg)) {

msg\_arg->ret = TEEC\_ERROR\_COMMUNICATION;

msg\_arg->ret\_origin = TEEC\_ORIGIN\_COMMS;

}

if (optee\_from\_msg\_param(param, arg->num\_params, msg\_arg->params)) {

msg\_arg->ret = TEEC\_ERROR\_COMMUNICATION;

msg\_arg->ret\_origin = TEEC\_ORIGIN\_COMMS;

}

arg->ret = msg\_arg->ret;

arg->ret\_origin = msg\_arg->ret\_origin;

out:

tee\_shm\_free(shm);

return rc;

}

int optee\_cancel\_req(struct tee\_context \*ctx, u32 cancel\_id, u32 session)

{

struct optee\_context\_data \*ctxdata = ctx->data;

struct tee\_shm \*shm;

struct optee\_msg\_arg \*msg\_arg;

phys\_addr\_t msg\_parg;

struct optee\_session \*sess;

/\* Check that the session is valid \*/

mutex\_lock(&ctxdata->mutex);

sess = find\_session(ctxdata, session);

mutex\_unlock(&ctxdata->mutex);

if (!sess)

return -EINVAL;

shm = get\_msg\_arg(ctx, 0, &msg\_arg, &msg\_parg);

if (IS\_ERR(shm))

return PTR\_ERR(shm);

msg\_arg->cmd = OPTEE\_MSG\_CMD\_CANCEL;

msg\_arg->session = session;

msg\_arg->cancel\_id = cancel\_id;

optee\_do\_call\_with\_arg(ctx, msg\_parg);

tee\_shm\_free(shm);

return 0;

}

/\*\*

\* optee\_enable\_shm\_cache() - Enables caching of some shared memory allocation

\* in OP-TEE

\* @optee: main service struct

\*/

void optee\_enable\_shm\_cache(struct optee \*optee)

{

struct optee\_call\_waiter w;

/\* We need to retry until secure world isn't busy. \*/

optee\_cq\_wait\_init(&optee->call\_queue, &w);

while (true) {

struct arm\_smccc\_res res;

optee->invoke\_fn(OPTEE\_SMC\_ENABLE\_SHM\_CACHE, 0, 0, 0, 0, 0, 0,

0, &res);

if (res.a0 == OPTEE\_SMC\_RETURN\_OK)

break;

optee\_cq\_wait\_for\_completion(&optee->call\_queue, &w);

}

optee\_cq\_wait\_final(&optee->call\_queue, &w);

}

/\*\*

\* optee\_disable\_shm\_cache() - Disables caching of some shared memory allocation

\* in OP-TEE

\* @optee: main service struct

\*/

void optee\_disable\_shm\_cache(struct optee \*optee)

{

struct optee\_call\_waiter w;

/\* We need to retry until secure world isn't busy. \*/

optee\_cq\_wait\_init(&optee->call\_queue, &w);

while (true) {

union {

struct arm\_smccc\_res smccc;

struct optee\_smc\_disable\_shm\_cache\_result result;

} res;

optee->invoke\_fn(OPTEE\_SMC\_DISABLE\_SHM\_CACHE, 0, 0, 0, 0, 0, 0,

0, &res.smccc);

if (res.result.status == OPTEE\_SMC\_RETURN\_ENOTAVAIL)

break; /\* All shm's freed \*/

if (res.result.status == OPTEE\_SMC\_RETURN\_OK) {

struct tee\_shm \*shm;

shm = reg\_pair\_to\_ptr(res.result.shm\_upper32,

res.result.shm\_lower32);

tee\_shm\_free(shm);

} else {

optee\_cq\_wait\_for\_completion(&optee->call\_queue, &w);

}

}

optee\_cq\_wait\_final(&optee->call\_queue, &w);

}

#define PAGELIST\_ENTRIES\_PER\_PAGE \

((OPTEE\_MSG\_NONCONTIG\_PAGE\_SIZE / sizeof(u64)) - 1)

/\*\*

\* optee\_fill\_pages\_list() - write list of user pages to given shared

\* buffer.

\*

\* @dst: page-aligned buffer where list of pages will be stored

\* @pages: array of pages that represents shared buffer

\* @num\_pages: number of entries in @pages

\* @page\_offset: offset of user buffer from page start

\*

\* @dst should be big enough to hold list of user page addresses and

\* links to the next pages of buffer

\*/

void optee\_fill\_pages\_list(u64 \*dst, struct page \*\*pages, int num\_pages,

size\_t page\_offset)

{

int n = 0;

phys\_addr\_t optee\_page;

/\*

\* Refer to OPTEE\_MSG\_ATTR\_NONCONTIG description in optee\_msg.h

\* for details.

\*/

struct {

u64 pages\_list[PAGELIST\_ENTRIES\_PER\_PAGE];

u64 next\_page\_data;

} \*pages\_data;

/\*

\* Currently OP-TEE uses 4k page size and it does not looks

\* like this will change in the future. On other hand, there are

\* no know ARM architectures with page size < 4k.

\* Thus the next built assert looks redundant. But the following

\* code heavily relies on this assumption, so it is better be

\* safe than sorry.

\*/

BUILD\_BUG\_ON(PAGE\_SIZE < OPTEE\_MSG\_NONCONTIG\_PAGE\_SIZE);

pages\_data = (void \*)dst;

/\*

\* If linux page is bigger than 4k, and user buffer offset is

\* larger than 4k/8k/12k/etc this will skip first 4k pages,

\* because they bear no value data for OP-TEE.

\*/

optee\_page = page\_to\_phys(\*pages) +

round\_down(page\_offset, OPTEE\_MSG\_NONCONTIG\_PAGE\_SIZE);

while (true) {

pages\_data->pages\_list[n++] = optee\_page;

if (n == PAGELIST\_ENTRIES\_PER\_PAGE) {

pages\_data->next\_page\_data =

virt\_to\_phys(pages\_data + 1);

pages\_data++;

n = 0;

}

optee\_page += OPTEE\_MSG\_NONCONTIG\_PAGE\_SIZE;

if (!(optee\_page & ~PAGE\_MASK)) {

if (!--num\_pages)

break;

pages++;

optee\_page = page\_to\_phys(\*pages);

}

}

}

/\*

\* The final entry in each pagelist page is a pointer to the next

\* pagelist page.

\*/

static size\_t get\_pages\_list\_size(size\_t num\_entries)

{

int pages = DIV\_ROUND\_UP(num\_entries, PAGELIST\_ENTRIES\_PER\_PAGE);

return pages \* OPTEE\_MSG\_NONCONTIG\_PAGE\_SIZE;

}

u64 \*optee\_allocate\_pages\_list(size\_t num\_entries)

{

return alloc\_pages\_exact(get\_pages\_list\_size(num\_entries), GFP\_KERNEL);

}

void optee\_free\_pages\_list(void \*list, size\_t num\_entries)

{

free\_pages\_exact(list, get\_pages\_list\_size(num\_entries));

}

static bool is\_normal\_memory(pgprot\_t p)

{

#if defined(CONFIG\_ARM)

return (pgprot\_val(p) & L\_PTE\_MT\_MASK) == L\_PTE\_MT\_WRITEALLOC;

#elif defined(CONFIG\_ARM64)

return (pgprot\_val(p) & PTE\_ATTRINDX\_MASK) == PTE\_ATTRINDX(MT\_NORMAL);

#else

#error "Unuspported architecture"

#endif

}

static int \_\_check\_mem\_type(struct vm\_area\_struct \*vma, unsigned long end)

{

while (vma && is\_normal\_memory(vma->vm\_page\_prot)) {

if (vma->vm\_end >= end)

return 0;

vma = vma->vm\_next;

}

return -EINVAL;

}

static int check\_mem\_type(unsigned long start, size\_t num\_pages)

{

struct mm\_struct \*mm = current->mm;

int rc;

down\_read(&mm->mmap\_sem);

rc = \_\_check\_mem\_type(find\_vma(mm, start),

start + num\_pages \* PAGE\_SIZE);

up\_read(&mm->mmap\_sem);

return rc;

}

int optee\_shm\_register(struct tee\_context \*ctx, struct tee\_shm \*shm,

struct page \*\*pages, size\_t num\_pages,

unsigned long start)

{

struct tee\_shm \*shm\_arg = NULL;

struct optee\_msg\_arg \*msg\_arg;

u64 \*pages\_list;

phys\_addr\_t msg\_parg;

int rc;

if (!num\_pages)

return -EINVAL;

rc = check\_mem\_type(start, num\_pages);

if (rc)

return rc;

pages\_list = optee\_allocate\_pages\_list(num\_pages);

if (!pages\_list)

return -ENOMEM;

shm\_arg = get\_msg\_arg(ctx, 1, &msg\_arg, &msg\_parg);

if (IS\_ERR(shm\_arg)) {

rc = PTR\_ERR(shm\_arg);

goto out;

}

optee\_fill\_pages\_list(pages\_list, pages, num\_pages,

tee\_shm\_get\_page\_offset(shm));

msg\_arg->cmd = OPTEE\_MSG\_CMD\_REGISTER\_SHM;

msg\_arg->params->attr = OPTEE\_MSG\_ATTR\_TYPE\_TMEM\_OUTPUT |

OPTEE\_MSG\_ATTR\_NONCONTIG;

msg\_arg->params->u.tmem.shm\_ref = (unsigned long)shm;

msg\_arg->params->u.tmem.size = tee\_shm\_get\_size(shm);

/\*

\* In the least bits of msg\_arg->params->u.tmem.buf\_ptr we

\* store buffer offset from 4k page, as described in OP-TEE ABI.

\*/

msg\_arg->params->u.tmem.buf\_ptr = virt\_to\_phys(pages\_list) |

(tee\_shm\_get\_page\_offset(shm) & (OPTEE\_MSG\_NONCONTIG\_PAGE\_SIZE - 1));

if (optee\_do\_call\_with\_arg(ctx, msg\_parg) ||

msg\_arg->ret != TEEC\_SUCCESS)

rc = -EINVAL;

tee\_shm\_free(shm\_arg);

out:

optee\_free\_pages\_list(pages\_list, num\_pages);

return rc;

}

int optee\_shm\_unregister(struct tee\_context \*ctx, struct tee\_shm \*shm)

{

struct tee\_shm \*shm\_arg;

struct optee\_msg\_arg \*msg\_arg;

phys\_addr\_t msg\_parg;

int rc = 0;

shm\_arg = get\_msg\_arg(ctx, 1, &msg\_arg, &msg\_parg);

if (IS\_ERR(shm\_arg))

return PTR\_ERR(shm\_arg);

msg\_arg->cmd = OPTEE\_MSG\_CMD\_UNREGISTER\_SHM;

msg\_arg->params[0].attr = OPTEE\_MSG\_ATTR\_TYPE\_RMEM\_INPUT;

msg\_arg->params[0].u.rmem.shm\_ref = (unsigned long)shm;

if (optee\_do\_call\_with\_arg(ctx, msg\_parg) ||

msg\_arg->ret != TEEC\_SUCCESS)

rc = -EINVAL;

tee\_shm\_free(shm\_arg);

return rc;

}

int optee\_shm\_register\_supp(struct tee\_context \*ctx, struct tee\_shm \*shm,

struct page \*\*pages, size\_t num\_pages,

unsigned long start)

{

/\*

\* We don't want to register supplicant memory in OP-TEE.

\* Instead information about it will be passed in RPC code.

\*/

return check\_mem\_type(start, num\_pages);

}

int optee\_shm\_unregister\_supp(struct tee\_context \*ctx, struct tee\_shm \*shm)

{

return 0;

}

shm = tee\_shm\_alloc(ctx, OPTEE\_MSG\_GET\_ARG\_SIZE(num\_params),

TEE\_SHM\_MAPPED);

if (IS\_ERR(shm))

return shm;

ma = tee\_shm\_get\_va(shm, 0);

if (IS\_ERR(ma)) {

rc = PTR\_ERR(ma);

goto out;

}

rc = tee\_shm\_get\_pa(shm, 0, msg\_parg);

if (rc)

goto out;

memset(ma, 0, OPTEE\_MSG\_GET\_ARG\_SIZE(num\_params));

ma->num\_params = num\_params;

\*msg\_arg = ma;

out:

if (rc) {

tee\_shm\_free(shm);

return ERR\_PTR(rc);

}

return shm;

}

int optee\_open\_session(struct tee\_context \*ctx,

struct tee\_ioctl\_open\_session\_arg \*arg,

struct tee\_param \*param)

{

struct optee\_context\_data \*ctxdata = ctx->data;

int rc;

struct tee\_shm \*shm;

struct optee\_msg\_arg \*msg\_arg;

phys\_addr\_t msg\_parg;

struct optee\_session \*sess = NULL;

/\* +2 for the meta parameters added below \*/

shm = get\_msg\_arg(ctx, arg->num\_params + 2, &msg\_arg, &msg\_parg);

if (IS\_ERR(shm))

return PTR\_ERR(shm);

msg\_arg->cmd = OPTEE\_MSG\_CMD\_OPEN\_SESSION;

msg\_arg->cancel\_id = arg->cancel\_id;

/\*

\* Initialize and add the meta parameters needed when opening a

\* session.

\*/

msg\_arg->params[0].attr = OPTEE\_MSG\_ATTR\_TYPE\_VALUE\_INPUT |

OPTEE\_MSG\_ATTR\_META;

msg\_arg->params[1].attr = OPTEE\_MSG\_ATTR\_TYPE\_VALUE\_INPUT |

OPTEE\_MSG\_ATTR\_META;

memcpy(&msg\_arg->params[0].u.value, arg->uuid, sizeof(arg->uuid));

memcpy(&msg\_arg->params[1].u.value, arg->uuid, sizeof(arg->clnt\_uuid));

msg\_arg->params[1].u.value.c = arg->clnt\_login;

rc = optee\_to\_msg\_param(msg\_arg->params + 2, arg->num\_params, param);

if (rc)

goto out;

sess = kzalloc(sizeof(\*sess), GFP\_KERNEL);

if (!sess) {

rc = -ENOMEM;

goto out;

}

if (optee\_do\_call\_with\_arg(ctx, msg\_parg)) {

msg\_arg->ret = TEEC\_ERROR\_COMMUNICATION;

msg\_arg->ret\_origin = TEEC\_ORIGIN\_COMMS;

}

if (msg\_arg->ret == TEEC\_SUCCESS) {

/\* A new session has been created, add it to the list. \*/

sess->session\_id = msg\_arg->session;

mutex\_lock(&ctxdata->mutex);

list\_add(&sess->list\_node, &ctxdata->sess\_list);

mutex\_unlock(&ctxdata->mutex);

} else {

kfree(sess);

}

if (optee\_from\_msg\_param(param, arg->num\_params, msg\_arg->params + 2)) {

arg->ret = TEEC\_ERROR\_COMMUNICATION;

arg->ret\_origin = TEEC\_ORIGIN\_COMMS;

/\* Close session again to avoid leakage \*/

optee\_close\_session(ctx, msg\_arg->session);

} else {

arg->session = msg\_arg->session;

arg->ret = msg\_arg->ret;

arg->ret\_origin = msg\_arg->ret\_origin;

}

out:

tee\_shm\_free(shm);

return rc;

}

int optee\_close\_session(struct tee\_context \*ctx, u32 session)

{

struct optee\_context\_data \*ctxdata = ctx->data;

struct tee\_shm \*shm;

struct optee\_msg\_arg \*msg\_arg;

phys\_addr\_t msg\_parg;

struct optee\_session \*sess;

/\* Check that the session is valid and remove it from the list \*/

mutex\_lock(&ctxdata->mutex);

sess = find\_session(ctxdata, session);

if (sess)

list\_del(&sess->list\_node);

mutex\_unlock(&ctxdata->mutex);

if (!sess)

return -EINVAL;

kfree(sess);

shm = get\_msg\_arg(ctx, 0, &msg\_arg, &msg\_parg);

if (IS\_ERR(shm))

return PTR\_ERR(shm);

msg\_arg->cmd = OPTEE\_MSG\_CMD\_CLOSE\_SESSION;

msg\_arg->session = session;

optee\_do\_call\_with\_arg(ctx, msg\_parg);

tee\_shm\_free(shm);

return 0;

}

int optee\_invoke\_func(struct tee\_context \*ctx, struct tee\_ioctl\_invoke\_arg \*arg,

struct tee\_param \*param)

{

struct optee\_context\_data \*ctxdata = ctx->data;

struct tee\_shm \*shm;

struct optee\_msg\_arg \*msg\_arg;

phys\_addr\_t msg\_parg;

struct optee\_session \*sess;

int rc;

/\* Check that the session is valid \*/

mutex\_lock(&ctxdata->mutex);

sess = find\_session(ctxdata, arg->session);

mutex\_unlock(&ctxdata->mutex);

if (!sess)

return -EINVAL;

shm = get\_msg\_arg(ctx, arg->num\_params, &msg\_arg, &msg\_parg);

if (IS\_ERR(shm))

return PTR\_ERR(shm);

msg\_arg->cmd = OPTEE\_MSG\_CMD\_INVOKE\_COMMAND;

msg\_arg->func = arg->func;

msg\_arg->session = arg->session;

msg\_arg->cancel\_id = arg->cancel\_id;

rc = optee\_to\_msg\_param(msg\_arg->params, arg->num\_params, param);

if (rc)

goto out;

if (optee\_do\_call\_with\_arg(ctx, msg\_parg)) {

msg\_arg->ret = TEEC\_ERROR\_COMMUNICATION;

msg\_arg->ret\_origin = TEEC\_ORIGIN\_COMMS;

}

if (optee\_from\_msg\_param(param, arg->num\_params, msg\_arg->params)) {

msg\_arg->ret = TEEC\_ERROR\_COMMUNICATION;

msg\_arg->ret\_origin = TEEC\_ORIGIN\_COMMS;

}

arg->ret = msg\_arg->ret;

arg->ret\_origin = msg\_arg->ret\_origin;

out:

tee\_shm\_free(shm);

return rc;

}

int optee\_cancel\_req(struct tee\_context \*ctx, u32 cancel\_id, u32 session)

{

struct optee\_context\_data \*ctxdata = ctx->data;

struct tee\_shm \*shm;

struct optee\_msg\_arg \*msg\_arg;

phys\_addr\_t msg\_parg;

struct optee\_session \*sess;

/\* Check that the session is valid \*/

mutex\_lock(&ctxdata->mutex);

sess = find\_session(ctxdata, session);

mutex\_unlock(&ctxdata->mutex);

if (!sess)

return -EINVAL;

shm = get\_msg\_arg(ctx, 0, &msg\_arg, &msg\_parg);

if (IS\_ERR(shm))

return PTR\_ERR(shm);

msg\_arg->cmd = OPTEE\_MSG\_CMD\_CANCEL;

msg\_arg->session = session;

msg\_arg->cancel\_id = cancel\_id;

optee\_do\_call\_with\_arg(ctx, msg\_parg);

tee\_shm\_free(shm);

return 0;

}

/\*\*

\* optee\_enable\_shm\_cache() - Enables caching of some shared memory allocation

\* in OP-TEE

\* @optee: main service struct

\*/

void optee\_enable\_shm\_cache(struct optee \*optee)

{

struct optee\_call\_waiter w;

/\* We need to retry until secure world isn't busy. \*/

optee\_cq\_wait\_init(&optee->call\_queue, &w);

while (true) {

struct arm\_smccc\_res res;

optee->invoke\_fn(OPTEE\_SMC\_ENABLE\_SHM\_CACHE, 0, 0, 0, 0, 0, 0,

0, &res);

if (res.a0 == OPTEE\_SMC\_RETURN\_OK)

break;

optee\_cq\_wait\_for\_completion(&optee->call\_queue, &w);

}

optee\_cq\_wait\_final(&optee->call\_queue, &w);

}

/\*\*

\* optee\_disable\_shm\_cache() - Disables caching of some shared memory allocation

\* in OP-TEE

\* @optee: main service struct

\*/

void optee\_disable\_shm\_cache(struct optee \*optee)

{

struct optee\_call\_waiter w;

/\* We need to retry until secure world isn't busy. \*/

optee\_cq\_wait\_init(&optee->call\_queue, &w);

while (true) {

union {

struct arm\_smccc\_res smccc;

struct optee\_smc\_disable\_shm\_cache\_result result;

} res;

optee->invoke\_fn(OPTEE\_SMC\_DISABLE\_SHM\_CACHE, 0, 0, 0, 0, 0, 0,

0, &res.smccc);

if (res.result.status == OPTEE\_SMC\_RETURN\_ENOTAVAIL)

break; /\* All shm's freed \*/

if (res.result.status == OPTEE\_SMC\_RETURN\_OK) {

struct tee\_shm \*shm;

shm = reg\_pair\_to\_ptr(res.result.shm\_upper32,

res.result.shm\_lower32);

tee\_shm\_free(shm);

} else {

optee\_cq\_wait\_for\_completion(&optee->call\_queue, &w);

}

}

optee\_cq\_wait\_final(&optee->call\_queue, &w);

}

#define PAGELIST\_ENTRIES\_PER\_PAGE \

((OPTEE\_MSG\_NONCONTIG\_PAGE\_SIZE / sizeof(u64)) - 1)

/\*\*

\* optee\_fill\_pages\_list() - write list of user pages to given shared

\* buffer.

\*

\* @dst: page-aligned buffer where list of pages will be stored

\* @pages: array of pages that represents shared buffer

\* @num\_pages: number of entries in @pages

\* @page\_offset: offset of user buffer from page start

\*

\* @dst should be big enough to hold list of user page addresses and

\* links to the next pages of buffer

\*/

void optee\_fill\_pages\_list(u64 \*dst, struct page \*\*pages, int num\_pages,

size\_t page\_offset)

{

int n = 0;

phys\_addr\_t optee\_page;

/\*

\* Refer to OPTEE\_MSG\_ATTR\_NONCONTIG description in optee\_msg.h

\* for details.

\*/

struct {

u64 pages\_list[PAGELIST\_ENTRIES\_PER\_PAGE];

u64 next\_page\_data;

} \*pages\_data;

/\*

\* Currently OP-TEE uses 4k page size and it does not looks

\* like this will change in the future. On other hand, there are

\* no know ARM architectures with page size < 4k.

\* Thus the next built assert looks redundant. But the following

\* code heavily relies on this assumption, so it is better be

\* safe than sorry.

\*/

BUILD\_BUG\_ON(PAGE\_SIZE < OPTEE\_MSG\_NONCONTIG\_PAGE\_SIZE);

pages\_data = (void \*)dst;

/\*

\* If linux page is bigger than 4k, and user buffer offset is

\* larger than 4k/8k/12k/etc this will skip first 4k pages,

\* because they bear no value data for OP-TEE.

\*/

optee\_page = page\_to\_phys(\*pages) +

round\_down(page\_offset, OPTEE\_MSG\_NONCONTIG\_PAGE\_SIZE);

while (true) {

pages\_data->pages\_list[n++] = optee\_page;

if (n == PAGELIST\_ENTRIES\_PER\_PAGE) {

pages\_data->next\_page\_data =

virt\_to\_phys(pages\_data + 1);

pages\_data++;

n = 0;

}

optee\_page += OPTEE\_MSG\_NONCONTIG\_PAGE\_SIZE;

if (!(optee\_page & ~PAGE\_MASK)) {

if (!--num\_pages)

break;

pages++;

optee\_page = page\_to\_phys(\*pages);

}

}

}

/\*

\* The final entry in each pagelist page is a pointer to the next

\* pagelist page.

\*/

static size\_t get\_pages\_list\_size(size\_t num\_entries)

{

int pages = DIV\_ROUND\_UP(num\_entries, PAGELIST\_ENTRIES\_PER\_PAGE);

return pages \* OPTEE\_MSG\_NONCONTIG\_PAGE\_SIZE;

}

u64 \*optee\_allocate\_pages\_list(size\_t num\_entries)

{

return alloc\_pages\_exact(get\_pages\_list\_size(num\_entries), GFP\_KERNEL);

}

void optee\_free\_pages\_list(void \*list, size\_t num\_entries)

{

free\_pages\_exact(list, get\_pages\_list\_size(num\_entries));

}

static bool is\_normal\_memory(pgprot\_t p)

{

#if defined(CONFIG\_ARM)

return (pgprot\_val(p) & L\_PTE\_MT\_MASK) == L\_PTE\_MT\_WRITEALLOC;

#elif defined(CONFIG\_ARM64)

return (pgprot\_val(p) & PTE\_ATTRINDX\_MASK) == PTE\_ATTRINDX(MT\_NORMAL);

#else

#error "Unuspported architecture"

#endif

}

static int \_\_check\_mem\_type(struct vm\_area\_struct \*vma, unsigned long end)

{

while (vma && is\_normal\_memory(vma->vm\_page\_prot)) {

if (vma->vm\_end >= end)

return 0;

vma = vma->vm\_next;

}

return -EINVAL;

}

static int check\_mem\_type(unsigned long start, size\_t num\_pages)

{

struct mm\_struct \*mm = current->mm;

int rc;

down\_read(&mm->mmap\_sem);

rc = \_\_check\_mem\_type(find\_vma(mm, start),

start + num\_pages \* PAGE\_SIZE);

up\_read(&mm->mmap\_sem);

return rc;

}

int optee\_shm\_register(struct tee\_context \*ctx, struct tee\_shm \*shm,

struct page \*\*pages, size\_t num\_pages,

unsigned long start)

{

struct tee\_shm \*shm\_arg = NULL;

struct optee\_msg\_arg \*msg\_arg;

u64 \*pages\_list;

phys\_addr\_t msg\_parg;

int rc;

if (!num\_pages)

return -EINVAL;

rc = check\_mem\_type(start, num\_pages);

if (rc)

return rc;

pages\_list = optee\_allocate\_pages\_list(num\_pages);

if (!pages\_list)

return -ENOMEM;

shm\_arg = get\_msg\_arg(ctx, 1, &msg\_arg, &msg\_parg);

if (IS\_ERR(shm\_arg)) {

rc = PTR\_ERR(shm\_arg);

goto out;

}

optee\_fill\_pages\_list(pages\_list, pages, num\_pages,

tee\_shm\_get\_page\_offset(shm));

msg\_arg->cmd = OPTEE\_MSG\_CMD\_REGISTER\_SHM;

msg\_arg->params->attr = OPTEE\_MSG\_ATTR\_TYPE\_TMEM\_OUTPUT |

OPTEE\_MSG\_ATTR\_NONCONTIG;

msg\_arg->params->u.tmem.shm\_ref = (unsigned long)shm;

msg\_arg->params->u.tmem.size = tee\_shm\_get\_size(shm);

/\*

\* In the least bits of msg\_arg->params->u.tmem.buf\_ptr we

\* store buffer offset from 4k page, as described in OP-TEE ABI.

\*/

msg\_arg->params->u.tmem.buf\_ptr = virt\_to\_phys(pages\_list) |

(tee\_shm\_get\_page\_offset(shm) & (OPTEE\_MSG\_NONCONTIG\_PAGE\_SIZE - 1));

if (optee\_do\_call\_with\_arg(ctx, msg\_parg) ||

msg\_arg->ret != TEEC\_SUCCESS)

rc = -EINVAL;

tee\_shm\_free(shm\_arg);

out:

optee\_free\_pages\_list(pages\_list, num\_pages);

return rc;

}

int optee\_shm\_unregister(struct tee\_context \*ctx, struct tee\_shm \*shm)

{

struct tee\_shm \*shm\_arg;

struct optee\_msg\_arg \*msg\_arg;

phys\_addr\_t msg\_parg;

int rc = 0;

shm\_arg = get\_msg\_arg(ctx, 1, &msg\_arg, &msg\_parg);

if (IS\_ERR(shm\_arg))

return PTR\_ERR(shm\_arg);

msg\_arg->cmd = OPTEE\_MSG\_CMD\_UNREGISTER\_SHM;

msg\_arg->params[0].attr = OPTEE\_MSG\_ATTR\_TYPE\_RMEM\_INPUT;

msg\_arg->params[0].u.rmem.shm\_ref = (unsigned long)shm;

if (optee\_do\_call\_with\_arg(ctx, msg\_parg) ||

msg\_arg->ret != TEEC\_SUCCESS)

rc = -EINVAL;

tee\_shm\_free(shm\_arg);

return rc;

}

int optee\_shm\_register\_supp(struct tee\_context \*ctx, struct tee\_shm \*shm,

struct page \*\*pages, size\_t num\_pages,

unsigned long start)

{

/\*

\* We don't want to register supplicant memory in OP-TEE.

\* Instead information about it will be passed in RPC code.

\*/

return check\_mem\_type(start, num\_pages);

}

int optee\_shm\_unregister\_supp(struct tee\_context \*ctx, struct tee\_shm \*shm)

{

return 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

rpc.c

/\*

\* Copyright (c) 2015-2016, Linaro Limited

\*

\* This software is licensed under the terms of the GNU General Public

\* License version 2, as published by the Free Software Foundation, and

\* may be copied, distributed, and modified under those terms.

\*

\* This program is distributed in the hope that it will be useful,

\* but WITHOUT ANY WARRANTY; without even the implied warranty of

\* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

\* GNU General Public License for more details.

\*

\*/

#define pr\_fmt(fmt) KBUILD\_MODNAME ": " fmt

#include <linux/delay.h>

#include <linux/device.h>

#include <linux/slab.h>

#include <linux/tee\_drv.h>

#include "optee\_private.h"

#include "optee\_smc.h"

struct wq\_entry {

struct list\_head link;

struct completion c;

u32 key;

};

void optee\_wait\_queue\_init(struct optee\_wait\_queue \*priv)

{

mutex\_init(&priv->mu);

INIT\_LIST\_HEAD(&priv->db);

}

void optee\_wait\_queue\_exit(struct optee\_wait\_queue \*priv)

{

mutex\_destroy(&priv->mu);

}

static void handle\_rpc\_func\_cmd\_get\_time(struct optee\_msg\_arg \*arg)

{

struct timespec64 ts;

if (arg->num\_params != 1)

goto bad;

if ((arg->params[0].attr & OPTEE\_MSG\_ATTR\_TYPE\_MASK) !=

OPTEE\_MSG\_ATTR\_TYPE\_VALUE\_OUTPUT)

goto bad;

ktime\_get\_real\_ts64(&ts);

arg->params[0].u.value.a = ts.tv\_sec;

arg->params[0].u.value.b = ts.tv\_nsec;

arg->ret = TEEC\_SUCCESS;

return;

bad:

arg->ret = TEEC\_ERROR\_BAD\_PARAMETERS;

}

static struct wq\_entry \*wq\_entry\_get(struct optee\_wait\_queue \*wq, u32 key)

{

struct wq\_entry \*w;

mutex\_lock(&wq->mu);

list\_for\_each\_entry(w, &wq->db, link)

if (w->key == key)

goto out;

w = kmalloc(sizeof(\*w), GFP\_KERNEL);

if (w) {

init\_completion(&w->c);

w->key = key;

list\_add\_tail(&w->link, &wq->db);

}

out:

mutex\_unlock(&wq->mu);

return w;

}

static void wq\_sleep(struct optee\_wait\_queue \*wq, u32 key)

{

struct wq\_entry \*w = wq\_entry\_get(wq, key);

if (w) {

wait\_for\_completion(&w->c);

mutex\_lock(&wq->mu);

list\_del(&w->link);

mutex\_unlock(&wq->mu);

kfree(w);

}

}

static void wq\_wakeup(struct optee\_wait\_queue \*wq, u32 key)

{

struct wq\_entry \*w = wq\_entry\_get(wq, key);

if (w)

complete(&w->c);

}

static void handle\_rpc\_func\_cmd\_wq(struct optee \*optee,

struct optee\_msg\_arg \*arg)

{

if (arg->num\_params != 1)

goto bad;

if ((arg->params[0].attr & OPTEE\_MSG\_ATTR\_TYPE\_MASK) !=

OPTEE\_MSG\_ATTR\_TYPE\_VALUE\_INPUT)

goto bad;

switch (arg->params[0].u.value.a) {

case OPTEE\_MSG\_RPC\_WAIT\_QUEUE\_SLEEP:

wq\_sleep(&optee->wait\_queue, arg->params[0].u.value.b);

break;

case OPTEE\_MSG\_RPC\_WAIT\_QUEUE\_WAKEUP:

wq\_wakeup(&optee->wait\_queue, arg->params[0].u.value.b);

break;

default:

goto bad;

}

arg->ret = TEEC\_SUCCESS;

return;

bad:

arg->ret = TEEC\_ERROR\_BAD\_PARAMETERS;

}

static void handle\_rpc\_func\_cmd\_wait(struct optee\_msg\_arg \*arg)

{

u32 msec\_to\_wait;

if (arg->num\_params != 1)

goto bad;

if ((arg->params[0].attr & OPTEE\_MSG\_ATTR\_TYPE\_MASK) !=

OPTEE\_MSG\_ATTR\_TYPE\_VALUE\_INPUT)

goto bad;

msec\_to\_wait = arg->params[0].u.value.a;

/\* Go to interruptible sleep \*/

msleep\_interruptible(msec\_to\_wait);

arg->ret = TEEC\_SUCCESS;

return;

bad:

arg->ret = TEEC\_ERROR\_BAD\_PARAMETERS;

}

static void handle\_rpc\_supp\_cmd(struct tee\_context \*ctx,

struct optee\_msg\_arg \*arg)

{

struct tee\_param \*params;

arg->ret\_origin = TEEC\_ORIGIN\_COMMS;

params = kmalloc\_array(arg->num\_params, sizeof(struct tee\_param),

GFP\_KERNEL);

if (!params) {

arg->ret = TEEC\_ERROR\_OUT\_OF\_MEMORY;

return;

}

if (optee\_from\_msg\_param(params, arg->num\_params, arg->params)) {

arg->ret = TEEC\_ERROR\_BAD\_PARAMETERS;

goto out;

}

arg->ret = optee\_supp\_thrd\_req(ctx, arg->cmd, arg->num\_params, params);

if (optee\_to\_msg\_param(arg->params, arg->num\_params, params))

arg->ret = TEEC\_ERROR\_BAD\_PARAMETERS;

out:

kfree(params);

}

static struct tee\_shm \*cmd\_alloc\_suppl(struct tee\_context \*ctx, size\_t sz)

{

u32 ret;

struct tee\_param param;

struct optee \*optee = tee\_get\_drvdata(ctx->teedev);

struct tee\_shm \*shm;

param.attr = TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_VALUE\_INOUT;

param.u.value.a = OPTEE\_MSG\_RPC\_SHM\_TYPE\_APPL;

param.u.value.b = sz;

param.u.value.c = 0;

ret = optee\_supp\_thrd\_req(ctx, OPTEE\_MSG\_RPC\_CMD\_SHM\_ALLOC, 1, &param);

if (ret)

return ERR\_PTR(-ENOMEM);

mutex\_lock(&optee->supp.mutex);

/\* Increases count as secure world doesn't have a reference \*/

shm = tee\_shm\_get\_from\_id(optee->supp.ctx, param.u.value.c);

mutex\_unlock(&optee->supp.mutex);

return shm;

}

static void handle\_rpc\_func\_cmd\_shm\_alloc(struct tee\_context \*ctx,

struct optee\_msg\_arg \*arg,

struct optee\_call\_ctx \*call\_ctx)

{

phys\_addr\_t pa;

struct tee\_shm \*shm;

size\_t sz;

size\_t n;

arg->ret\_origin = TEEC\_ORIGIN\_COMMS;

if (!arg->num\_params ||

arg->params[0].attr != OPTEE\_MSG\_ATTR\_TYPE\_VALUE\_INPUT) {

arg->ret = TEEC\_ERROR\_BAD\_PARAMETERS;

return;

}

for (n = 1; n < arg->num\_params; n++) {

if (arg->params[n].attr != OPTEE\_MSG\_ATTR\_TYPE\_NONE) {

arg->ret = TEEC\_ERROR\_BAD\_PARAMETERS;

return;

}

}

sz = arg->params[0].u.value.b;

switch (arg->params[0].u.value.a) {

case OPTEE\_MSG\_RPC\_SHM\_TYPE\_APPL:

shm = cmd\_alloc\_suppl(ctx, sz);

break;

case OPTEE\_MSG\_RPC\_SHM\_TYPE\_KERNEL:

shm = tee\_shm\_alloc(ctx, sz, TEE\_SHM\_MAPPED);

break;

default:

arg->ret = TEEC\_ERROR\_BAD\_PARAMETERS;

return;

}

if (IS\_ERR(shm)) {

arg->ret = TEEC\_ERROR\_OUT\_OF\_MEMORY;

return;

}

if (tee\_shm\_get\_pa(shm, 0, &pa)) {

arg->ret = TEEC\_ERROR\_BAD\_PARAMETERS;

goto bad;

}

sz = tee\_shm\_get\_size(shm);

if (tee\_shm\_is\_registered(shm)) {

struct page \*\*pages;

u64 \*pages\_list;

size\_t page\_num;

pages = tee\_shm\_get\_pages(shm, &page\_num);

if (!pages || !page\_num) {

arg->ret = TEEC\_ERROR\_OUT\_OF\_MEMORY;

goto bad;

}

pages\_list = optee\_allocate\_pages\_list(page\_num);

if (!pages\_list) {

arg->ret = TEEC\_ERROR\_OUT\_OF\_MEMORY;

goto bad;

}

call\_ctx->pages\_list = pages\_list;

call\_ctx->num\_entries = page\_num;

arg->params[0].attr = OPTEE\_MSG\_ATTR\_TYPE\_TMEM\_OUTPUT |

OPTEE\_MSG\_ATTR\_NONCONTIG;

/\*

\* In the least bits of u.tmem.buf\_ptr we store buffer offset

\* from 4k page, as described in OP-TEE ABI.

\*/

arg->params[0].u.tmem.buf\_ptr = virt\_to\_phys(pages\_list) |

(tee\_shm\_get\_page\_offset(shm) &

(OPTEE\_MSG\_NONCONTIG\_PAGE\_SIZE - 1));

arg->params[0].u.tmem.size = tee\_shm\_get\_size(shm);

arg->params[0].u.tmem.shm\_ref = (unsigned long)shm;

optee\_fill\_pages\_list(pages\_list, pages, page\_num,

tee\_shm\_get\_page\_offset(shm));

} else {

arg->params[0].attr = OPTEE\_MSG\_ATTR\_TYPE\_TMEM\_OUTPUT;

arg->params[0].u.tmem.buf\_ptr = pa;

arg->params[0].u.tmem.size = sz;

arg->params[0].u.tmem.shm\_ref = (unsigned long)shm;

}

arg->ret = TEEC\_SUCCESS;

return;

bad:

tee\_shm\_free(shm);

}

static void cmd\_free\_suppl(struct tee\_context \*ctx, struct tee\_shm \*shm)

{

struct tee\_param param;

param.attr = TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_VALUE\_INOUT;

param.u.value.a = OPTEE\_MSG\_RPC\_SHM\_TYPE\_APPL;

param.u.value.b = tee\_shm\_get\_id(shm);

param.u.value.c = 0;

/\*

\* Match the tee\_shm\_get\_from\_id() in cmd\_alloc\_suppl() as secure

\* world has released its reference.

\*

\* It's better to do this before sending the request to supplicant

\* as we'd like to let the process doing the initial allocation to

\* do release the last reference too in order to avoid stacking

\* many pending fput() on the client process. This could otherwise

\* happen if secure world does many allocate and free in a single

\* invoke.

\*/

tee\_shm\_put(shm);

optee\_supp\_thrd\_req(ctx, OPTEE\_MSG\_RPC\_CMD\_SHM\_FREE, 1, &param);

}

static void handle\_rpc\_func\_cmd\_shm\_free(struct tee\_context \*ctx,

struct optee\_msg\_arg \*arg)

{

struct tee\_shm \*shm;

arg->ret\_origin = TEEC\_ORIGIN\_COMMS;

if (arg->num\_params != 1 ||

arg->params[0].attr != OPTEE\_MSG\_ATTR\_TYPE\_VALUE\_INPUT) {

arg->ret = TEEC\_ERROR\_BAD\_PARAMETERS;

return;

}

shm = (struct tee\_shm \*)(unsigned long)arg->params[0].u.value.b;

switch (arg->params[0].u.value.a) {

case OPTEE\_MSG\_RPC\_SHM\_TYPE\_APPL:

cmd\_free\_suppl(ctx, shm);

break;

case OPTEE\_MSG\_RPC\_SHM\_TYPE\_KERNEL:

tee\_shm\_free(shm);

break;

default:

arg->ret = TEEC\_ERROR\_BAD\_PARAMETERS;

}

arg->ret = TEEC\_SUCCESS;

}

static void free\_pages\_list(struct optee\_call\_ctx \*call\_ctx)

{

if (call\_ctx->pages\_list) {

optee\_free\_pages\_list(call\_ctx->pages\_list,

call\_ctx->num\_entries);

call\_ctx->pages\_list = NULL;

call\_ctx->num\_entries = 0;

}

}

void optee\_rpc\_finalize\_call(struct optee\_call\_ctx \*call\_ctx)

{

free\_pages\_list(call\_ctx);

}

static void handle\_rpc\_func\_cmd(struct tee\_context \*ctx, struct optee \*optee,

struct tee\_shm \*shm,

struct optee\_call\_ctx \*call\_ctx)

{

struct optee\_msg\_arg \*arg;

arg = tee\_shm\_get\_va(shm, 0);

if (IS\_ERR(arg)) {

pr\_err("%s: tee\_shm\_get\_va %p failed\n", \_\_func\_\_, shm);

return;

}

switch (arg->cmd) {

case OPTEE\_MSG\_RPC\_CMD\_GET\_TIME:

handle\_rpc\_func\_cmd\_get\_time(arg);

break;

case OPTEE\_MSG\_RPC\_CMD\_WAIT\_QUEUE:

handle\_rpc\_func\_cmd\_wq(optee, arg);

break;

case OPTEE\_MSG\_RPC\_CMD\_SUSPEND:

handle\_rpc\_func\_cmd\_wait(arg);

break;

case OPTEE\_MSG\_RPC\_CMD\_SHM\_ALLOC:

free\_pages\_list(call\_ctx);

handle\_rpc\_func\_cmd\_shm\_alloc(ctx, arg, call\_ctx);

break;

case OPTEE\_MSG\_RPC\_CMD\_SHM\_FREE:

handle\_rpc\_func\_cmd\_shm\_free(ctx, arg);

break;

default:

handle\_rpc\_supp\_cmd(ctx, arg);

}

}

/\*\*

\* optee\_handle\_rpc() - handle RPC from secure world

\* @ctx: context doing the RPC

\* @param: value of registers for the RPC

\* @call\_ctx: call context. Preserved during one OP-TEE invocation

\*

\* Result of RPC is written back into @param.

\*/

void optee\_handle\_rpc(struct tee\_context \*ctx, struct optee\_rpc\_param \*param,

struct optee\_call\_ctx \*call\_ctx)

{

struct tee\_device \*teedev = ctx->teedev;

struct optee \*optee = tee\_get\_drvdata(teedev);

struct tee\_shm \*shm;

phys\_addr\_t pa;

switch (OPTEE\_SMC\_RETURN\_GET\_RPC\_FUNC(param->a0)) {

case OPTEE\_SMC\_RPC\_FUNC\_ALLOC:

shm = tee\_shm\_alloc(ctx, param->a1, TEE\_SHM\_MAPPED);

if (!IS\_ERR(shm) && !tee\_shm\_get\_pa(shm, 0, &pa)) {

reg\_pair\_from\_64(&param->a1, &param->a2, pa);

reg\_pair\_from\_64(&param->a4, &param->a5,

(unsigned long)shm);

} else {

param->a1 = 0;

param->a2 = 0;

param->a4 = 0;

param->a5 = 0;

}

break;

case OPTEE\_SMC\_RPC\_FUNC\_FREE:

shm = reg\_pair\_to\_ptr(param->a1, param->a2);

tee\_shm\_free(shm);

break;

case OPTEE\_SMC\_RPC\_FUNC\_FOREIGN\_INTR:

/\*

\* A foreign interrupt was raised while secure world was

\* executing, since they are handled in Linux a dummy RPC is

\* performed to let Linux take the interrupt through the normal

\* vector.

\*/

break;

case OPTEE\_SMC\_RPC\_FUNC\_CMD:

shm = reg\_pair\_to\_ptr(param->a1, param->a2);

handle\_rpc\_func\_cmd(ctx, optee, shm, call\_ctx);

break;

default:

pr\_warn("Unknown RPC func 0x%x\n",

(u32)OPTEE\_SMC\_RETURN\_GET\_RPC\_FUNC(param->a0));

break;

}

param->a0 = OPTEE\_SMC\_CALL\_RETURN\_FROM\_RPC;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

shm\_pool.c

/\*

\* Copyright (c) 2015, Linaro Limited

\* Copyright (c) 2017, EPAM Systems

\*

\* This software is licensed under the terms of the GNU General Public

\* License version 2, as published by the Free Software Foundation, and

\* may be copied, distributed, and modified under those terms.

\*

\* This program is distributed in the hope that it will be useful,

\* but WITHOUT ANY WARRANTY; without even the implied warranty of

\* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

\* GNU General Public License for more details.

\*

\*/

#include <linux/device.h>

#include <linux/dma-buf.h>

#include <linux/genalloc.h>

#include <linux/slab.h>

#include <linux/tee\_drv.h>

#include "optee\_private.h"

#include "optee\_smc.h"

#include "shm\_pool.h"

static int pool\_op\_alloc(struct tee\_shm\_pool\_mgr \*poolm,

struct tee\_shm \*shm, size\_t size)

{

unsigned int order = get\_order(size);

struct page \*page;

page = alloc\_pages(GFP\_KERNEL | \_\_GFP\_ZERO, order);

if (!page)

return -ENOMEM;

shm->kaddr = page\_address(page);

shm->paddr = page\_to\_phys(page);

shm->size = PAGE\_SIZE << order;

return 0;

}

static void pool\_op\_free(struct tee\_shm\_pool\_mgr \*poolm,

struct tee\_shm \*shm)

{

free\_pages((unsigned long)shm->kaddr, get\_order(shm->size));

shm->kaddr = NULL;

}

static void pool\_op\_destroy\_poolmgr(struct tee\_shm\_pool\_mgr \*poolm)

{

kfree(poolm);

}

static const struct tee\_shm\_pool\_mgr\_ops pool\_ops = {

.alloc = pool\_op\_alloc,

.free = pool\_op\_free,

.destroy\_poolmgr = pool\_op\_destroy\_poolmgr,

};

/\*\*

\* optee\_shm\_pool\_alloc\_pages() - create page-based allocator pool

\*

\* This pool is used when OP-TEE supports dymanic SHM. In this case

\* command buffers and such are allocated from kernel's own memory.

\*/

struct tee\_shm\_pool\_mgr \*optee\_shm\_pool\_alloc\_pages(void)

{

struct tee\_shm\_pool\_mgr \*mgr = kzalloc(sizeof(\*mgr), GFP\_KERNEL);

if (!mgr)

return ERR\_PTR(-ENOMEM);

mgr->ops = &pool\_ops;

return mgr;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

supp.c

/\*

\* Copyright (c) 2015, Linaro Limited

\*

\* This software is licensed under the terms of the GNU General Public

\* License version 2, as published by the Free Software Foundation, and

\* may be copied, distributed, and modified under those terms.

\*

\* This program is distributed in the hope that it will be useful,

\* but WITHOUT ANY WARRANTY; without even the implied warranty of

\* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

\* GNU General Public License for more details.

\*

\*/

#include <linux/device.h>

#include <linux/slab.h>

#include <linux/uaccess.h>

#include "optee\_private.h"

struct optee\_supp\_req {

struct list\_head link;

bool busy;

u32 func;

u32 ret;

size\_t num\_params;

struct tee\_param \*param;

struct completion c;

};

void optee\_supp\_init(struct optee\_supp \*supp)

{

memset(supp, 0, sizeof(\*supp));

mutex\_init(&supp->mutex);

init\_completion(&supp->reqs\_c);

idr\_init(&supp->idr);

INIT\_LIST\_HEAD(&supp->reqs);

supp->req\_id = -1;

}

void optee\_supp\_uninit(struct optee\_supp \*supp)

{

mutex\_destroy(&supp->mutex);

idr\_destroy(&supp->idr);

}

void optee\_supp\_release(struct optee\_supp \*supp)

{

int id;

struct optee\_supp\_req \*req;

struct optee\_supp\_req \*req\_tmp;

mutex\_lock(&supp->mutex);

/\* Abort all request retrieved by supplicant \*/

idr\_for\_each\_entry(&supp->idr, req, id) {

req->busy = false;

idr\_remove(&supp->idr, id);

req->ret = TEEC\_ERROR\_COMMUNICATION;

complete(&req->c);

}

/\* Abort all queued requests \*/

list\_for\_each\_entry\_safe(req, req\_tmp, &supp->reqs, link) {

list\_del(&req->link);

req->ret = TEEC\_ERROR\_COMMUNICATION;

complete(&req->c);

}

supp->ctx = NULL;

supp->req\_id = -1;

mutex\_unlock(&supp->mutex);

}

/\*\*

\* optee\_supp\_thrd\_req() - request service from supplicant

\* @ctx: context doing the request

\* @func: function requested

\* @num\_params: number of elements in @param array

\* @param: parameters for function

\*

\* Returns result of operation to be passed to secure world

\*/

u32 optee\_supp\_thrd\_req(struct tee\_context \*ctx, u32 func, size\_t num\_params,

struct tee\_param \*param)

{

struct optee \*optee = tee\_get\_drvdata(ctx->teedev);

struct optee\_supp \*supp = &optee->supp;

struct optee\_supp\_req \*req = kzalloc(sizeof(\*req), GFP\_KERNEL);

bool interruptable;

u32 ret;

if (!req)

return TEEC\_ERROR\_OUT\_OF\_MEMORY;

init\_completion(&req->c);

req->func = func;

req->num\_params = num\_params;

req->param = param;

/\* Insert the request in the request list \*/

mutex\_lock(&supp->mutex);

list\_add\_tail(&req->link, &supp->reqs);

mutex\_unlock(&supp->mutex);

/\* Tell an eventual waiter there's a new request \*/

complete(&supp->reqs\_c);

/\*

\* Wait for supplicant to process and return result, once we've

\* returned from wait\_for\_completion(&req->c) successfully we have

\* exclusive access again.

\*/

while (wait\_for\_completion\_interruptible(&req->c)) {

mutex\_lock(&supp->mutex);

interruptable = !supp->ctx;

if (interruptable) {

/\*

\* There's no supplicant available and since the

\* supp->mutex currently is held none can

\* become available until the mutex released

\* again.

\*

\* Interrupting an RPC to supplicant is only

\* allowed as a way of slightly improving the user

\* experience in case the supplicant hasn't been

\* started yet. During normal operation the supplicant

\* will serve all requests in a timely manner and

\* interrupting then wouldn't make sense.

\*/

interruptable = !req->busy;

if (!req->busy)

list\_del(&req->link);

}

mutex\_unlock(&supp->mutex);

if (interruptable) {

req->ret = TEEC\_ERROR\_COMMUNICATION;

break;

}

}

ret = req->ret;

kfree(req);

return ret;

}

static struct optee\_supp\_req \*supp\_pop\_entry(struct optee\_supp \*supp,

int num\_params, int \*id)

{

struct optee\_supp\_req \*req;

if (supp->req\_id != -1) {

/\*

\* Supplicant should not mix synchronous and asnynchronous

\* requests.

\*/

return ERR\_PTR(-EINVAL);

}

if (list\_empty(&supp->reqs))

return NULL;

req = list\_first\_entry(&supp->reqs, struct optee\_supp\_req, link);

if (num\_params < req->num\_params) {

/\* Not enough room for parameters \*/

return ERR\_PTR(-EINVAL);

}

\*id = idr\_alloc(&supp->idr, req, 1, 0, GFP\_KERNEL);

if (\*id < 0)

return ERR\_PTR(-ENOMEM);

list\_del(&req->link);

req->busy = true;

return req;

}

static int supp\_check\_recv\_params(size\_t num\_params, struct tee\_param \*params,

size\_t \*num\_meta)

{

size\_t n;

if (!num\_params)

return -EINVAL;

/\*

\* If there's memrefs we need to decrease those as they where

\* increased earlier and we'll even refuse to accept any below.

\*/

for (n = 0; n < num\_params; n++)

if (tee\_param\_is\_memref(params + n) && params[n].u.memref.shm)

tee\_shm\_put(params[n].u.memref.shm);

/\*

\* We only expect parameters as TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_NONE with

\* or without the TEE\_IOCTL\_PARAM\_ATTR\_META bit set.

\*/

for (n = 0; n < num\_params; n++)

if (params[n].attr &&

params[n].attr != TEE\_IOCTL\_PARAM\_ATTR\_META)

return -EINVAL;

/\* At most we'll need one meta parameter so no need to check for more \*/

if (params->attr == TEE\_IOCTL\_PARAM\_ATTR\_META)

\*num\_meta = 1;

else

\*num\_meta = 0;

return 0;

}

/\*\*

\* optee\_supp\_recv() - receive request for supplicant

\* @ctx: context receiving the request

\* @func: requested function in supplicant

\* @num\_params: number of elements allocated in @param, updated with number

\* used elements

\* @param: space for parameters for @func

\*

\* Returns 0 on success or <0 on failure

\*/

int optee\_supp\_recv(struct tee\_context \*ctx, u32 \*func, u32 \*num\_params,

struct tee\_param \*param)

{

struct tee\_device \*teedev = ctx->teedev;

struct optee \*optee = tee\_get\_drvdata(teedev);

struct optee\_supp \*supp = &optee->supp;

struct optee\_supp\_req \*req = NULL;

int id;

size\_t num\_meta;

int rc;

rc = supp\_check\_recv\_params(\*num\_params, param, &num\_meta);

if (rc)

return rc;

while (true) {

mutex\_lock(&supp->mutex);

req = supp\_pop\_entry(supp, \*num\_params - num\_meta, &id);

mutex\_unlock(&supp->mutex);

if (req) {

if (IS\_ERR(req))

return PTR\_ERR(req);

break;

}

/\*

\* If we didn't get a request we'll block in

\* wait\_for\_completion() to avoid needless spinning.

\*

\* This is where supplicant will be hanging most of

\* the time, let's make this interruptable so we

\* can easily restart supplicant if needed.

\*/

if (wait\_for\_completion\_interruptible(&supp->reqs\_c))

return -ERESTARTSYS;

}

if (num\_meta) {

/\*

\* tee-supplicant support meta parameters -> requsts can be

\* processed asynchronously.

\*/

param->attr = TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_VALUE\_INOUT |

TEE\_IOCTL\_PARAM\_ATTR\_META;

param->u.value.a = id;

param->u.value.b = 0;

param->u.value.c = 0;

} else {

mutex\_lock(&supp->mutex);

supp->req\_id = id;

mutex\_unlock(&supp->mutex);

}

\*func = req->func;

\*num\_params = req->num\_params + num\_meta;

memcpy(param + num\_meta, req->param,

sizeof(struct tee\_param) \* req->num\_params);

return 0;

}

static struct optee\_supp\_req \*supp\_pop\_req(struct optee\_supp \*supp,

size\_t num\_params,

struct tee\_param \*param,

size\_t \*num\_meta)

{

struct optee\_supp\_req \*req;

int id;

size\_t nm;

const u32 attr = TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_VALUE\_INOUT |

TEE\_IOCTL\_PARAM\_ATTR\_META;

if (!num\_params)

return ERR\_PTR(-EINVAL);

if (supp->req\_id == -1) {

if (param->attr != attr)

return ERR\_PTR(-EINVAL);

id = param->u.value.a;

nm = 1;

} else {

id = supp->req\_id;

nm = 0;

}

req = idr\_find(&supp->idr, id);

if (!req)

return ERR\_PTR(-ENOENT);

if ((num\_params - nm) != req->num\_params)

return ERR\_PTR(-EINVAL);

req->busy = false;

idr\_remove(&supp->idr, id);

supp->req\_id = -1;

\*num\_meta = nm;

return req;

}

/\*\*

\* optee\_supp\_send() - send result of request from supplicant

\* @ctx: context sending result

\* @ret: return value of request

\* @num\_params: number of parameters returned

\* @param: returned parameters

\*

\* Returns 0 on success or <0 on failure.

\*/

int optee\_supp\_send(struct tee\_context \*ctx, u32 ret, u32 num\_params,

struct tee\_param \*param)

{

struct tee\_device \*teedev = ctx->teedev;

struct optee \*optee = tee\_get\_drvdata(teedev);

struct optee\_supp \*supp = &optee->supp;

struct optee\_supp\_req \*req;

size\_t n;

size\_t num\_meta;

mutex\_lock(&supp->mutex);

req = supp\_pop\_req(supp, num\_params, param, &num\_meta);

mutex\_unlock(&supp->mutex);

if (IS\_ERR(req)) {

/\* Something is wrong, let supplicant restart. \*/

return PTR\_ERR(req);

}

/\* Update out and in/out parameters \*/

for (n = 0; n < req->num\_params; n++) {

struct tee\_param \*p = req->param + n;

switch (p->attr & TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_MASK) {

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_VALUE\_OUTPUT:

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_VALUE\_INOUT:

p->u.value.a = param[n + num\_meta].u.value.a;

p->u.value.b = param[n + num\_meta].u.value.b;

p->u.value.c = param[n + num\_meta].u.value.c;

break;

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_MEMREF\_OUTPUT:

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_MEMREF\_INOUT:

p->u.memref.size = param[n + num\_meta].u.memref.size;

break;

default:

break;

}

}

req->ret = ret;

/\* Let the requesting thread continue \*/

complete(&req->c);

return 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

tee\_core.c

/\*

\* Copyright (c) 2015-2016, Linaro Limited

\*

\* This software is licensed under the terms of the GNU General Public

\* License version 2, as published by the Free Software Foundation, and

\* may be copied, distributed, and modified under those terms.

\*

\* This program is distributed in the hope that it will be useful,

\* but WITHOUT ANY WARRANTY; without even the implied warranty of

\* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

\* GNU General Public License for more details.

\*

\*/

#define pr\_fmt(fmt) "%s: " fmt, \_\_func\_\_

#include <linux/cdev.h>

#include <linux/device.h>

#include <linux/fs.h>

#include <linux/idr.h>

#include <linux/module.h>

#include <linux/slab.h>

#include <linux/tee\_drv.h>

#include <linux/uaccess.h>

#include "tee\_private.h"

#define TEE\_NUM\_DEVICES 32

#define TEE\_IOCTL\_PARAM\_SIZE(x) (sizeof(struct tee\_param) \* (x))

/\*

\* Unprivileged devices in the lower half range and privileged devices in

\* the upper half range.

\*/

static DECLARE\_BITMAP(dev\_mask, TEE\_NUM\_DEVICES);

static DEFINE\_SPINLOCK(driver\_lock);

static struct class \*tee\_class;

static dev\_t tee\_devt;

static int tee\_open(struct inode \*inode, struct file \*filp)

{

int rc;

struct tee\_device \*teedev;

struct tee\_context \*ctx;

teedev = container\_of(inode->i\_cdev, struct tee\_device, cdev);

if (!tee\_device\_get(teedev))

return -EINVAL;

ctx = kzalloc(sizeof(\*ctx), GFP\_KERNEL);

if (!ctx) {

rc = -ENOMEM;

goto err;

}

kref\_init(&ctx->refcount);

ctx->teedev = teedev;

INIT\_LIST\_HEAD(&ctx->list\_shm);

filp->private\_data = ctx;

rc = teedev->desc->ops->open(ctx);

if (rc)

goto err;

return 0;

err:

kfree(ctx);

tee\_device\_put(teedev);

return rc;

}

void teedev\_ctx\_get(struct tee\_context \*ctx)

{

if (ctx->releasing)

return;

kref\_get(&ctx->refcount);

}

static void teedev\_ctx\_release(struct kref \*ref)

{

struct tee\_context \*ctx = container\_of(ref, struct tee\_context,

refcount);

ctx->releasing = true;

ctx->teedev->desc->ops->release(ctx);

kfree(ctx);

}

void teedev\_ctx\_put(struct tee\_context \*ctx)

{

if (ctx->releasing)

return;

kref\_put(&ctx->refcount, teedev\_ctx\_release);

}

static void teedev\_close\_context(struct tee\_context \*ctx)

{/\*

\* Copyright (c) 2015-2016, Linaro Limited

\*

\* This software is licensed under the terms of the GNU General Public

\* License version 2, as published by the Free Software Foundation, and

\* may be copied, distributed, and modified under those terms.

\*

\* This program is distributed in the hope that it will be useful,

\* but WITHOUT ANY WARRANTY; without even the implied warranty of

\* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

\* GNU General Public License for more details.

\*

\*/

#define pr\_fmt(fmt) "%s: " fmt, \_\_func\_\_

#include <linux/cdev.h>

#include <linux/device.h>

#include <linux/fs.h>

#include <linux/idr.h>

#include <linux/module.h>

#include <linux/slab.h>

#include <linux/tee\_drv.h>

#include <linux/uaccess.h>

#include "tee\_private.h"

#define TEE\_NUM\_DEVICES 32

#define TEE\_IOCTL\_PARAM\_SIZE(x) (sizeof(struct tee\_param) \* (x))

/\*

\* Unprivileged devices in the lower half range and privileged devices in

\* the upper half range.

\*/

static DECLARE\_BITMAP(dev\_mask, TEE\_NUM\_DEVICES);

static DEFINE\_SPINLOCK(driver\_lock);

static struct class \*tee\_class;

static dev\_t tee\_devt;

static int tee\_open(struct inode \*inode, struct file \*filp)

{

int rc;

struct tee\_device \*teedev;

struct tee\_context \*ctx;

teedev = container\_of(inode->i\_cdev, struct tee\_device, cdev);

if (!tee\_device\_get(teedev))

return -EINVAL;

ctx = kzalloc(sizeof(\*ctx), GFP\_KERNEL);

if (!ctx) {

rc = -ENOMEM;

goto err;

}

kref\_init(&ctx->refcount);

ctx->teedev = teedev;

INIT\_LIST\_HEAD(&ctx->list\_shm);

filp->private\_data = ctx;

rc = teedev->desc->ops->open(ctx);

if (rc)

goto err;

return 0;

err:

kfree(ctx);

tee\_device\_put(teedev);

return rc;

}

void teedev\_ctx\_get(struct tee\_context \*ctx)

{

if (ctx->releasing)

return;

kref\_get(&ctx->refcount);

}

static void teedev\_ctx\_release(struct kref \*ref)

{

struct tee\_context \*ctx = container\_of(ref, struct tee\_context,

refcount);

ctx->releasing = true;

ctx->teedev->desc->ops->release(ctx);

kfree(ctx);

}

void teedev\_ctx\_put(struct tee\_context \*ctx)

{

if (ctx->releasing)

return;

kref\_put(&ctx->refcount, teedev\_ctx\_release);

}

static void teedev\_close\_context(struct tee\_context \*ctx)

{

tee\_device\_put(ctx->teedev);

teedev\_ctx\_put(ctx);

}

static int tee\_release(struct inode \*inode, struct file \*filp)

{

teedev\_close\_context(filp->private\_data);

return 0;

}

static int tee\_ioctl\_version(struct tee\_context \*ctx,

struct tee\_ioctl\_version\_data \_\_user \*uvers)

{

struct tee\_ioctl\_version\_data vers;

ctx->teedev->desc->ops->get\_version(ctx->teedev, &vers);

if (ctx->teedev->desc->flags & TEE\_DESC\_PRIVILEGED)

vers.gen\_caps |= TEE\_GEN\_CAP\_PRIVILEGED;

if (copy\_to\_user(uvers, &vers, sizeof(vers)))

return -EFAULT;

return 0;

}

static int tee\_ioctl\_shm\_alloc(struct tee\_context \*ctx,

struct tee\_ioctl\_shm\_alloc\_data \_\_user \*udata)

{

long ret;

struct tee\_ioctl\_shm\_alloc\_data data;

struct tee\_shm \*shm;

if (copy\_from\_user(&data, udata, sizeof(data)))

return -EFAULT;

/\* Currently no input flags are supported \*/

if (data.flags)

return -EINVAL;

shm = tee\_shm\_alloc(ctx, data.size, TEE\_SHM\_MAPPED | TEE\_SHM\_DMA\_BUF);

if (IS\_ERR(shm))

return PTR\_ERR(shm);

data.id = shm->id;

data.flags = shm->flags;

data.size = shm->size;

if (copy\_to\_user(udata, &data, sizeof(data)))

ret = -EFAULT;

else

ret = tee\_shm\_get\_fd(shm);

/\*

\* When user space closes the file descriptor the shared memory

\* should be freed or if tee\_shm\_get\_fd() failed then it will

\* be freed immediately.

\*/

tee\_shm\_put(shm);

return ret;

}

static int

tee\_ioctl\_shm\_register(struct tee\_context \*ctx,

struct tee\_ioctl\_shm\_register\_data \_\_user \*udata)

{

long ret;

struct tee\_ioctl\_shm\_register\_data data;

struct tee\_shm \*shm;

if (copy\_from\_user(&data, udata, sizeof(data)))

return -EFAULT;

/\* Currently no input flags are supported \*/

if (data.flags)

return -EINVAL;

shm = tee\_shm\_register(ctx, data.addr, data.length,

TEE\_SHM\_DMA\_BUF | TEE\_SHM\_USER\_MAPPED);

if (IS\_ERR(shm))

return PTR\_ERR(shm);

data.id = shm->id;

data.flags = shm->flags;

data.length = shm->size;

if (copy\_to\_user(udata, &data, sizeof(data)))

ret = -EFAULT;

else

ret = tee\_shm\_get\_fd(shm);

/\*

\* When user space closes the file descriptor the shared memory

\* should be freed or if tee\_shm\_get\_fd() failed then it will

\* be freed immediately.

\*/

tee\_shm\_put(shm);

return ret;

}

static int params\_from\_user(struct tee\_context \*ctx, struct tee\_param \*params,

size\_t num\_params,

struct tee\_ioctl\_param \_\_user \*uparams)

{

size\_t n;

for (n = 0; n < num\_params; n++) {

struct tee\_shm \*shm;

struct tee\_ioctl\_param ip;

if (copy\_from\_user(&ip, uparams + n, sizeof(ip)))

return -EFAULT;

/\* All unused attribute bits has to be zero \*/

if (ip.attr & ~TEE\_IOCTL\_PARAM\_ATTR\_MASK)

return -EINVAL;

params[n].attr = ip.attr;

switch (ip.attr & TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_MASK) {

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_NONE:

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_VALUE\_OUTPUT:

break;

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_VALUE\_INPUT:

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_VALUE\_INOUT:

params[n].u.value.a = ip.a;

params[n].u.value.b = ip.b;

params[n].u.value.c = ip.c;

break;

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_MEMREF\_INPUT:

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_MEMREF\_OUTPUT:

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_MEMREF\_INOUT:

/\*

\* If we fail to get a pointer to a shared memory

\* object (and increase the ref count) from an

\* identifier we return an error. All pointers that

\* has been added in params have an increased ref

\* count. It's the callers responibility to do

\* tee\_shm\_put() on all resolved pointers.

\*/

shm = tee\_shm\_get\_from\_id(ctx, ip.c);

if (IS\_ERR(shm))

return PTR\_ERR(shm);

/\*

\* Ensure offset + size does not overflow offset

\* and does not overflow the size of the referred

\* shared memory object.

\*/

if ((ip.a + ip.b) < ip.a ||

(ip.a + ip.b) > shm->size) {

tee\_shm\_put(shm);

return -EINVAL;

}

params[n].u.memref.shm\_offs = ip.a;

params[n].u.memref.size = ip.b;

params[n].u.memref.shm = shm;

break;

default:

/\* Unknown attribute \*/

return -EINVAL;

}

}

return 0;

}

static int params\_to\_user(struct tee\_ioctl\_param \_\_user \*uparams,

size\_t num\_params, struct tee\_param \*params)

{

size\_t n;

for (n = 0; n < num\_params; n++) {

struct tee\_ioctl\_param \_\_user \*up = uparams + n;

struct tee\_param \*p = params + n;

switch (p->attr) {

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_VALUE\_OUTPUT:

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_VALUE\_INOUT:

if (put\_user(p->u.value.a, &up->a) ||

put\_user(p->u.value.b, &up->b) ||

put\_user(p->u.value.c, &up->c))

return -EFAULT;

break;

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_MEMREF\_OUTPUT:

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_MEMREF\_INOUT:

if (put\_user((u64)p->u.memref.size, &up->b))

return -EFAULT;

default:

break;

}

}

return 0;

}

static int tee\_ioctl\_open\_session(struct tee\_context \*ctx,

struct tee\_ioctl\_buf\_data \_\_user \*ubuf)

{

int rc;

size\_t n;

struct tee\_ioctl\_buf\_data buf;

struct tee\_ioctl\_open\_session\_arg \_\_user \*uarg;

struct tee\_ioctl\_open\_session\_arg arg;

struct tee\_ioctl\_param \_\_user \*uparams = NULL;

struct tee\_param \*params = NULL;

bool have\_session = false;

if (!ctx->teedev->desc->ops->open\_session)

return -EINVAL;

if (copy\_from\_user(&buf, ubuf, sizeof(buf)))

return -EFAULT;

if (buf.buf\_len > TEE\_MAX\_ARG\_SIZE ||

buf.buf\_len < sizeof(struct tee\_ioctl\_open\_session\_arg))

return -EINVAL;

uarg = u64\_to\_user\_ptr(buf.buf\_ptr);

if (copy\_from\_user(&arg, uarg, sizeof(arg)))

return -EFAULT;

if (sizeof(arg) + TEE\_IOCTL\_PARAM\_SIZE(arg.num\_params) != buf.buf\_len)

return -EINVAL;

if (arg.num\_params) {

params = kcalloc(arg.num\_params, sizeof(struct tee\_param),

GFP\_KERNEL);

if (!params)

return -ENOMEM;

uparams = uarg->params;

rc = params\_from\_user(ctx, params, arg.num\_params, uparams);

if (rc)

goto out;

}

rc = ctx->teedev->desc->ops->open\_session(ctx, &arg, params);

if (rc)

goto out;

have\_session = true;

if (put\_user(arg.session, &uarg->session) ||

put\_user(arg.ret, &uarg->ret) ||

put\_user(arg.ret\_origin, &uarg->ret\_origin)) {

rc = -EFAULT;

goto out;

}

rc = params\_to\_user(uparams, arg.num\_params, params);

out:

/\*

\* If we've succeeded to open the session but failed to communicate

\* it back to user space, close the session again to avoid leakage.

\*/

if (rc && have\_session && ctx->teedev->desc->ops->close\_session)

ctx->teedev->desc->ops->close\_session(ctx, arg.session);

if (params) {

/\* Decrease ref count for all valid shared memory pointers \*/

for (n = 0; n < arg.num\_params; n++)

if (tee\_param\_is\_memref(params + n) &&

params[n].u.memref.shm)

tee\_shm\_put(params[n].u.memref.shm);

kfree(params);

}

return rc;

}

static int tee\_ioctl\_invoke(struct tee\_context \*ctx,

struct tee\_ioctl\_buf\_data \_\_user \*ubuf)

{

int rc;

size\_t n;

struct tee\_ioctl\_buf\_data buf;

struct tee\_ioctl\_invoke\_arg \_\_user \*uarg;

struct tee\_ioctl\_invoke\_arg arg;

struct tee\_ioctl\_param \_\_user \*uparams = NULL;

struct tee\_param \*params = NULL;

if (!ctx->teedev->desc->ops->invoke\_func)

return -EINVAL;

if (copy\_from\_user(&buf, ubuf, sizeof(buf)))

return -EFAULT;

if (buf.buf\_len > TEE\_MAX\_ARG\_SIZE ||

buf.buf\_len < sizeof(struct tee\_ioctl\_invoke\_arg))

return -EINVAL;

uarg = u64\_to\_user\_ptr(buf.buf\_ptr);

if (copy\_from\_user(&arg, uarg, sizeof(arg)))

return -EFAULT;

if (sizeof(arg) + TEE\_IOCTL\_PARAM\_SIZE(arg.num\_params) != buf.buf\_len)

return -EINVAL;

if (arg.num\_params) {

params = kcalloc(arg.num\_params, sizeof(struct tee\_param),

GFP\_KERNEL);

if (!params)

return -ENOMEM;

uparams = uarg->params;

rc = params\_from\_user(ctx, params, arg.num\_params, uparams);

if (rc)

goto out;

}

rc = ctx->teedev->desc->ops->invoke\_func(ctx, &arg, params);

if (rc)

goto out;

if (put\_user(arg.ret, &uarg->ret) ||

put\_user(arg.ret\_origin, &uarg->ret\_origin)) {

rc = -EFAULT;

goto out;

}

rc = params\_to\_user(uparams, arg.num\_params, params);

out:

if (params) {

/\* Decrease ref count for all valid shared memory pointers \*/

for (n = 0; n < arg.num\_params; n++)

if (tee\_param\_is\_memref(params + n) &&

params[n].u.memref.shm)

tee\_shm\_put(params[n].u.memref.shm);

kfree(params);

}

return rc;

}

static int tee\_ioctl\_cancel(struct tee\_context \*ctx,

struct tee\_ioctl\_cancel\_arg \_\_user \*uarg)

{

struct tee\_ioctl\_cancel\_arg arg;

if (!ctx->teedev->desc->ops->cancel\_req)

return -EINVAL;

if (copy\_from\_user(&arg, uarg, sizeof(arg)))

return -EFAULT;

return ctx->teedev->desc->ops->cancel\_req(ctx, arg.cancel\_id,

arg.session);

}

static int

tee\_ioctl\_close\_session(struct tee\_context \*ctx,

struct tee\_ioctl\_close\_session\_arg \_\_user \*uarg)

{

struct tee\_ioctl\_close\_session\_arg arg;

if (!ctx->teedev->desc->ops->close\_session)

return -EINVAL;

if (copy\_from\_user(&arg, uarg, sizeof(arg)))

return -EFAULT;

return ctx->teedev->desc->ops->close\_session(ctx, arg.session);

}

static int params\_to\_supp(struct tee\_context \*ctx,

struct tee\_ioctl\_param \_\_user \*uparams,

size\_t num\_params, struct tee\_param \*params)

{

size\_t n;

for (n = 0; n < num\_params; n++) {

struct tee\_ioctl\_param ip;

struct tee\_param \*p = params + n;

ip.attr = p->attr;

switch (p->attr & TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_MASK) {

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_VALUE\_INPUT:

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_VALUE\_INOUT:

ip.a = p->u.value.a;

ip.b = p->u.value.b;

ip.c = p->u.value.c;

break;

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_MEMREF\_INPUT:

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_MEMREF\_OUTPUT:

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_MEMREF\_INOUT:

ip.b = p->u.memref.size;

if (!p->u.memref.shm) {

ip.a = 0;

ip.c = (u64)-1; /\* invalid shm id \*/

break;

}

ip.a = p->u.memref.shm\_offs;

ip.c = p->u.memref.shm->id;

break;

default:

ip.a = 0;

ip.b = 0;

ip.c = 0;

break;

}

if (copy\_to\_user(uparams + n, &ip, sizeof(ip)))

return -EFAULT;

}

return 0;

}

static int tee\_ioctl\_supp\_recv(struct tee\_context \*ctx,

struct tee\_ioctl\_buf\_data \_\_user \*ubuf)

{

int rc;

struct tee\_ioctl\_buf\_data buf;

struct tee\_iocl\_supp\_recv\_arg \_\_user \*uarg;

struct tee\_param \*params;

u32 num\_params;

u32 func;

if (!ctx->teedev->desc->ops->supp\_recv)

return -EINVAL;

if (copy\_from\_user(&buf, ubuf, sizeof(buf)))

return -EFAULT;

if (buf.buf\_len > TEE\_MAX\_ARG\_SIZE ||

buf.buf\_len < sizeof(struct tee\_iocl\_supp\_recv\_arg))

return -EINVAL;

uarg = u64\_to\_user\_ptr(buf.buf\_ptr);

if (get\_user(num\_params, &uarg->num\_params))

return -EFAULT;

if (sizeof(\*uarg) + TEE\_IOCTL\_PARAM\_SIZE(num\_params) != buf.buf\_len)

return -EINVAL;

params = kcalloc(num\_params, sizeof(struct tee\_param), GFP\_KERNEL);

if (!params)

return -ENOMEM;

rc = params\_from\_user(ctx, params, num\_params, uarg->params);

if (rc)

goto out;

rc = ctx->teedev->desc->ops->supp\_recv(ctx, &func, &num\_params, params);

if (rc)

goto out;

if (put\_user(func, &uarg->func) ||

put\_user(num\_params, &uarg->num\_params)) {

rc = -EFAULT;

goto out;

}

rc = params\_to\_supp(ctx, uarg->params, num\_params, params);

out:

kfree(params);

return rc;

}

static int params\_from\_supp(struct tee\_param \*params, size\_t num\_params,

struct tee\_ioctl\_param \_\_user \*uparams)

{

size\_t n;

for (n = 0; n < num\_params; n++) {

struct tee\_param \*p = params + n;

struct tee\_ioctl\_param ip;

if (copy\_from\_user(&ip, uparams + n, sizeof(ip)))

return -EFAULT;

/\* All unused attribute bits has to be zero \*/

if (ip.attr & ~TEE\_IOCTL\_PARAM\_ATTR\_MASK)

return -EINVAL;

p->attr = ip.attr;

switch (ip.attr & TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_MASK) {

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_VALUE\_OUTPUT:

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_VALUE\_INOUT:

/\* Only out and in/out values can be updated \*/

p->u.value.a = ip.a;

p->u.value.b = ip.b;

p->u.value.c = ip.c;

break;

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_MEMREF\_OUTPUT:

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_MEMREF\_INOUT:

/\*

\* Only the size of the memref can be updated.

\* Since we don't have access to the original

\* parameters here, only store the supplied size.

\* The driver will copy the updated size into the

\* original parameters.

\*/

p->u.memref.shm = NULL;

p->u.memref.shm\_offs = 0;

p->u.memref.size = ip.b;

break;

default:

memset(&p->u, 0, sizeof(p->u));

break;

}

}

return 0;

}

static int tee\_ioctl\_supp\_send(struct tee\_context \*ctx,

struct tee\_ioctl\_buf\_data \_\_user \*ubuf)

{

long rc;

struct tee\_ioctl\_buf\_data buf;

struct tee\_iocl\_supp\_send\_arg \_\_user \*uarg;

struct tee\_param \*params;

u32 num\_params;

u32 ret;

/\* Not valid for this driver \*/

if (!ctx->teedev->desc->ops->supp\_send)

return -EINVAL;

if (copy\_from\_user(&buf, ubuf, sizeof(buf)))

return -EFAULT;

if (buf.buf\_len > TEE\_MAX\_ARG\_SIZE ||

buf.buf\_len < sizeof(struct tee\_iocl\_supp\_send\_arg))

return -EINVAL;

uarg = u64\_to\_user\_ptr(buf.buf\_ptr);

if (get\_user(ret, &uarg->ret) ||

get\_user(num\_params, &uarg->num\_params))

return -EFAULT;

if (sizeof(\*uarg) + TEE\_IOCTL\_PARAM\_SIZE(num\_params) > buf.buf\_len)

return -EINVAL;

params = kcalloc(num\_params, sizeof(struct tee\_param), GFP\_KERNEL);

if (!params)

return -ENOMEM;

rc = params\_from\_supp(params, num\_params, uarg->params);

if (rc)

goto out;

rc = ctx->teedev->desc->ops->supp\_send(ctx, ret, num\_params, params);

out:

kfree(params);

return rc;

}

static long tee\_ioctl(struct file \*filp, unsigned int cmd, unsigned long arg)

{

struct tee\_context \*ctx = filp->private\_data;

void \_\_user \*uarg = (void \_\_user \*)arg;

switch (cmd) {

case TEE\_IOC\_VERSION:

return tee\_ioctl\_version(ctx, uarg);

case TEE\_IOC\_SHM\_ALLOC:

return tee\_ioctl\_shm\_alloc(ctx, uarg);

case TEE\_IOC\_SHM\_REGISTER:

return tee\_ioctl\_shm\_register(ctx, uarg);

case TEE\_IOC\_OPEN\_SESSION:

return tee\_ioctl\_open\_session(ctx, uarg);

case TEE\_IOC\_INVOKE:

return tee\_ioctl\_invoke(ctx, uarg);

case TEE\_IOC\_CANCEL:

return tee\_ioctl\_cancel(ctx, uarg);

case TEE\_IOC\_CLOSE\_SESSION:

return tee\_ioctl\_close\_session(ctx, uarg);

case TEE\_IOC\_SUPPL\_RECV:

return tee\_ioctl\_supp\_recv(ctx, uarg);

case TEE\_IOC\_SUPPL\_SEND:

return tee\_ioctl\_supp\_send(ctx, uarg);

default:

return -EINVAL;

}

}

static const struct file\_operations tee\_fops = {

.owner = THIS\_MODULE,

.open = tee\_open,

.release = tee\_release,

.unlocked\_ioctl = tee\_ioctl,

.compat\_ioctl = tee\_ioctl,

};

static void tee\_release\_device(struct device \*dev)

{

struct tee\_device \*teedev = container\_of(dev, struct tee\_device, dev);

spin\_lock(&driver\_lock);

clear\_bit(teedev->id, dev\_mask);

spin\_unlock(&driver\_lock);

mutex\_destroy(&teedev->mutex);

idr\_destroy(&teedev->idr);

kfree(teedev);

}

/\*\*

\* tee\_device\_alloc() - Allocate a new struct tee\_device instance

\* @teedesc: Descriptor for this driver

\* @dev: Parent device for this device

\* @pool: Shared memory pool, NULL if not used

\* @driver\_data: Private driver data for this device

\*

\* Allocates a new struct tee\_device instance. The device is

\* removed by tee\_device\_unregister().

\*

\* @returns a pointer to a 'struct tee\_device' or an ERR\_PTR on failure

\*/

struct tee\_device \*tee\_device\_alloc(const struct tee\_desc \*teedesc,

struct device \*dev,

struct tee\_shm\_pool \*pool,

void \*driver\_data)

{

struct tee\_device \*teedev;

void \*ret;

int rc, max\_id;

int offs = 0;

if (!teedesc || !teedesc->name || !teedesc->ops ||

!teedesc->ops->get\_version || !teedesc->ops->open ||

!teedesc->ops->release || !pool)

return ERR\_PTR(-EINVAL);

teedev = kzalloc(sizeof(\*teedev), GFP\_KERNEL);

if (!teedev) {

ret = ERR\_PTR(-ENOMEM);

goto err;

}

max\_id = TEE\_NUM\_DEVICES / 2;

if (teedesc->flags & TEE\_DESC\_PRIVILEGED) {

offs = TEE\_NUM\_DEVICES / 2;

max\_id = TEE\_NUM\_DEVICES;

}

spin\_lock(&driver\_lock);

teedev->id = find\_next\_zero\_bit(dev\_mask, max\_id, offs);

if (teedev->id < max\_id)

set\_bit(teedev->id, dev\_mask);

spin\_unlock(&driver\_lock);

if (teedev->id >= max\_id) {

ret = ERR\_PTR(-ENOMEM);

goto err;

}

snprintf(teedev->name, sizeof(teedev->name), "tee%s%d",

teedesc->flags & TEE\_DESC\_PRIVILEGED ? "priv" : "",

teedev->id - offs);

teedev->dev.class = tee\_class;

teedev->dev.release = tee\_release\_device;

teedev->dev.parent = dev;

teedev->dev.devt = MKDEV(MAJOR(tee\_devt), teedev->id);

rc = dev\_set\_name(&teedev->dev, "%s", teedev->name);

if (rc) {

ret = ERR\_PTR(rc);

goto err\_devt;

}

cdev\_init(&teedev->cdev, &tee\_fops);

teedev->cdev.owner = teedesc->owner;

teedev->cdev.kobj.parent = &teedev->dev.kobj;

dev\_set\_drvdata(&teedev->dev, driver\_data);

device\_initialize(&teedev->dev);

/\* 1 as tee\_device\_unregister() does one final tee\_device\_put() \*/

teedev->num\_users = 1;

init\_completion(&teedev->c\_no\_users);

mutex\_init(&teedev->mutex);

idr\_init(&teedev->idr);

teedev->desc = teedesc;

teedev->pool = pool;

return teedev;

err\_devt:

unregister\_chrdev\_region(teedev->dev.devt, 1);

err:

pr\_err("could not register %s driver\n",

teedesc->flags & TEE\_DESC\_PRIVILEGED ? "privileged" : "client");

if (teedev && teedev->id < TEE\_NUM\_DEVICES) {

spin\_lock(&driver\_lock);

clear\_bit(teedev->id, dev\_mask);

spin\_unlock(&driver\_lock);

}

kfree(teedev);

return ret;

}

EXPORT\_SYMBOL\_GPL(tee\_device\_alloc);

static ssize\_t implementation\_id\_show(struct device \*dev,

struct device\_attribute \*attr, char \*buf)

{

struct tee\_device \*teedev = container\_of(dev, struct tee\_device, dev);

struct tee\_ioctl\_version\_data vers;

teedev->desc->ops->get\_version(teedev, &vers);

return scnprintf(buf, PAGE\_SIZE, "%d\n", vers.impl\_id);

}

static DEVICE\_ATTR\_RO(implementation\_id);

static struct attribute \*tee\_dev\_attrs[] = {

&dev\_attr\_implementation\_id.attr,

NULL

};

static const struct attribute\_group tee\_dev\_group = {

.attrs = tee\_dev\_attrs,

};

/\*\*

\* tee\_device\_register() - Registers a TEE device

\* @teedev: Device to register

\*

\* tee\_device\_unregister() need to be called to remove the @teedev if

\* this function fails.

\*

\* @returns < 0 on failure

\*/

int tee\_device\_register(struct tee\_device \*teedev)

{

int rc;

if (teedev->flags & TEE\_DEVICE\_FLAG\_REGISTERED) {

dev\_err(&teedev->dev, "attempt to register twice\n");

return -EINVAL;

}

rc = cdev\_add(&teedev->cdev, teedev->dev.devt, 1);

if (rc) {

dev\_err(&teedev->dev,

"unable to cdev\_add() %s, major %d, minor %d, err=%d\n",

teedev->name, MAJOR(teedev->dev.devt),

MINOR(teedev->dev.devt), rc);

return rc;

}

rc = device\_add(&teedev->dev);

if (rc) {

dev\_err(&teedev->dev,

"unable to device\_add() %s, major %d, minor %d, err=%d\n",

teedev->name, MAJOR(teedev->dev.devt),

MINOR(teedev->dev.devt), rc);

goto err\_device\_add;

}

rc = sysfs\_create\_group(&teedev->dev.kobj, &tee\_dev\_group);

if (rc) {

dev\_err(&teedev->dev,

"failed to create sysfs attributes, err=%d\n", rc);

goto err\_sysfs\_create\_group;

}

teedev->flags |= TEE\_DEVICE\_FLAG\_REGISTERED;

return 0;

err\_sysfs\_create\_group:

device\_del(&teedev->dev);

err\_device\_add:

cdev\_del(&teedev->cdev);

return rc;

}

EXPORT\_SYMBOL\_GPL(tee\_device\_register);

void tee\_device\_put(struct tee\_device \*teedev)

{

mutex\_lock(&teedev->mutex);

/\* Shouldn't put in this state \*/

if (!WARN\_ON(!teedev->desc)) {

teedev->num\_users--;

if (!teedev->num\_users) {

teedev->desc = NULL;

complete(&teedev->c\_no\_users);

}

}

mutex\_unlock(&teedev->mutex);

}

bool tee\_device\_get(struct tee\_device \*teedev)

{

mutex\_lock(&teedev->mutex);

if (!teedev->desc) {

mutex\_unlock(&teedev->mutex);

return false;

}

teedev->num\_users++;

mutex\_unlock(&teedev->mutex);

return true;

}

/\*\*

\* tee\_device\_unregister() - Removes a TEE device

\* @teedev: Device to unregister

\*

\* This function should be called to remove the @teedev even if

\* tee\_device\_register() hasn't been called yet. Does nothing if

\* @teedev is NULL.

\*/

void tee\_device\_unregister(struct tee\_device \*teedev)

{

if (!teedev)

return;

if (teedev->flags & TEE\_DEVICE\_FLAG\_REGISTERED) {

sysfs\_remove\_group(&teedev->dev.kobj, &tee\_dev\_group);

cdev\_del(&teedev->cdev);

device\_del(&teedev->dev);

}

tee\_device\_put(teedev);

wait\_for\_completion(&teedev->c\_no\_users);

/\*

\* No need to take a mutex any longer now since teedev->desc was

\* set to NULL before teedev->c\_no\_users was completed.

\*/

teedev->pool = NULL;

put\_device(&teedev->dev);

}

EXPORT\_SYMBOL\_GPL(tee\_device\_unregister);

/\*\*

\* tee\_get\_drvdata() - Return driver\_data pointer

\* @teedev: Device containing the driver\_data pointer

\* @returns the driver\_data pointer supplied to tee\_register().

\*/

void \*tee\_get\_drvdata(struct tee\_device \*teedev)

{

return dev\_get\_drvdata(&teedev->dev);

}

EXPORT\_SYMBOL\_GPL(tee\_get\_drvdata);

static int \_\_init tee\_init(void)

{

int rc;

tee\_class = class\_create(THIS\_MODULE, "tee");

if (IS\_ERR(tee\_class)) {

pr\_err("couldn't create class\n");

return PTR\_ERR(tee\_class);

}

rc = alloc\_chrdev\_region(&tee\_devt, 0, TEE\_NUM\_DEVICES, "tee");

if (rc) {

pr\_err("failed to allocate char dev region\n");

class\_destroy(tee\_class);

tee\_class = NULL;

}

return rc;

}

static void \_\_exit tee\_exit(void)

{

class\_destroy(tee\_class);

tee\_class = NULL;

unregister\_chrdev\_region(tee\_devt, TEE\_NUM\_DEVICES);

}

subsys\_initcall(tee\_init);

module\_exit(tee\_exit);

MODULE\_AUTHOR("Linaro");

MODULE\_DESCRIPTION("TEE Driver");

MODULE\_VERSION("1.0");

MODULE\_LICENSE("GPL v2");

tee\_device\_put(ctx->teedev);

teedev\_ctx\_put(ctx);

}

static int tee\_release(struct inode \*inode, struct file \*filp)

{

teedev\_close\_context(filp->private\_data);

return 0;

}

static int tee\_ioctl\_version(struct tee\_context \*ctx,

struct tee\_ioctl\_version\_data \_\_user \*uvers)

{

struct tee\_ioctl\_version\_data vers;

ctx->teedev->desc->ops->get\_version(ctx->teedev, &vers);

if (ctx->teedev->desc->flags & TEE\_DESC\_PRIVILEGED)

vers.gen\_caps |= TEE\_GEN\_CAP\_PRIVILEGED;

if (copy\_to\_user(uvers, &vers, sizeof(vers)))

return -EFAULT;

return 0;

}

static int tee\_ioctl\_shm\_alloc(struct tee\_context \*ctx,

struct tee\_ioctl\_shm\_alloc\_data \_\_user \*udata)

{

long ret;

struct tee\_ioctl\_shm\_alloc\_data data;

struct tee\_shm \*shm;

if (copy\_from\_user(&data, udata, sizeof(data)))

return -EFAULT;

/\* Currently no input flags are supported \*/

if (data.flags)

return -EINVAL;

shm = tee\_shm\_alloc(ctx, data.size, TEE\_SHM\_MAPPED | TEE\_SHM\_DMA\_BUF);

if (IS\_ERR(shm))

return PTR\_ERR(shm);

data.id = shm->id;

data.flags = shm->flags;

data.size = shm->size;

if (copy\_to\_user(udata, &data, sizeof(data)))

ret = -EFAULT;

else

ret = tee\_shm\_get\_fd(shm);

/\*

\* When user space closes the file descriptor the shared memory

\* should be freed or if tee\_shm\_get\_fd() failed then it will

\* be freed immediately.

\*/

tee\_shm\_put(shm);

return ret;

}

static int

tee\_ioctl\_shm\_register(struct tee\_context \*ctx,

struct tee\_ioctl\_shm\_register\_data \_\_user \*udata)

{

long ret;

struct tee\_ioctl\_shm\_register\_data data;

struct tee\_shm \*shm;

if (copy\_from\_user(&data, udata, sizeof(data)))

return -EFAULT;

/\* Currently no input flags are supported \*/

if (data.flags)

return -EINVAL;

shm = tee\_shm\_register(ctx, data.addr, data.length,

TEE\_SHM\_DMA\_BUF | TEE\_SHM\_USER\_MAPPED);

if (IS\_ERR(shm))

return PTR\_ERR(shm);

data.id = shm->id;

data.flags = shm->flags;

data.length = shm->size;

if (copy\_to\_user(udata, &data, sizeof(data)))

ret = -EFAULT;

else

ret = tee\_shm\_get\_fd(shm);

/\*

\* When user space closes the file descriptor the shared memory

\* should be freed or if tee\_shm\_get\_fd() failed then it will

\* be freed immediately.

\*/

tee\_shm\_put(shm);

return ret;

}

static int params\_from\_user(struct tee\_context \*ctx, struct tee\_param \*params,

size\_t num\_params,

struct tee\_ioctl\_param \_\_user \*uparams)

{

size\_t n;

for (n = 0; n < num\_params; n++) {

struct tee\_shm \*shm;

struct tee\_ioctl\_param ip;

if (copy\_from\_user(&ip, uparams + n, sizeof(ip)))

return -EFAULT;

/\* All unused attribute bits has to be zero \*/

if (ip.attr & ~TEE\_IOCTL\_PARAM\_ATTR\_MASK)

return -EINVAL;

params[n].attr = ip.attr;

switch (ip.attr & TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_MASK) {

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_NONE:

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_VALUE\_OUTPUT:

break;

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_VALUE\_INPUT:

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_VALUE\_INOUT:

params[n].u.value.a = ip.a;

params[n].u.value.b = ip.b;

params[n].u.value.c = ip.c;

break;

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_MEMREF\_INPUT:

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_MEMREF\_OUTPUT:

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_MEMREF\_INOUT:

/\*

\* If we fail to get a pointer to a shared memory

\* object (and increase the ref count) from an

\* identifier we return an error. All pointers that

\* has been added in params have an increased ref

\* count. It's the callers responibility to do

\* tee\_shm\_put() on all resolved pointers.

\*/

shm = tee\_shm\_get\_from\_id(ctx, ip.c);

if (IS\_ERR(shm))

return PTR\_ERR(shm);

/\*

\* Ensure offset + size does not overflow offset

\* and does not overflow the size of the referred

\* shared memory object.

\*/

if ((ip.a + ip.b) < ip.a ||

(ip.a + ip.b) > shm->size) {

tee\_shm\_put(shm);

return -EINVAL;

}

params[n].u.memref.shm\_offs = ip.a;

params[n].u.memref.size = ip.b;

params[n].u.memref.shm = shm;

break;

default:

/\* Unknown attribute \*/

return -EINVAL;

}

}

return 0;

}

static int params\_to\_user(struct tee\_ioctl\_param \_\_user \*uparams,

size\_t num\_params, struct tee\_param \*params)

{

size\_t n;

for (n = 0; n < num\_params; n++) {

struct tee\_ioctl\_param \_\_user \*up = uparams + n;

struct tee\_param \*p = params + n;

switch (p->attr) {

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_VALUE\_OUTPUT:

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_VALUE\_INOUT:

if (put\_user(p->u.value.a, &up->a) ||

put\_user(p->u.value.b, &up->b) ||

put\_user(p->u.value.c, &up->c))

return -EFAULT;

break;

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_MEMREF\_OUTPUT:

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_MEMREF\_INOUT:

if (put\_user((u64)p->u.memref.size, &up->b))

return -EFAULT;

default:

break;

}

}

return 0;

}

static int tee\_ioctl\_open\_session(struct tee\_context \*ctx,

struct tee\_ioctl\_buf\_data \_\_user \*ubuf)

{

int rc;

size\_t n;

struct tee\_ioctl\_buf\_data buf;

struct tee\_ioctl\_open\_session\_arg \_\_user \*uarg;

struct tee\_ioctl\_open\_session\_arg arg;

struct tee\_ioctl\_param \_\_user \*uparams = NULL;

struct tee\_param \*params = NULL;

bool have\_session = false;

if (!ctx->teedev->desc->ops->open\_session)

return -EINVAL;

if (copy\_from\_user(&buf, ubuf, sizeof(buf)))

return -EFAULT;

if (buf.buf\_len > TEE\_MAX\_ARG\_SIZE ||

buf.buf\_len < sizeof(struct tee\_ioctl\_open\_session\_arg))

return -EINVAL;

uarg = u64\_to\_user\_ptr(buf.buf\_ptr);

if (copy\_from\_user(&arg, uarg, sizeof(arg)))

return -EFAULT;

if (sizeof(arg) + TEE\_IOCTL\_PARAM\_SIZE(arg.num\_params) != buf.buf\_len)

return -EINVAL;

if (arg.num\_params) {

params = kcalloc(arg.num\_params, sizeof(struct tee\_param),

GFP\_KERNEL);

if (!params)

return -ENOMEM;

uparams = uarg->params;

rc = params\_from\_user(ctx, params, arg.num\_params, uparams);

if (rc)

goto out;

}

rc = ctx->teedev->desc->ops->open\_session(ctx, &arg, params);

if (rc)

goto out;

have\_session = true;

if (put\_user(arg.session, &uarg->session) ||

put\_user(arg.ret, &uarg->ret) ||

put\_user(arg.ret\_origin, &uarg->ret\_origin)) {

rc = -EFAULT;

goto out;

}

rc = params\_to\_user(uparams, arg.num\_params, params);

out:

/\*

\* If we've succeeded to open the session but failed to communicate

\* it back to user space, close the session again to avoid leakage.

\*/

if (rc && have\_session && ctx->teedev->desc->ops->close\_session)

ctx->teedev->desc->ops->close\_session(ctx, arg.session);

if (params) {

/\* Decrease ref count for all valid shared memory pointers \*/

for (n = 0; n < arg.num\_params; n++)

if (tee\_param\_is\_memref(params + n) &&

params[n].u.memref.shm)

tee\_shm\_put(params[n].u.memref.shm);

kfree(params);

}

return rc;

}

static int tee\_ioctl\_invoke(struct tee\_context \*ctx,

struct tee\_ioctl\_buf\_data \_\_user \*ubuf)

{

int rc;

size\_t n;

struct tee\_ioctl\_buf\_data buf;

struct tee\_ioctl\_invoke\_arg \_\_user \*uarg;

struct tee\_ioctl\_invoke\_arg arg;

struct tee\_ioctl\_param \_\_user \*uparams = NULL;

struct tee\_param \*params = NULL;

if (!ctx->teedev->desc->ops->invoke\_func)

return -EINVAL;

if (copy\_from\_user(&buf, ubuf, sizeof(buf)))

return -EFAULT;

if (buf.buf\_len > TEE\_MAX\_ARG\_SIZE ||

buf.buf\_len < sizeof(struct tee\_ioctl\_invoke\_arg))

return -EINVAL;

uarg = u64\_to\_user\_ptr(buf.buf\_ptr);

if (copy\_from\_user(&arg, uarg, sizeof(arg)))

return -EFAULT;

if (sizeof(arg) + TEE\_IOCTL\_PARAM\_SIZE(arg.num\_params) != buf.buf\_len)

return -EINVAL;

if (arg.num\_params) {

params = kcalloc(arg.num\_params, sizeof(struct tee\_param),

GFP\_KERNEL);

if (!params)

return -ENOMEM;

uparams = uarg->params;

rc = params\_from\_user(ctx, params, arg.num\_params, uparams);

if (rc)

goto out;

}

rc = ctx->teedev->desc->ops->invoke\_func(ctx, &arg, params);

if (rc)

goto out;

if (put\_user(arg.ret, &uarg->ret) ||

put\_user(arg.ret\_origin, &uarg->ret\_origin)) {

rc = -EFAULT;

goto out;

}

rc = params\_to\_user(uparams, arg.num\_params, params);

out:

if (params) {

/\* Decrease ref count for all valid shared memory pointers \*/

for (n = 0; n < arg.num\_params; n++)

if (tee\_param\_is\_memref(params + n) &&

params[n].u.memref.shm)

tee\_shm\_put(params[n].u.memref.shm);

kfree(params);

}

return rc;

}

static int tee\_ioctl\_cancel(struct tee\_context \*ctx,

struct tee\_ioctl\_cancel\_arg \_\_user \*uarg)

{

struct tee\_ioctl\_cancel\_arg arg;

if (!ctx->teedev->desc->ops->cancel\_req)

return -EINVAL;

if (copy\_from\_user(&arg, uarg, sizeof(arg)))

return -EFAULT;

return ctx->teedev->desc->ops->cancel\_req(ctx, arg.cancel\_id,

arg.session);

}

static int

tee\_ioctl\_close\_session(struct tee\_context \*ctx,

struct tee\_ioctl\_close\_session\_arg \_\_user \*uarg)

{

struct tee\_ioctl\_close\_session\_arg arg;

if (!ctx->teedev->desc->ops->close\_session)

return -EINVAL;

if (copy\_from\_user(&arg, uarg, sizeof(arg)))

return -EFAULT;

return ctx->teedev->desc->ops->close\_session(ctx, arg.session);

}

static int params\_to\_supp(struct tee\_context \*ctx,

struct tee\_ioctl\_param \_\_user \*uparams,

size\_t num\_params, struct tee\_param \*params)

{

size\_t n;

for (n = 0; n < num\_params; n++) {

struct tee\_ioctl\_param ip;

struct tee\_param \*p = params + n;

ip.attr = p->attr;

switch (p->attr & TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_MASK) {

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_VALUE\_INPUT:

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_VALUE\_INOUT:

ip.a = p->u.value.a;

ip.b = p->u.value.b;

ip.c = p->u.value.c;

break;

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_MEMREF\_INPUT:

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_MEMREF\_OUTPUT:

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_MEMREF\_INOUT:

ip.b = p->u.memref.size;

if (!p->u.memref.shm) {

ip.a = 0;

ip.c = (u64)-1; /\* invalid shm id \*/

break;

}

ip.a = p->u.memref.shm\_offs;

ip.c = p->u.memref.shm->id;

break;

default:

ip.a = 0;

ip.b = 0;

ip.c = 0;

break;

}

if (copy\_to\_user(uparams + n, &ip, sizeof(ip)))

return -EFAULT;

}

return 0;

}

static int tee\_ioctl\_supp\_recv(struct tee\_context \*ctx,

struct tee\_ioctl\_buf\_data \_\_user \*ubuf)

{

int rc;

struct tee\_ioctl\_buf\_data buf;

struct tee\_iocl\_supp\_recv\_arg \_\_user \*uarg;

struct tee\_param \*params;

u32 num\_params;

u32 func;

if (!ctx->teedev->desc->ops->supp\_recv)

return -EINVAL;

if (copy\_from\_user(&buf, ubuf, sizeof(buf)))

return -EFAULT;

if (buf.buf\_len > TEE\_MAX\_ARG\_SIZE ||

buf.buf\_len < sizeof(struct tee\_iocl\_supp\_recv\_arg))

return -EINVAL;

uarg = u64\_to\_user\_ptr(buf.buf\_ptr);

if (get\_user(num\_params, &uarg->num\_params))

return -EFAULT;

if (sizeof(\*uarg) + TEE\_IOCTL\_PARAM\_SIZE(num\_params) != buf.buf\_len)

return -EINVAL;

params = kcalloc(num\_params, sizeof(struct tee\_param), GFP\_KERNEL);

if (!params)

return -ENOMEM;

rc = params\_from\_user(ctx, params, num\_params, uarg->params);

if (rc)

goto out;

rc = ctx->teedev->desc->ops->supp\_recv(ctx, &func, &num\_params, params);

if (rc)

goto out;

if (put\_user(func, &uarg->func) ||

put\_user(num\_params, &uarg->num\_params)) {

rc = -EFAULT;

goto out;

}

rc = params\_to\_supp(ctx, uarg->params, num\_params, params);

out:

kfree(params);

return rc;

}

static int params\_from\_supp(struct tee\_param \*params, size\_t num\_params,

struct tee\_ioctl\_param \_\_user \*uparams)

{

size\_t n;

for (n = 0; n < num\_params; n++) {

struct tee\_param \*p = params + n;

struct tee\_ioctl\_param ip;

if (copy\_from\_user(&ip, uparams + n, sizeof(ip)))

return -EFAULT;

/\* All unused attribute bits has to be zero \*/

if (ip.attr & ~TEE\_IOCTL\_PARAM\_ATTR\_MASK)

return -EINVAL;

p->attr = ip.attr;

switch (ip.attr & TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_MASK) {

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_VALUE\_OUTPUT:

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_VALUE\_INOUT:

/\* Only out and in/out values can be updated \*/

p->u.value.a = ip.a;

p->u.value.b = ip.b;

p->u.value.c = ip.c;

break;

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_MEMREF\_OUTPUT:

case TEE\_IOCTL\_PARAM\_ATTR\_TYPE\_MEMREF\_INOUT:

/\*

\* Only the size of the memref can be updated.

\* Since we don't have access to the original

\* parameters here, only store the supplied size.

\* The driver will copy the updated size into the

\* original parameters.

\*/

p->u.memref.shm = NULL;

p->u.memref.shm\_offs = 0;

p->u.memref.size = ip.b;

break;

default:

memset(&p->u, 0, sizeof(p->u));

break;

}

}

return 0;

}

static int tee\_ioctl\_supp\_send(struct tee\_context \*ctx,

struct tee\_ioctl\_buf\_data \_\_user \*ubuf)

{

long rc;

struct tee\_ioctl\_buf\_data buf;

struct tee\_iocl\_supp\_send\_arg \_\_user \*uarg;

struct tee\_param \*params;

u32 num\_params;

u32 ret;

/\* Not valid for this driver \*/

if (!ctx->teedev->desc->ops->supp\_send)

return -EINVAL;

if (copy\_from\_user(&buf, ubuf, sizeof(buf)))

return -EFAULT;

if (buf.buf\_len > TEE\_MAX\_ARG\_SIZE ||

buf.buf\_len < sizeof(struct tee\_iocl\_supp\_send\_arg))

return -EINVAL;

uarg = u64\_to\_user\_ptr(buf.buf\_ptr);

if (get\_user(ret, &uarg->ret) ||

get\_user(num\_params, &uarg->num\_params))

return -EFAULT;

if (sizeof(\*uarg) + TEE\_IOCTL\_PARAM\_SIZE(num\_params) > buf.buf\_len)

return -EINVAL;

params = kcalloc(num\_params, sizeof(struct tee\_param), GFP\_KERNEL);

if (!params)

return -ENOMEM;

rc = params\_from\_supp(params, num\_params, uarg->params);

if (rc)

goto out;

rc = ctx->teedev->desc->ops->supp\_send(ctx, ret, num\_params, params);

out:

kfree(params);

return rc;

}

static long tee\_ioctl(struct file \*filp, unsigned int cmd, unsigned long arg)

{

struct tee\_context \*ctx = filp->private\_data;

void \_\_user \*uarg = (void \_\_user \*)arg;

switch (cmd) {

case TEE\_IOC\_VERSION:

return tee\_ioctl\_version(ctx, uarg);

case TEE\_IOC\_SHM\_ALLOC:

return tee\_ioctl\_shm\_alloc(ctx, uarg);

case TEE\_IOC\_SHM\_REGISTER:

return tee\_ioctl\_shm\_register(ctx, uarg);

case TEE\_IOC\_OPEN\_SESSION:

return tee\_ioctl\_open\_session(ctx, uarg);

case TEE\_IOC\_INVOKE:

return tee\_ioctl\_invoke(ctx, uarg);

case TEE\_IOC\_CANCEL:

return tee\_ioctl\_cancel(ctx, uarg);

case TEE\_IOC\_CLOSE\_SESSION:

return tee\_ioctl\_close\_session(ctx, uarg);

case TEE\_IOC\_SUPPL\_RECV:

return tee\_ioctl\_supp\_recv(ctx, uarg);

case TEE\_IOC\_SUPPL\_SEND:

return tee\_ioctl\_supp\_send(ctx, uarg);

default:

return -EINVAL;

}

}

static const struct file\_operations tee\_fops = {

.owner = THIS\_MODULE,

.open = tee\_open,

.release = tee\_release,

.unlocked\_ioctl = tee\_ioctl,

.compat\_ioctl = tee\_ioctl,

};

static void tee\_release\_device(struct device \*dev)

{

struct tee\_device \*teedev = container\_of(dev, struct tee\_device, dev);

spin\_lock(&driver\_lock);

clear\_bit(teedev->id, dev\_mask);

spin\_unlock(&driver\_lock);

mutex\_destroy(&teedev->mutex);

idr\_destroy(&teedev->idr);

kfree(teedev);

}

/\*\*

\* tee\_device\_alloc() - Allocate a new struct tee\_device instance

\* @teedesc: Descriptor for this driver

\* @dev: Parent device for this device

\* @pool: Shared memory pool, NULL if not used

\* @driver\_data: Private driver data for this device

\*

\* Allocates a new struct tee\_device instance. The device is

\* removed by tee\_device\_unregister().

\*

\* @returns a pointer to a 'struct tee\_device' or an ERR\_PTR on failure

\*/

struct tee\_device \*tee\_device\_alloc(const struct tee\_desc \*teedesc,

struct device \*dev,

struct tee\_shm\_pool \*pool,

void \*driver\_data)

{

struct tee\_device \*teedev;

void \*ret;

int rc, max\_id;

int offs = 0;

if (!teedesc || !teedesc->name || !teedesc->ops ||

!teedesc->ops->get\_version || !teedesc->ops->open ||

!teedesc->ops->release || !pool)

return ERR\_PTR(-EINVAL);

teedev = kzalloc(sizeof(\*teedev), GFP\_KERNEL);

if (!teedev) {

ret = ERR\_PTR(-ENOMEM);

goto err;

}

max\_id = TEE\_NUM\_DEVICES / 2;

if (teedesc->flags & TEE\_DESC\_PRIVILEGED) {

offs = TEE\_NUM\_DEVICES / 2;

max\_id = TEE\_NUM\_DEVICES;

}

spin\_lock(&driver\_lock);

teedev->id = find\_next\_zero\_bit(dev\_mask, max\_id, offs);

if (teedev->id < max\_id)

set\_bit(teedev->id, dev\_mask);

spin\_unlock(&driver\_lock);

if (teedev->id >= max\_id) {

ret = ERR\_PTR(-ENOMEM);

goto err;

}

snprintf(teedev->name, sizeof(teedev->name), "tee%s%d",

teedesc->flags & TEE\_DESC\_PRIVILEGED ? "priv" : "",

teedev->id - offs);

teedev->dev.class = tee\_class;

teedev->dev.release = tee\_release\_device;

teedev->dev.parent = dev;

teedev->dev.devt = MKDEV(MAJOR(tee\_devt), teedev->id);

rc = dev\_set\_name(&teedev->dev, "%s", teedev->name);

if (rc) {

ret = ERR\_PTR(rc);

goto err\_devt;

}

cdev\_init(&teedev->cdev, &tee\_fops);

teedev->cdev.owner = teedesc->owner;

teedev->cdev.kobj.parent = &teedev->dev.kobj;

dev\_set\_drvdata(&teedev->dev, driver\_data);

device\_initialize(&teedev->dev);

/\* 1 as tee\_device\_unregister() does one final tee\_device\_put() \*/

teedev->num\_users = 1;

init\_completion(&teedev->c\_no\_users);

mutex\_init(&teedev->mutex);

idr\_init(&teedev->idr);

teedev->desc = teedesc;

teedev->pool = pool;

return teedev;

err\_devt:

unregister\_chrdev\_region(teedev->dev.devt, 1);

err:

pr\_err("could not register %s driver\n",

teedesc->flags & TEE\_DESC\_PRIVILEGED ? "privileged" : "client");

if (teedev && teedev->id < TEE\_NUM\_DEVICES) {

spin\_lock(&driver\_lock);

clear\_bit(teedev->id, dev\_mask);

spin\_unlock(&driver\_lock);

}

kfree(teedev);

return ret;

}

EXPORT\_SYMBOL\_GPL(tee\_device\_alloc);

static ssize\_t implementation\_id\_show(struct device \*dev,

struct device\_attribute \*attr, char \*buf)

{

struct tee\_device \*teedev = container\_of(dev, struct tee\_device, dev);

struct tee\_ioctl\_version\_data vers;

teedev->desc->ops->get\_version(teedev, &vers);

return scnprintf(buf, PAGE\_SIZE, "%d\n", vers.impl\_id);

}

static DEVICE\_ATTR\_RO(implementation\_id);

static struct attribute \*tee\_dev\_attrs[] = {

&dev\_attr\_implementation\_id.attr,

NULL

};

static const struct attribute\_group tee\_dev\_group = {

.attrs = tee\_dev\_attrs,

};

/\*\*

\* tee\_device\_register() - Registers a TEE device

\* @teedev: Device to register

\*

\* tee\_device\_unregister() need to be called to remove the @teedev if

\* this function fails.

\*

\* @returns < 0 on failure

\*/

int tee\_device\_register(struct tee\_device \*teedev)

{

int rc;

if (teedev->flags & TEE\_DEVICE\_FLAG\_REGISTERED) {

dev\_err(&teedev->dev, "attempt to register twice\n");

return -EINVAL;

}

rc = cdev\_add(&teedev->cdev, teedev->dev.devt, 1);

if (rc) {

dev\_err(&teedev->dev,

"unable to cdev\_add() %s, major %d, minor %d, err=%d\n",

teedev->name, MAJOR(teedev->dev.devt),

MINOR(teedev->dev.devt), rc);

return rc;

}

rc = device\_add(&teedev->dev);

if (rc) {

dev\_err(&teedev->dev,

"unable to device\_add() %s, major %d, minor %d, err=%d\n",

teedev->name, MAJOR(teedev->dev.devt),

MINOR(teedev->dev.devt), rc);

goto err\_device\_add;

}

rc = sysfs\_create\_group(&teedev->dev.kobj, &tee\_dev\_group);

if (rc) {

dev\_err(&teedev->dev,

"failed to create sysfs attributes, err=%d\n", rc);

goto err\_sysfs\_create\_group;

}

teedev->flags |= TEE\_DEVICE\_FLAG\_REGISTERED;

return 0;

err\_sysfs\_create\_group:

device\_del(&teedev->dev);

err\_device\_add:

cdev\_del(&teedev->cdev);

return rc;

}

EXPORT\_SYMBOL\_GPL(tee\_device\_register);

void tee\_device\_put(struct tee\_device \*teedev)

{

mutex\_lock(&teedev->mutex);

/\* Shouldn't put in this state \*/

if (!WARN\_ON(!teedev->desc)) {

teedev->num\_users--;

if (!teedev->num\_users) {

teedev->desc = NULL;

complete(&teedev->c\_no\_users);

}

}

mutex\_unlock(&teedev->mutex);

}

bool tee\_device\_get(struct tee\_device \*teedev)

{

mutex\_lock(&teedev->mutex);

if (!teedev->desc) {

mutex\_unlock(&teedev->mutex);

return false;

}

teedev->num\_users++;

mutex\_unlock(&teedev->mutex);

return true;

}

/\*\*

\* tee\_device\_unregister() - Removes a TEE device

\* @teedev: Device to unregister

\*

\* This function should be called to remove the @teedev even if

\* tee\_device\_register() hasn't been called yet. Does nothing if

\* @teedev is NULL.

\*/

void tee\_device\_unregister(struct tee\_device \*teedev)

{

if (!teedev)

return;

if (teedev->flags & TEE\_DEVICE\_FLAG\_REGISTERED) {

sysfs\_remove\_group(&teedev->dev.kobj, &tee\_dev\_group);

cdev\_del(&teedev->cdev);

device\_del(&teedev->dev);

}

tee\_device\_put(teedev);

wait\_for\_completion(&teedev->c\_no\_users);

/\*

\* No need to take a mutex any longer now since teedev->desc was

\* set to NULL before teedev->c\_no\_users was completed.

\*/

teedev->pool = NULL;

put\_device(&teedev->dev);

}

EXPORT\_SYMBOL\_GPL(tee\_device\_unregister);

/\*\*

\* tee\_get\_drvdata() - Return driver\_data pointer

\* @teedev: Device containing the driver\_data pointer

\* @returns the driver\_data pointer supplied to tee\_register().

\*/

void \*tee\_get\_drvdata(struct tee\_device \*teedev)

{

return dev\_get\_drvdata(&teedev->dev);

}

EXPORT\_SYMBOL\_GPL(tee\_get\_drvdata);

static int \_\_init tee\_init(void)

{

int rc;

tee\_class = class\_create(THIS\_MODULE, "tee");

if (IS\_ERR(tee\_class)) {

pr\_err("couldn't create class\n");

return PTR\_ERR(tee\_class);

}

rc = alloc\_chrdev\_region(&tee\_devt, 0, TEE\_NUM\_DEVICES, "tee");

if (rc) {

pr\_err("failed to allocate char dev region\n");

class\_destroy(tee\_class);

tee\_class = NULL;

}

return rc;

}

static void \_\_exit tee\_exit(void)

{

class\_destroy(tee\_class);

tee\_class = NULL;

unregister\_chrdev\_region(tee\_devt, TEE\_NUM\_DEVICES);

}

subsys\_initcall(tee\_init);

module\_exit(tee\_exit);

MODULE\_AUTHOR("Linaro");

MODULE\_DESCRIPTION("TEE Driver");

MODULE\_VERSION("1.0");

MODULE\_LICENSE("GPL v2");

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

tee\_shm.c

/\*

\* Copyright (c) 2015-2016, Linaro Limited

\*

\* This software is licensed under the terms of the GNU General Public

\* License version 2, as published by the Free Software Foundation, and

\* may be copied, distributed, and modified under those terms.

\*

\* This program is distributed in the hope that it will be useful,

\* but WITHOUT ANY WARRANTY; without even the implied warranty of

\* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

\* GNU General Public License for more details.

\*

\*/

#include <linux/device.h>

#include <linux/dma-buf.h>

#include <linux/fdtable.h>

#include <linux/idr.h>

#include <linux/sched.h>

#include <linux/slab.h>

#include <linux/tee\_drv.h>

#include "tee\_private.h"

static void tee\_shm\_release(struct tee\_shm \*shm)

{

struct tee\_device \*teedev = shm->teedev;

mutex\_lock(&teedev->mutex);

idr\_remove(&teedev->idr, shm->id);

if (shm->ctx)

list\_del(&shm->link);

mutex\_unlock(&teedev->mutex);

if (shm->flags & TEE\_SHM\_POOL) {

struct tee\_shm\_pool\_mgr \*poolm;

if (shm->flags & TEE\_SHM\_DMA\_BUF)

poolm = teedev->pool->dma\_buf\_mgr;

else

poolm = teedev->pool->private\_mgr;

poolm->ops->free(poolm, shm);

} else if (shm->flags & TEE\_SHM\_REGISTER) {

size\_t n;

int rc = teedev->desc->ops->shm\_unregister(shm->ctx, shm);

if (rc)

dev\_err(teedev->dev.parent,

"unregister shm %p failed: %d", shm, rc);

for (n = 0; n < shm->num\_pages; n++)

put\_page(shm->pages[n]);

kfree(shm->pages);

}

if (shm->ctx)

teedev\_ctx\_put(shm->ctx);

kfree(shm);

tee\_device\_put(teedev);

}

static struct sg\_table \*tee\_shm\_op\_map\_dma\_buf(struct dma\_buf\_attachment

\*attach, enum dma\_data\_direction dir)

{

return NULL;

}

static void tee\_shm\_op\_unmap\_dma\_buf(struct dma\_buf\_attachment \*attach,

struct sg\_table \*table,

enum dma\_data\_direction dir)

{

}

static void tee\_shm\_op\_release(struct dma\_buf \*dmabuf)

{

struct tee\_shm \*shm = dmabuf->priv;

tee\_shm\_release(shm);

}

static void \*tee\_shm\_op\_map(struct dma\_buf \*dmabuf, unsigned long pgnum)

{

return NULL;

}

static int tee\_shm\_op\_mmap(struct dma\_buf \*dmabuf, struct vm\_area\_struct \*vma)

{

struct tee\_shm \*shm = dmabuf->priv;

size\_t size = vma->vm\_end - vma->vm\_start;

/\* Refuse sharing shared memory provided by application \*/

if (shm->flags & TEE\_SHM\_REGISTER)

return -EINVAL;

return remap\_pfn\_range(vma, vma->vm\_start, shm->paddr >> PAGE\_SHIFT,

size, vma->vm\_page\_prot);

}

static const struct dma\_buf\_ops tee\_shm\_dma\_buf\_ops = {

.map\_dma\_buf = tee\_shm\_op\_map\_dma\_buf,

.unmap\_dma\_buf = tee\_shm\_op\_unmap\_dma\_buf,

.release = tee\_shm\_op\_release,

.map = tee\_shm\_op\_map,

.mmap = tee\_shm\_op\_mmap,

};

static struct tee\_shm \*\_\_tee\_shm\_alloc(struct tee\_context \*ctx,

struct tee\_device \*teedev,

size\_t size, u32 flags)

{

struct tee\_shm\_pool\_mgr \*poolm = NULL;

struct tee\_shm \*shm;

void \*ret;

int rc;

if (ctx && ctx->teedev != teedev) {

dev\_err(teedev->dev.parent, "ctx and teedev mismatch\n");

return ERR\_PTR(-EINVAL);

}

if (!(flags & TEE\_SHM\_MAPPED)) {

dev\_err(teedev->dev.parent,

"only mapped allocations supported\n");

return ERR\_PTR(-EINVAL);

}

if ((flags & ~(TEE\_SHM\_MAPPED | TEE\_SHM\_DMA\_BUF))) {

dev\_err(teedev->dev.parent, "invalid shm flags 0x%x", flags);

return ERR\_PTR(-EINVAL);

}

if (!tee\_device\_get(teedev))

return ERR\_PTR(-EINVAL);

if (!teedev->pool) {

/\* teedev has been detached from driver \*/

ret = ERR\_PTR(-EINVAL);

goto err\_dev\_put;

}

shm = kzalloc(sizeof(\*shm), GFP\_KERNEL);

if (!shm) {

ret = ERR\_PTR(-ENOMEM);

goto err\_dev\_put;

}

shm->flags = flags | TEE\_SHM\_POOL;

shm->teedev = teedev;

shm->ctx = ctx;

if (flags & TEE\_SHM\_DMA\_BUF)

poolm = teedev->pool->dma\_buf\_mgr;

else

poolm = teedev->pool->private\_mgr;

rc = poolm->ops->alloc(poolm, shm, size);

if (rc) {

ret = ERR\_PTR(rc);

goto err\_kfree;

}

mutex\_lock(&teedev->mutex);

shm->id = idr\_alloc(&teedev->idr, shm, 1, 0, GFP\_KERNEL);

mutex\_unlock(&teedev->mutex);

if (shm->id < 0) {

ret = ERR\_PTR(shm->id);

goto err\_pool\_free;

}

if (flags & TEE\_SHM\_DMA\_BUF) {

DEFINE\_DMA\_BUF\_EXPORT\_INFO(exp\_info);

exp\_info.ops = &tee\_shm\_dma\_buf\_ops;

exp\_info.size = shm->size;

exp\_info.flags = O\_RDWR;

exp\_info.priv = shm;

shm->dmabuf = dma\_buf\_export(&exp\_info);

if (IS\_ERR(shm->dmabuf)) {

ret = ERR\_CAST(shm->dmabuf);

goto err\_rem;

}

}

if (ctx) {

teedev\_ctx\_get(ctx);

mutex\_lock(&teedev->mutex);

list\_add\_tail(&shm->link, &ctx->list\_shm);

mutex\_unlock(&teedev->mutex);

}

return shm;

err\_rem:

mutex\_lock(&teedev->mutex);

idr\_remove(&teedev->idr, shm->id);

mutex\_unlock(&teedev->mutex);

err\_pool\_free:

poolm->ops->free(poolm, shm);

err\_kfree:

kfree(shm);

err\_dev\_put:

tee\_device\_put(teedev);

return ret;

}

/\*\*

\* tee\_shm\_alloc() - Allocate shared memory

\* @ctx: Context that allocates the shared memory

\* @size: Requested size of shared memory

\* @flags: Flags setting properties for the requested shared memory.

\*

\* Memory allocated as global shared memory is automatically freed when the

\* TEE file pointer is closed. The @flags field uses the bits defined by

\* TEE\_SHM\_\* in <linux/tee\_drv.h>. TEE\_SHM\_MAPPED must currently always be

\* set. If TEE\_SHM\_DMA\_BUF global shared memory will be allocated and

\* associated with a dma-buf handle, else driver private memory.

\*/

struct tee\_shm \*tee\_shm\_alloc(struct tee\_context \*ctx, size\_t size, u32 flags)

{

return \_\_tee\_shm\_alloc(ctx, ctx->teedev, size, flags);

}

EXPORT\_SYMBOL\_GPL(tee\_shm\_alloc);

struct tee\_shm \*tee\_shm\_priv\_alloc(struct tee\_device \*teedev, size\_t size)

{

return \_\_tee\_shm\_alloc(NULL, teedev, size, TEE\_SHM\_MAPPED);

}

EXPORT\_SYMBOL\_GPL(tee\_shm\_priv\_alloc);

struct tee\_shm \*tee\_shm\_register(struct tee\_context \*ctx, unsigned long addr,

size\_t length, u32 flags)

{

struct tee\_device \*teedev = ctx->teedev;

const u32 req\_flags = TEE\_SHM\_DMA\_BUF | TEE\_SHM\_USER\_MAPPED;

struct tee\_shm \*shm;

void \*ret;

int rc;

int num\_pages;

unsigned long start;

if (flags != req\_flags)

return ERR\_PTR(-ENOTSUPP);

if (!tee\_device\_get(teedev))

return ERR\_PTR(-EINVAL);

if (!teedev->desc->ops->shm\_register ||

!teedev->desc->ops->shm\_unregister) {

tee\_device\_put(teedev);

return ERR\_PTR(-ENOTSUPP);

}

teedev\_ctx\_get(ctx);

shm = kzalloc(sizeof(\*shm), GFP\_KERNEL);

if (!shm) {

ret = ERR\_PTR(-ENOMEM);

goto err;

}

shm->flags = flags | TEE\_SHM\_REGISTER;

shm->teedev = teedev;

shm->ctx = ctx;

shm->id = -1;

start = rounddown(addr, PAGE\_SIZE);

shm->offset = addr - start;

shm->size = length;

num\_pages = (roundup(addr + length, PAGE\_SIZE) - start) / PAGE\_SIZE;

shm->pages = kcalloc(num\_pages, sizeof(\*shm->pages), GFP\_KERNEL);

if (!shm->pages) {

ret = ERR\_PTR(-ENOMEM);

goto err;

}

rc = get\_user\_pages\_fast(start, num\_pages, 1, shm->pages);

if (rc > 0)

shm->num\_pages = rc;

if (rc != num\_pages) {

if (rc >= 0)

rc = -ENOMEM;

ret = ERR\_PTR(rc);

goto err;

}

mutex\_lock(&teedev->mutex);

shm->id = idr\_alloc(&teedev->idr, shm, 1, 0, GFP\_KERNEL);

mutex\_unlock(&teedev->mutex);

if (shm->id < 0) {

ret = ERR\_PTR(shm->id);

goto err;

}

rc = teedev->desc->ops->shm\_register(ctx, shm, shm->pages,

shm->num\_pages, start);

if (rc) {

ret = ERR\_PTR(rc);

goto err;

}

if (flags & TEE\_SHM\_DMA\_BUF) {

DEFINE\_DMA\_BUF\_EXPORT\_INFO(exp\_info);

exp\_info.ops = &tee\_shm\_dma\_buf\_ops;

exp\_info.size = shm->size;

exp\_info.flags = O\_RDWR;

exp\_info.priv = shm;

shm->dmabuf = dma\_buf\_export(&exp\_info);

if (IS\_ERR(shm->dmabuf)) {

ret = ERR\_CAST(shm->dmabuf);

teedev->desc->ops->shm\_unregister(ctx, shm);

goto err;

}

}

mutex\_lock(&teedev->mutex);

list\_add\_tail(&shm->link, &ctx->list\_shm);

mutex\_unlock(&teedev->mutex);

return shm;

err:

if (shm) {

size\_t n;

if (shm->id >= 0) {

mutex\_lock(&teedev->mutex);

idr\_remove(&teedev->idr, shm->id);

mutex\_unlock(&teedev->mutex);

}

if (shm->pages) {

for (n = 0; n < shm->num\_pages; n++)

put\_page(shm->pages[n]);

kfree(shm->pages);

}

}

kfree(shm);

teedev\_ctx\_put(ctx);

tee\_device\_put(teedev);

return ret;

}

EXPORT\_SYMBOL\_GPL(tee\_shm\_register);

/\*\*

\* tee\_shm\_get\_fd() - Increase reference count and return file descriptor

\* @shm: Shared memory handle

\* @returns user space file descriptor to shared memory

\*/

int tee\_shm\_get\_fd(struct tee\_shm \*shm)

{

int fd;

if (!(shm->flags & TEE\_SHM\_DMA\_BUF))

return -EINVAL;

get\_dma\_buf(shm->dmabuf);

fd = dma\_buf\_fd(shm->dmabuf, O\_CLOEXEC);

if (fd < 0)

dma\_buf\_put(shm->dmabuf);

return fd;

}

/\*\*

\* tee\_shm\_free() - Free shared memory

\* @shm: Handle to shared memory to free

\*/

void tee\_shm\_free(struct tee\_shm \*shm)

{

/\*

\* dma\_buf\_put() decreases the dmabuf reference counter and will

\* call tee\_shm\_release() when the last reference is gone.

\*

\* In the case of driver private memory we call tee\_shm\_release

\* directly instead as it doesn't have a reference counter.

\*/

if (shm->flags & TEE\_SHM\_DMA\_BUF)

dma\_buf\_put(shm->dmabuf);

else

tee\_shm\_release(shm);

}

EXPORT\_SYMBOL\_GPL(tee\_shm\_free);

/\*\*

\* tee\_shm\_va2pa() - Get physical address of a virtual address

\* @shm: Shared memory handle

\* @va: Virtual address to tranlsate

\* @pa: Returned physical address

\* @returns 0 on success and < 0 on failure

\*/

int tee\_shm\_va2pa(struct tee\_shm \*shm, void \*va, phys\_addr\_t \*pa)

{

if (!(shm->flags & TEE\_SHM\_MAPPED))

return -EINVAL;

/\* Check that we're in the range of the shm \*/

if ((char \*)va < (char \*)shm->kaddr)

return -EINVAL;

if ((char \*)va >= ((char \*)shm->kaddr + shm->size))

return -EINVAL;

return tee\_shm\_get\_pa(

shm, (unsigned long)va - (unsigned long)shm->kaddr, pa);

}

EXPORT\_SYMBOL\_GPL(tee\_shm\_va2pa);

/\*\*

\* tee\_shm\_pa2va() - Get virtual address of a physical address

\* @shm: Shared memory handle

\* @pa: Physical address to tranlsate

\* @va: Returned virtual address

\* @returns 0 on success and < 0 on failure

\*/

int tee\_shm\_pa2va(struct tee\_shm \*shm, phys\_addr\_t pa, void \*\*va)

{

if (!(shm->flags & TEE\_SHM\_MAPPED))

return -EINVAL;

/\* Check that we're in the range of the shm \*/

if (pa < shm->paddr)

return -EINVAL;

if (pa >= (shm->paddr + shm->size))

return -EINVAL;

if (va) {

void \*v = tee\_shm\_get\_va(shm, pa - shm->paddr);

if (IS\_ERR(v))

return PTR\_ERR(v);

\*va = v;

}

return 0;

}

EXPORT\_SYMBOL\_GPL(tee\_shm\_pa2va);

/\*\*

\* tee\_shm\_get\_va() - Get virtual address of a shared memory plus an offset

\* @shm: Shared memory handle

\* @offs: Offset from start of this shared memory

\* @returns virtual address of the shared memory + offs if offs is within

\* the bounds of this shared memory, else an ERR\_PTR

\*/

void \*tee\_shm\_get\_va(struct tee\_shm \*shm, size\_t offs)

{

if (!(shm->flags & TEE\_SHM\_MAPPED))

return ERR\_PTR(-EINVAL);

if (offs >= shm->size)

return ERR\_PTR(-EINVAL);

return (char \*)shm->kaddr + offs;

}

EXPORT\_SYMBOL\_GPL(tee\_shm\_get\_va);

/\*\*

\* tee\_shm\_get\_pa() - Get physical address of a shared memory plus an offset

\* @shm: Shared memory handle

\* @offs: Offset from start of this shared memory

\* @pa: Physical address to return

\* @returns 0 if offs is within the bounds of this shared memory, else an

\* error code.

\*/

int tee\_shm\_get\_pa(struct tee\_shm \*shm, size\_t offs, phys\_addr\_t \*pa)

{

if (offs >= shm->size)

return -EINVAL;

if (pa)

\*pa = shm->paddr + offs;

return 0;

}

EXPORT\_SYMBOL\_GPL(tee\_shm\_get\_pa);

/\*\*

\* tee\_shm\_get\_from\_id() - Find shared memory object and increase reference

\* count

\* @ctx: Context owning the shared memory

\* @id: Id of shared memory object

\* @returns a pointer to 'struct tee\_shm' on success or an ERR\_PTR on failure

\*/

struct tee\_shm \*tee\_shm\_get\_from\_id(struct tee\_context \*ctx, int id)

{

struct tee\_device \*teedev;

struct tee\_shm \*shm;

if (!ctx)

return ERR\_PTR(-EINVAL);

teedev = ctx->teedev;

mutex\_lock(&teedev->mutex);

shm = idr\_find(&teedev->idr, id);

if (!shm || shm->ctx != ctx)

shm = ERR\_PTR(-EINVAL);

else if (shm->flags & TEE\_SHM\_DMA\_BUF)

get\_dma\_buf(shm->dmabuf);

mutex\_unlock(&teedev->mutex);

return shm;

}

EXPORT\_SYMBOL\_GPL(tee\_shm\_get\_from\_id);

/\*\*

\* tee\_shm\_put() - Decrease reference count on a shared memory handle

\* @shm: Shared memory handle

\*/

void tee\_shm\_put(struct tee\_shm \*shm)

{

if (shm->flags & TEE\_SHM\_DMA\_BUF)

dma\_buf\_put(shm->dmabuf);

}

EXPORT\_SYMBOL\_GPL(tee\_shm\_put);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

tee\_shm\_pool.c

/\*

\* Copyright (c) 2015, Linaro Limited

\*

\* This software is licensed under the terms of the GNU General Public

\* License version 2, as published by the Free Software Foundation, and

\* may be copied, distributed, and modified under those terms.

\*

\* This program is distributed in the hope that it will be useful,

\* but WITHOUT ANY WARRANTY; without even the implied warranty of

\* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

\* GNU General Public License for more details.

\*

\*/

#include <linux/device.h>

#include <linux/dma-buf.h>

#include <linux/genalloc.h>

#include <linux/slab.h>

#include <linux/tee\_drv.h>

#include "tee\_private.h"

static int pool\_op\_gen\_alloc(struct tee\_shm\_pool\_mgr \*poolm,

struct tee\_shm \*shm, size\_t size)

{

unsigned long va;

struct gen\_pool \*genpool = poolm->private\_data;

size\_t s = roundup(size, 1 << genpool->min\_alloc\_order);

va = gen\_pool\_alloc(genpool, s);

if (!va)

return -ENOMEM;

memset((void \*)va, 0, s);

shm->kaddr = (void \*)va;

shm->paddr = gen\_pool\_virt\_to\_phys(genpool, va);

shm->size = s;

return 0;

}

static void pool\_op\_gen\_free(struct tee\_shm\_pool\_mgr \*poolm,

struct tee\_shm \*shm)

{

gen\_pool\_free(poolm->private\_data, (unsigned long)shm->kaddr,

shm->size);

shm->kaddr = NULL;

}

static void pool\_op\_gen\_destroy\_poolmgr(struct tee\_shm\_pool\_mgr \*poolm)

{

gen\_pool\_destroy(poolm->private\_data);

kfree(poolm);

}

static const struct tee\_shm\_pool\_mgr\_ops pool\_ops\_generic = {

.alloc = pool\_op\_gen\_alloc,

.free = pool\_op\_gen\_free,

.destroy\_poolmgr = pool\_op\_gen\_destroy\_poolmgr,

};

/\*\*

\* tee\_shm\_pool\_alloc\_res\_mem() - Create a shared memory pool from reserved

\* memory range

\* @priv\_info: Information for driver private shared memory pool

\* @dmabuf\_info: Information for dma-buf shared memory pool

\*

\* Start and end of pools will must be page aligned.

\*

\* Allocation with the flag TEE\_SHM\_DMA\_BUF set will use the range supplied

\* in @dmabuf, others will use the range provided by @priv.

\*

\* @returns pointer to a 'struct tee\_shm\_pool' or an ERR\_PTR on failure.

\*/

struct tee\_shm\_pool \*

tee\_shm\_pool\_alloc\_res\_mem(struct tee\_shm\_pool\_mem\_info \*priv\_info,

struct tee\_shm\_pool\_mem\_info \*dmabuf\_info)

{

struct tee\_shm\_pool\_mgr \*priv\_mgr;

struct tee\_shm\_pool\_mgr \*dmabuf\_mgr;

void \*rc;

/\*

\* Create the pool for driver private shared memory

\*/

rc = tee\_shm\_pool\_mgr\_alloc\_res\_mem(priv\_info->vaddr, priv\_info->paddr,

priv\_info->size,

3 /\* 8 byte aligned \*/);

if (IS\_ERR(rc))

return rc;

priv\_mgr = rc;

/\*

\* Create the pool for dma\_buf shared memory

\*/

rc = tee\_shm\_pool\_mgr\_alloc\_res\_mem(dmabuf\_info->vaddr,

dmabuf\_info->paddr,

dmabuf\_info->size, PAGE\_SHIFT);

if (IS\_ERR(rc))

goto err\_free\_priv\_mgr;

dmabuf\_mgr = rc;

rc = tee\_shm\_pool\_alloc(priv\_mgr, dmabuf\_mgr);

if (IS\_ERR(rc))

goto err\_free\_dmabuf\_mgr;

return rc;

err\_free\_dmabuf\_mgr:

tee\_shm\_pool\_mgr\_destroy(dmabuf\_mgr);

err\_free\_priv\_mgr:

tee\_shm\_pool\_mgr\_destroy(priv\_mgr);

return rc;

}

EXPORT\_SYMBOL\_GPL(tee\_shm\_pool\_alloc\_res\_mem);

struct tee\_shm\_pool\_mgr \*tee\_shm\_pool\_mgr\_alloc\_res\_mem(unsigned long vaddr,

phys\_addr\_t paddr,

size\_t size,

int min\_alloc\_order)

{

const size\_t page\_mask = PAGE\_SIZE - 1;

struct tee\_shm\_pool\_mgr \*mgr;

int rc;

/\* Start and end must be page aligned \*/

if (vaddr & page\_mask || paddr & page\_mask || size & page\_mask)

return ERR\_PTR(-EINVAL);

mgr = kzalloc(sizeof(\*mgr), GFP\_KERNEL);

if (!mgr)

return ERR\_PTR(-ENOMEM);

mgr->private\_data = gen\_pool\_create(min\_alloc\_order, -1);

if (!mgr->private\_data) {

rc = -ENOMEM;

goto err;

}

gen\_pool\_set\_algo(mgr->private\_data, gen\_pool\_best\_fit, NULL);

rc = gen\_pool\_add\_virt(mgr->private\_data, vaddr, paddr, size, -1);

if (rc) {

gen\_pool\_destroy(mgr->private\_data);

goto err;

}

mgr->ops = &pool\_ops\_generic;

return mgr;

err:

kfree(mgr);

return ERR\_PTR(rc);

}

EXPORT\_SYMBOL\_GPL(tee\_shm\_pool\_mgr\_alloc\_res\_mem);

static bool check\_mgr\_ops(struct tee\_shm\_pool\_mgr \*mgr)

{

return mgr && mgr->ops && mgr->ops->alloc && mgr->ops->free &&

mgr->ops->destroy\_poolmgr;

}

struct tee\_shm\_pool \*tee\_shm\_pool\_alloc(struct tee\_shm\_pool\_mgr \*priv\_mgr,

struct tee\_shm\_pool\_mgr \*dmabuf\_mgr)

{

struct tee\_shm\_pool \*pool;

if (!check\_mgr\_ops(priv\_mgr) || !check\_mgr\_ops(dmabuf\_mgr))

return ERR\_PTR(-EINVAL);

pool = kzalloc(sizeof(\*pool), GFP\_KERNEL);

if (!pool)

return ERR\_PTR(-ENOMEM);

pool->private\_mgr = priv\_mgr;

pool->dma\_buf\_mgr = dmabuf\_mgr;

return pool;

}

EXPORT\_SYMBOL\_GPL(tee\_shm\_pool\_alloc);

/\*\*

\* tee\_shm\_pool\_free() - Free a shared memory pool

\* @pool: The shared memory pool to free

\*

\* There must be no remaining shared memory allocated from this pool when

\* this function is called.

\*/

void tee\_shm\_pool\_free(struct tee\_shm\_pool \*pool)

{

if (pool->private\_mgr)

tee\_shm\_pool\_mgr\_destroy(pool->private\_mgr);

if (pool->dma\_buf\_mgr)

tee\_shm\_pool\_mgr\_destroy(pool->dma\_buf\_mgr);

kfree(pool);

}

EXPORT\_SYMBOL\_GPL(tee\_shm\_pool\_free);