Linux Target Documentation

The kernel development community

Jan 15, 2023

CONTENTS

1 TCM Userspace Design 1

2 The TCM v4 fabric module script generator 9

3 TCM mod builder script 13

4 Target export device script 29

CHAPTER

ONE

TCM USERSPACE DESIGN

1.1 Design

TCM is another name for LIO, an in-kernel iSCSI target (server). Existing TCM targets run in the kernel. TCMU (TCM in Userspace) allows userspace programs to be written which act as iSCSI targets. This document describes the design.

The existing kernel provides modules for different SCSI transport protocols. TCM also modularizes the data storage. There are existing modules for file, block device, RAM or using another SCSI device as storage. These are called "backstores" or "storage engines". These built-in modules are implemented entirely as kernel code.

1.1.1 Background

In addition to modularizing the transport protocol used for carrying SCSI commands ("fabrics"), the Linux kernel target, LIO, also modularizes the actual data storage as well. These are referred to as "backstores" or "storage engines". The target comes with backstores that allow a file, a block device, RAM, or another SCSI device to be used for the local storage needed for the exported SCSI LUN. Like the rest of LIO, these are implemented entirely as kernel code.

These backstores cover the most common use cases, but not all. One new use case that other non-kernel target solutions, such as tgt, are able to support is using Gluster's GLFS or Ceph's RBD as a backstore. The target then serves as a translator, allowing initiators to store data in these non-traditional networked storage systems, while still only using standard protocols themselves.

If the target is a userspace process, supporting these is easy. tgt, for example, needs only a small adapter module for each, because the modules just use the available userspace libraries for RBD and GLFS.

Adding support for these backstores in LIO is considerably more difficult, because LIO is entirely kernel code. Instead of undertaking the significant work to port the GLFS or RBD APIs and protocols to the kernel, another approach is to create a userspace pass-through backstore for LIO, "TCMU".

1.1.2 Benefits

In addition to allowing relatively easy support for RBD and GLFS, TCMU will also allow easier development of new backstores. TCMU combines with the LIO loopback fabric to become something similar to FUSE (Filesystem in Userspace), but at the SCSI layer instead of the filesystem layer. A SUSE, if you will.

The disadvantage is there are more distinct components to configure, and potentially to malfunction. This is unavoidable, but hopefully not fatal if we're careful to keep things as simple as possible.

1.1.3 Design constraints

- Good performance: high throughput, low latency
- Cleanly handle if userspace:
  1) never attaches
  2) hangs
  3) dies
  4) misbehaves
- Allow future flexibility in user & kernel implementations
- Be reasonably memory-efficient
- Simple to configure & run
- Simple to write a userspace backend

1.1.4 Implementation overview

The core of the TCMU interface is a memory region that is shared between kernel and userspace. Within this region is: a control area (mailbox); a lockless producer/consumer circular buffer for commands to be passed up, and status returned; and an in/out data buffer area.

TCMU uses the pre-existing UIO subsystem. UIO allows device driver development in userspace, and this is conceptually very close to the TCMU use case, except instead of a physical device, TCMU implements a memory-mapped layout designed for SCSI commands. Using UIO also benefits TCMU by handling device introspection (e.g. a way for userspace to determine how large the shared region is) and signaling mechanisms in both directions.

There are no embedded pointers in the memory region. Everything is expressed as an offset from the region's starting address. This allows the ring to still work if the user process dies and is restarted with the region mapped at a different virtual address.

See target_core_user.h for the struct definitions.

1.1.5 The Mailbox

The mailbox is always at the start of the shared memory region, and contains a version, details about the starting offset and size of the command ring, and head and tail pointers to be used by the kernel and userspace (respectively) to put commands on the ring, and indicate when the commands are completed.

version - 1 (userspace should abort if otherwise)

flags:

- **TCMU_MAILBOX_FLAG_CAP_OOOC:** indicates out-of-order completion is supported. See "The Command Ring" for details.

cmdr_off The offset of the start of the command ring from the start of the memory region, to account for the mailbox size.

cmdr_size The size of the command ring. This does *not* need to be a power of two.

cmd_head Modified by the kernel to indicate when a command has been placed on the ring.

cmd_tail Modified by userspace to indicate when it has completed processing of a command.

1.1.6 The Command Ring

Commands are placed on the ring by the kernel incrementing mailbox.cmd_head by the size of the command, modulo cmdr_size, and then signaling userspace via uio_event_notify(). Once the command is completed, userspace updates mailbox.cmd_tail in the same way and signals the kernel via a 4-byte write(). When cmd_head equals cmd_tail, the ring is empty – no commands are currently waiting to be processed by userspace.

TCMU commands are 8-byte aligned. They start with a common header containing "len_op", a 32-bit value that stores the length, as well as the opcode in the lowest unused bits. It also contains cmd_id and flags fields for setting by the kernel (kflags) and userspace (uflags).

Currently only two opcodes are defined, TCMU_OP_CMD and TCMU_OP_PAD.

When the opcode is CMD, the entry in the command ring is a struct tcmu_cmd_entry. Userspace finds the SCSI CDB (Command Data Block) via tcmu_cmd_entry.req.cdb_off. This is an offset from the start of the overall shared memory region, not the entry. The data in/out buffers are accessible via tht req.iov[] array. iov_cnt contains the number of entries in iov[] needed to describe either the Data-In or Data-Out buffers. For bidirectional commands, iov_cnt specifies how many iovec entries cover the Data-Out area, and iov_bidi_cnt specifies how many iovec entries immediately after that in iov[] cover the Data-In area. Just like other fields, iov.iov_base is an offset from the start of the region.

When completing a command, userspace sets rsp.scsi_status, and rsp.sense_buffer if necessary. Userspace then increments mailbox.cmd_tail by entry.hdr.length (mod cmdr_size) and signals the kernel via the UIO method, a 4-byte write to the file descriptor.

If TCMU_MAILBOX_FLAG_CAP_OOOC is set for mailbox->flags, kernel is capable of handling out-of-order completions. In this case, userspace can handle command in different order other than original. Since kernel would still process the commands in the same order it appeared in the command ring, userspace need to update the cmd->id when completing the command(a.k.a steal the original command's entry).

When the opcode is PAD, userspace only updates cmd_tail as above – it's a no-op. (The kernel inserts PAD entries to ensure each CMD entry is contiguous within the command ring.)

More opcodes may be added in the future. If userspace encounters an opcode it does not handle, it must set UNKNOWN_OP bit (bit 0) in hdr.uflags, update cmd_tail, and proceed with processing additional commands, if any.

1.1.7 The Data Area

This is shared-memory space after the command ring. The organization of this area is not defined in the TCMU interface, and userspace should access only the parts referenced by pending iovs.

1.1.8 Device Discovery

Other devices may be using UIO besides TCMU. Unrelated user processes may also be handling different sets of TCMU devices. TCMU userspace processes must find their devices by scanning sysfs class/uio/uio*/name. For TCMU devices, these names will be of the format:

```
tcm-user/<hba_num>/<device_name>/<subtype>/<path>
```

where "tcm-user" is common for all TCMU-backed UIO devices. <hba_num> and <device_name> allow userspace to find the device's path in the kernel target's configfs tree. Assuming the usual mount point, it is found at:

```
/sys/kernel/config/target/core/user_<hba_num>/<device_name>
```

This location contains attributes such as "hw_block_size", that userspace needs to know for correct operation.

<subtype> will be a userspace-process-unique string to identify the TCMU device as expecting to be backed by a certain handler, and <path> will be an additional handler-specific string for the user process to configure the device, if needed. The name cannot contain ':', due to LIO limitations.

For all devices so discovered, the user handler opens /dev/uioX and calls mmap():

```
mmap(NULL, size, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0)
```

where size must be equal to the value read from /sys/class/uio/uioX/maps/map0/size.

1.1.9 Device Events

If a new device is added or removed, a notification will be broadcast over netlink, using a generic netlink family name of "TCM-USER" and a multicast group named "config". This will include the UIO name as described in the previous section, as well as the UIO minor number. This should allow userspace to identify both the UIO device and the LIO device, so that after determining the device is supported (based on subtype) it can take the appropriate action.

1.1.10 Other contingencies

Userspace handler process never attaches:

- TCMU will post commands, and then abort them after a timeout period (30 seconds.)

Userspace handler process is killed:

- It is still possible to restart and re-connect to TCMU devices. Command ring is preserved. However, after the timeout period, the kernel will abort pending tasks.

Userspace handler process hangs:

- The kernel will abort pending tasks after a timeout period.

Userspace handler process is malicious:

- The process can trivially break the handling of devices it controls, but should not be able to access kernel memory outside its shared memory areas.

1.2 Writing a user pass-through handler (with example code)

A user process handing a TCMU device must support the following:

a) Discovering and configuring TCMU uio devices

b) Waiting for events on the device(s)

c) Managing the command ring: Parsing operations and commands, performing work as needed, setting response fields (scsi_status and possibly sense_buffer), updating cmd_tail, and notifying the kernel that work has been finished

First, consider instead writing a plugin for tcmu-runner. tcmu-runner implements all of this, and provides a higher-level API for plugin authors.

TCMU is designed so that multiple unrelated processes can manage TCMU devices separately. All handlers should make sure to only open their devices, based opon a known subtype string.

a) Discovering and configuring TCMU UIO devices:

```
/* error checking omitted for brevity */

int fd, dev_fd;
char buf[256];
unsigned long long map_len;
void *map;

fd = open("/sys/class/uio/uio0/name", O_RDONLY);
ret = read(fd, buf, sizeof(buf));
close(fd);
buf[ret-1] = '\0'; /* null-terminate and chop off the \n */

/* we only want uio devices whose name is a format we expect */
if (strncmp(buf, "tcm-user", 8))
  exit(-1);
```

```
/* Further checking for subtype also needed here */

fd = open(/sys/class/uio/%s/maps/map0/size, O_RDONLY);
ret = read(fd, buf, sizeof(buf));
close(fd);
str_buf[ret-1] = '\0'; /* null-terminate and chop off the \n */

map_len = strtoull(buf, NULL, 0);

dev_fd = open("/dev/uio0", O_RDWR);
map = mmap(NULL, map_len, PROT_READ|PROT_WRITE, MAP_SHARED, dev_fd, 0);


b) Waiting for events on the device(s)

while (1) {
  char buf[4];

  int ret = read(dev_fd, buf, 4); /* will block */

  handle_device_events(dev_fd, map);
}
```

c) Managing the command ring:

```
#include <linux/target_core_user.h>

int handle_device_events(int fd, void *map)
{
  struct tcmu_mailbox *mb = map;
  struct tcmu_cmd_entry *ent = (void *) mb + mb->cmdr_off + mb->cmd_tail;
  int did_some_work = 0;

  /* Process events from cmd ring until we catch up with cmd_head */
  while (ent != (void *)mb + mb->cmdr_off + mb->cmd_head) {

    if (tcmu_hdr_get_op(ent->hdr.len_op) == TCMU_OP_CMD) {
      uint8_t *cdb = (void *)mb + ent->req.cdb_off;
      bool success = true;

      /* Handle command here. */
      printf("SCSI opcode: 0x%x\n", cdb[0]);

      /* Set response fields */
      if (success)
        ent->rsp.scsi_status = SCSI_NO_SENSE;
      else {
        /* Also fill in rsp->sense_buffer here */
        ent->rsp.scsi_status = SCSI_CHECK_CONDITION;
      }
    }
```

```
    else if (tcmu_hdr_get_op(ent->hdr.len_op) != TCMU_OP_PAD) {
      /* Tell the kernel we didn't handle unknown opcodes */
      ent->hdr.uflags |= TCMU_UFLAG_UNKNOWN_OP;
    }
    else {
      /* Do nothing for PAD entries except update cmd_tail */
    }

    /* update cmd_tail */
    mb->cmd_tail = (mb->cmd_tail + tcmu_hdr_get_len(&ent->hdr)) % mb->cmdr_
↪size;
    ent = (void *) mb + mb->cmdr_off + mb->cmd_tail;
    did_some_work = 1;
  }

  /* Notify the kernel that work has been finished */
  if (did_some_work) {
    uint32_t buf = 0;

    write(fd, &buf, 4);
  }

  return 0;
}
```

1.3 A final note

Please be careful to return codes as defined by the SCSI specifications. These are different than some values defined in the scsi/scsi.h include file. For example, CHECK CONDITION's status code is 2, not 1.

CHAPTER

TWO

THE TCM V4 FABRIC MODULE SCRIPT GENERATOR

Greetings all,

This document is intended to be a mini-HOWTO for using the tcm_mod_builder.py script to generate a brand new functional TCM v4 fabric .ko module of your very own, that once built can be immediately be loaded to start access the new TCM/ConfigFS fabric skeleton, by simply using:

```
modprobe $TCM_NEW_MOD
mkdir -p /sys/kernel/config/target/$TCM_NEW_MOD
```

This script will create a new drivers/target/$TCM_NEW_MOD/, and will do the following

1) Generate new API callers for drivers/target/target_core_fabric_configs.c logic ->make_tpg(), ->drop_tpg(), ->make_wwn(), ->drop_wwn(). These are created into $TCM_NEW_MOD/$TCM_NEW_MOD_configfs.c
2) Generate basic infrastructure for loading/unloading LKMs and TCM/ConfigFS fabric module using a skeleton struct target_core_fabric_ops API template.
3) Based on user defined T10 Proto_Ident for the new fabric module being built, the TransportID / Initiator and Target WWPN related handlers for SPC-3 persistent reservation are automatically generated in $TCM_NEW_MOD/$TCM_NEW_MOD_fabric.c using drivers/target/target_core_fabric_lib.c logic.
4) NOP API calls for all other Data I/O path and fabric dependent attribute logic in $TCM_NEW_MOD/$TCM_NEW_MOD_fabric.c

tcm_mod_builder.py depends upon the mandatory '-p $PROTO_IDENT' and '-m $FABRIC_MOD_name' parameters, and actually running the script looks like:

```
target:/mnt/sdb/lio-core-2.6.git/Documentation/target# python tcm_mod_builder.
 ↪py -p iSCSI -m tcm_nab5000
tcm_dir: /mnt/sdb/lio-core-2.6.git/Documentation/target/../../
Set fabric_mod_name: tcm_nab5000
Set fabric_mod_dir:
/mnt/sdb/lio-core-2.6.git/Documentation/target/../../drivers/target/tcm_nab5000
Using proto_ident: iSCSI
Creating fabric_mod_dir:
/mnt/sdb/lio-core-2.6.git/Documentation/target/../../drivers/target/tcm_nab5000
Writing file:
/mnt/sdb/lio-core-2.6.git/Documentation/target/../../drivers/target/tcm_
 ↪nab5000/tcm_nab5000_base.h
Using tcm_mod_scan_fabric_ops:
```

```
/mnt/sdb/lio-core-2.6.git/Documentation/target/../../include/target/target_
 ↪core_fabric_ops.h
Writing file:
/mnt/sdb/lio-core-2.6.git/Documentation/target/../../drivers/target/tcm_
 ↪nab5000/tcm_nab5000_fabric.c
Writing file:
/mnt/sdb/lio-core-2.6.git/Documentation/target/../../drivers/target/tcm_
 ↪nab5000/tcm_nab5000_fabric.h
Writing file:
/mnt/sdb/lio-core-2.6.git/Documentation/target/../../drivers/target/tcm_
 ↪nab5000/tcm_nab5000_configfs.c
Writing file:
/mnt/sdb/lio-core-2.6.git/Documentation/target/../../drivers/target/tcm_
 ↪nab5000/Kbuild
Writing file:
/mnt/sdb/lio-core-2.6.git/Documentation/target/../../drivers/target/tcm_
 ↪nab5000/Kconfig
Would you like to add tcm_nab5000to drivers/target/Kbuild..? [yes,no]: yes
Would you like to add tcm_nab5000to drivers/target/Kconfig..? [yes,no]: yes
```

At the end of tcm_mod_builder.py. the script will ask to add the following line to drivers/target/Kbuild:

```
obj-$(CONFIG_TCM_NAB5000)       += tcm_nab5000/
```

and the same for drivers/target/Kconfig:

```
source "drivers/target/tcm_nab5000/Kconfig"
```

1) Run 'make menuconfig' and select the new CONFIG_TCM_NAB5000 item:

```
<M>   TCM_NAB5000 fabric module
```

2) Build using 'make modules', once completed you will have:

```
target:/mnt/sdb/lio-core-2.6.git# ls -la drivers/target/tcm_nab5000/
total 1348
drwxr-xr-x 2 root root   4096 2010-10-05 03:23 .
drwxr-xr-x 9 root root   4096 2010-10-05 03:22 ..
-rw-r--r-- 1 root root    282 2010-10-05 03:22 Kbuild
-rw-r--r-- 1 root root    171 2010-10-05 03:22 Kconfig
-rw-r--r-- 1 root root     49 2010-10-05 03:23 modules.order
-rw-r--r-- 1 root root    738 2010-10-05 03:22 tcm_nab5000_base.h
-rw-r--r-- 1 root root   9096 2010-10-05 03:22 tcm_nab5000_configfs.c
-rw-r--r-- 1 root root 191200 2010-10-05 03:23 tcm_nab5000_configfs.o
-rw-r--r-- 1 root root  40504 2010-10-05 03:23 .tcm_nab5000_configfs.o.cmd
-rw-r--r-- 1 root root   5414 2010-10-05 03:22 tcm_nab5000_fabric.c
-rw-r--r-- 1 root root   2016 2010-10-05 03:22 tcm_nab5000_fabric.h
-rw-r--r-- 1 root root 190932 2010-10-05 03:23 tcm_nab5000_fabric.o
-rw-r--r-- 1 root root  40713 2010-10-05 03:23 .tcm_nab5000_fabric.o.cmd
-rw-r--r-- 1 root root 401861 2010-10-05 03:23 tcm_nab5000.ko
-rw-r--r-- 1 root root    265 2010-10-05 03:23 .tcm_nab5000.ko.cmd
```

```
-rw-r--r-- 1 root root    459 2010-10-05 03:23 tcm_nab5000.mod.c
-rw-r--r-- 1 root root  23896 2010-10-05 03:23 tcm_nab5000.mod.o
-rw-r--r-- 1 root root  22655 2010-10-05 03:23 .tcm_nab5000.mod.o.cmd
-rw-r--r-- 1 root root 379022 2010-10-05 03:23 tcm_nab5000.o
-rw-r--r-- 1 root root    211 2010-10-05 03:23 .tcm_nab5000.o.cmd
```

3) Load the new module, create a lun_0 configfs group, and add new TCM Core IBLOCK backstore symlink to port:

```
target:/mnt/sdb/lio-core-2.6.git# insmod drivers/target/tcm_nab5000.ko
target:/mnt/sdb/lio-core-2.6.git# mkdir -p /sys/kernel/config/target/
 ↪nab5000/iqn.foo/tpgt_1/lun/lun_0
target:/mnt/sdb/lio-core-2.6.git# cd /sys/kernel/config/target/nab5000/iqn.
 ↪foo/tpgt_1/lun/lun_0/
target:/sys/kernel/config/target/nab5000/iqn.foo/tpgt_1/lun/lun_0# ln -s /
 ↪sys/kernel/config/target/core/iblock_0/lvm_test0 nab5000_port

target:/sys/kernel/config/target/nab5000/iqn.foo/tpgt_1/lun/lun_0# cd -
target:/mnt/sdb/lio-core-2.6.git# tree /sys/kernel/config/target/nab5000/
/sys/kernel/config/target/nab5000/
|-- discovery_auth
|-- iqn.foo
|   `-- tpgt_1
|       |-- acls
|       |-- attrib
|       |-- lun
|       |   `-- lun_0
|       |       |-- alua_tg_pt_gp
|       |       |-- alua_tg_pt_offline
|       |       |-- alua_tg_pt_status
|       |       |-- alua_tg_pt_write_md
|   |   |       `-- nab5000_port -> ../../../../../../target/core/iblock_0/lvm_
 ↪test0
|       |-- np
|       `-- param
`-- version

target:/mnt/sdb/lio-core-2.6.git# lsmod
Module                  Size  Used by
tcm_nab5000             3935  4
iscsi_target_mod      193211  0
target_core_stgt        8090  0
target_core_pscsi      11122  1
target_core_file        9172  2
target_core_iblock      9280  1
target_core_mod       228575  31
tcm_nab5000,iscsi_target_mod,target_core_stgt,target_core_pscsi,target_
 ↪core_file,target_core_iblock
libfc                  73681  0
scsi_debug             56265  0
scsi_tgt                8666  1 target_core_stgt
```

```
configfs                20644  2 target_core_mod
```

2.1 Future TODO items

1) Add more T10 proto_idents
2) Make tcm_mod_dump_fabric_ops() smarter and generate function pointer defs directly from include/target/target_core_fabric_ops.h:struct target_core_fabric_ops structure members.

October 5th, 2010

Nicholas A. Bellinger <nab@linux-iscsi.org>

CHAPTER

THREE

TCM MOD BUILDER SCRIPT

```
#!/usr/bin/env python
# The TCM v4 multi-protocol fabric module generation script for drivers/target/
 ↪$NEW_MOD
#
# Copyright (c) 2010 Rising Tide Systems
# Copyright (c) 2010 Linux-iSCSI.org
#
# Author: nab@kernel.org
#
import os, sys
import subprocess as sub
import string
import re
import optparse

tcm_dir = ""

fabric_ops = []
fabric_mod_dir = ""
fabric_mod_port = ""
fabric_mod_init_port = ""

def tcm_mod_err(msg):
        print msg
        sys.exit(1)

def tcm_mod_create_module_subdir(fabric_mod_dir_var):

        if os.path.isdir(fabric_mod_dir_var) == True:
                return 1

        print "Creating fabric_mod_dir: " + fabric_mod_dir_var
        ret = os.mkdir(fabric_mod_dir_var)
        if ret:
                tcm_mod_err("Unable to mkdir " + fabric_mod_dir_var)

        return

def tcm_mod_build_FC_include(fabric_mod_dir_var, fabric_mod_name):
```

```
        global fabric_mod_port
        global fabric_mod_init_port
        buf = ""

        f = fabric_mod_dir_var + "/" + fabric_mod_name + "_base.h"
        print "Writing file: " + f

        p = open(f, 'w');
        if not p:
                tcm_mod_err("Unable to open file: " + f)

        buf = "#define " + fabric_mod_name.upper() + "_VERSION        \"v0.1\"\
↪n"
        buf += "#define " + fabric_mod_name.upper() + "_NAMELEN        32\n"
        buf += "\n"
        buf += "struct " + fabric_mod_name + "_tpg {\n"
        buf += "        /* FC lport target portal group tag for TCM */\n"
        buf += "        u16 lport_tpgt;\n"
        buf += "        /* Pointer back to " + fabric_mod_name + "_lport */\n"
        buf += "        struct " + fabric_mod_name + "_lport *lport;\n"
        buf += "        /* Returned by " + fabric_mod_name + "_make_tpg() */\n"
        buf += "        struct se_portal_group se_tpg;\n"
        buf += "};\n"
        buf += "\n"
        buf += "struct " + fabric_mod_name + "_lport {\n"
        buf += "        /* Binary World Wide unique Port Name for FC Target␣
↪Lport */\n"
        buf += "        u64 lport_wwpn;\n"
        buf += "        /* ASCII formatted WWPN for FC Target Lport */\n"
        buf += "        char lport_name[" + fabric_mod_name.upper() + "_
↪NAMELEN];\n"
        buf += "        /* Returned by " + fabric_mod_name + "_make_lport() */\
↪n"
        buf += "        struct se_wwn lport_wwn;\n"
        buf += "};\n"

        ret = p.write(buf)
        if ret:
                tcm_mod_err("Unable to write f: " + f)

        p.close()

        fabric_mod_port = "lport"
        fabric_mod_init_port = "nport"

        return

def tcm_mod_build_SAS_include(fabric_mod_dir_var, fabric_mod_name):
        global fabric_mod_port
        global fabric_mod_init_port
```

```
        buf = ""

        f = fabric_mod_dir_var + "/" + fabric_mod_name + "_base.h"
        print "Writing file: " + f

        p = open(f, 'w');
        if not p:
                tcm_mod_err("Unable to open file: " + f)

        buf = "#define " + fabric_mod_name.upper() + "_VERSION  \"v0.1\"\n"
        buf += "#define " + fabric_mod_name.upper() + "_NAMELEN 32\n"
        buf += "\n"
        buf += "struct " + fabric_mod_name + "_tpg {\n"
        buf += "        /* SAS port target portal group tag for TCM */\n"
        buf += "        u16 tport_tpgt;\n"
        buf += "        /* Pointer back to " + fabric_mod_name + "_tport */\n"
        buf += "        struct " + fabric_mod_name + "_tport *tport;\n"
        buf += "        /* Returned by " + fabric_mod_name + "_make_tpg() */\n"
        buf += "        struct se_portal_group se_tpg;\n"
        buf += "};\n\n"
        buf += "struct " + fabric_mod_name + "_tport {\n"
        buf += "        /* Binary World Wide unique Port Name for SAS Target␣
↪port */\n"
        buf += "        u64 tport_wwpn;\n"
        buf += "        /* ASCII formatted WWPN for SAS Target port */\n"
        buf += "        char tport_name[" + fabric_mod_name.upper() + "_
↪NAMELEN];\n"
        buf += "        /* Returned by " + fabric_mod_name + "_make_tport() */\
↪n"
        buf += "        struct se_wwn tport_wwn;\n"
        buf += "};\n"

        ret = p.write(buf)
        if ret:
                tcm_mod_err("Unable to write f: " + f)

        p.close()

        fabric_mod_port = "tport"
        fabric_mod_init_port = "iport"

        return

def tcm_mod_build_iSCSI_include(fabric_mod_dir_var, fabric_mod_name):
        global fabric_mod_port
        global fabric_mod_init_port
        buf = ""

        f = fabric_mod_dir_var + "/" + fabric_mod_name + "_base.h"
        print "Writing file: " + f
```

```
        p = open(f, 'w');
        if not p:
                tcm_mod_err("Unable to open file: " + f)

        buf = "#define " + fabric_mod_name.upper() + "_VERSION  \"v0.1\"\n"
        buf += "#define " + fabric_mod_name.upper() + "_NAMELEN 32\n"
        buf += "\n"
        buf += "struct " + fabric_mod_name + "_tpg {\n"
        buf += "        /* iSCSI target portal group tag for TCM */\n"
        buf += "        u16 tport_tpgt;\n"
        buf += "        /* Pointer back to " + fabric_mod_name + "_tport */\n"
        buf += "        struct " + fabric_mod_name + "_tport *tport;\n"
        buf += "        /* Returned by " + fabric_mod_name + "_make_tpg() */\n"
        buf += "        struct se_portal_group se_tpg;\n"
        buf += "};\n\n"
        buf += "struct " + fabric_mod_name + "_tport {\n"
        buf += "        /* ASCII formatted TargetName for IQN */\n"
        buf += "        char tport_name[" + fabric_mod_name.upper() + "_
↪NAMELEN];\n"
        buf += "        /* Returned by " + fabric_mod_name + "_make_tport() */\
↪n"
        buf += "        struct se_wwn tport_wwn;\n"
        buf += "};\n"

        ret = p.write(buf)
        if ret:
                tcm_mod_err("Unable to write f: " + f)

        p.close()

        fabric_mod_port = "tport"
        fabric_mod_init_port = "iport"

        return

def tcm_mod_build_base_includes(proto_ident, fabric_mod_dir_val, fabric_mod_
↪name):

        if proto_ident == "FC":
                tcm_mod_build_FC_include(fabric_mod_dir_val, fabric_mod_name)
        elif proto_ident == "SAS":
                tcm_mod_build_SAS_include(fabric_mod_dir_val, fabric_mod_name)
        elif proto_ident == "iSCSI":
                tcm_mod_build_iSCSI_include(fabric_mod_dir_val, fabric_mod_
↪name)
        else:
                print "Unsupported proto_ident: " + proto_ident
                sys.exit(1)
```

```
        return

def tcm_mod_build_configfs(proto_ident, fabric_mod_dir_var, fabric_mod_name):
        buf = ""

        f = fabric_mod_dir_var + "/" + fabric_mod_name + "_configfs.c"
        print "Writing file: " + f

        p = open(f, 'w');
        if not p:
                tcm_mod_err("Unable to open file: " + f)

        buf = "#include <linux/module.h>\n"
        buf += "#include <linux/moduleparam.h>\n"
        buf += "#include <linux/version.h>\n"
        buf += "#include <generated/utsrelease.h>\n"
        buf += "#include <linux/utsname.h>\n"
        buf += "#include <linux/init.h>\n"
        buf += "#include <linux/slab.h>\n"
        buf += "#include <linux/kthread.h>\n"
        buf += "#include <linux/types.h>\n"
        buf += "#include <linux/string.h>\n"
        buf += "#include <linux/configfs.h>\n"
        buf += "#include <linux/ctype.h>\n"
        buf += "#include <asm/unaligned.h>\n"
        buf += "#include <scsi/scsi_proto.h>\n\n"
        buf += "#include <target/target_core_base.h>\n"
        buf += "#include <target/target_core_fabric.h>\n"
        buf += "#include \"" + fabric_mod_name + "_base.h\"\n"
        buf += "#include \"" + fabric_mod_name + "_fabric.h\"\n\n"

        buf += "static const struct target_core_fabric_ops " + fabric_mod_name␣
↪+ "_ops;\n\n"

        buf += "static struct se_portal_group *" + fabric_mod_name + "_make_
↪tpg(\n"
        buf += "        struct se_wwn *wwn,\n"
        buf += "        struct config_group *group,\n"
        buf += "        const char *name)\n"
        buf += "{\n"
        buf += "        struct " + fabric_mod_name + "_" + fabric_mod_port + "*
↪" + fabric_mod_port + " = container_of(wwn,\n"
        buf += "                        struct " + fabric_mod_name + "_" +␣
↪fabric_mod_port + ", " + fabric_mod_port + "_wwn);\n\n"
        buf += "        struct " + fabric_mod_name + "_tpg *tpg;\n"
        buf += "        unsigned long tpgt;\n"
        buf += "        int ret;\n\n"
        buf += "        if (strstr(name, \"tpgt_\") != name)\n"
        buf += "                return ERR_PTR(-EINVAL);\n"
        buf += "        if (kstrtoul(name + 5, 10, &tpgt) || tpgt > UINT_MAX)\n
↪"
```

```
        buf += "                return ERR_PTR(-EINVAL);\n\n"
        buf += "        tpg = kzalloc(sizeof(struct " + fabric_mod_name + "_
↪tpg), GFP_KERNEL);\n"
        buf += "        if (!tpg) {\n"
        buf += "                printk(KERN_ERR \"Unable to allocate struct "␣
↪+ fabric_mod_name + "_tpg\");\n"
        buf += "                return ERR_PTR(-ENOMEM);\n"
        buf += "        }\n"
        buf += "        tpg->" + fabric_mod_port + " = " + fabric_mod_port + ";
↪\n"
        buf += "        tpg->" + fabric_mod_port + "_tpgt = tpgt;\n\n"

        if proto_ident == "FC":
                buf += "        ret = core_tpg_register(wwn, &tpg->se_tpg,␣
↪SCSI_PROTOCOL_FCP);\n"
        elif proto_ident == "SAS":
                buf += "        ret = core_tpg_register(wwn, &tpg->se_tpg,␣
↪SCSI_PROTOCOL_SAS);\n"
        elif proto_ident == "iSCSI":
                buf += "        ret = core_tpg_register(wwn, &tpg->se_tpg,␣
↪SCSI_PROTOCOL_ISCSI);\n"

        buf += "        if (ret < 0) {\n"
        buf += "                kfree(tpg);\n"
        buf += "                return NULL;\n"
        buf += "        }\n"
        buf += "        return &tpg->se_tpg;\n"
        buf += "}\n\n"
        buf += "static void " + fabric_mod_name + "_drop_tpg(struct se_portal_
↪group *se_tpg)\n"
        buf += "{\n"
        buf += "        struct " + fabric_mod_name + "_tpg *tpg = container_
↪of(se_tpg,\n"
        buf += "                                    struct " + fabric_mod_name + "_
↪tpg, se_tpg);\n\n"
        buf += "        core_tpg_deregister(se_tpg);\n"
        buf += "        kfree(tpg);\n"
        buf += "}\n\n"

        buf += "static struct se_wwn *" + fabric_mod_name + "_make_" + fabric_
↪mod_port + "(\n"
        buf += "        struct target_fabric_configfs *tf,\n"
        buf += "        struct config_group *group,\n"
        buf += "        const char *name)\n"
        buf += "{\n"
        buf += "        struct " + fabric_mod_name + "_" + fabric_mod_port + "␣
↪*" + fabric_mod_port + ";\n"

        if proto_ident == "FC" or proto_ident == "SAS":
                buf += "        u64 wwpn = 0;\n\n"
```

```
        buf += "        /* if (" + fabric_mod_name + "_parse_wwn(name, &wwpn,␣
↪1) < 0)\n"
        buf += "                return ERR_PTR(-EINVAL); */\n\n"
        buf += "        " + fabric_mod_port + " = kzalloc(sizeof(struct " +␣
↪fabric_mod_name + "_" + fabric_mod_port + "), GFP_KERNEL);\n"
        buf += "        if (!" + fabric_mod_port + ") {\n"
        buf += "                printk(KERN_ERR \"Unable to allocate struct "␣
↪+ fabric_mod_name + "_" + fabric_mod_port + "\");\n"
        buf += "                return ERR_PTR(-ENOMEM);\n"
        buf += "        }\n"

        if proto_ident == "FC" or proto_ident == "SAS":
                buf += "        " + fabric_mod_port + "->" + fabric_mod_port +
↪"_wwpn = wwpn;\n"

        buf += "        /* " + fabric_mod_name + "_format_wwn(&" + fabric_mod_
↪port + "->" + fabric_mod_port + "_name[0], " + fabric_mod_name.upper() + "_
↪NAMELEN, wwpn); */\n\n"
        buf += "        return &" + fabric_mod_port + "->" + fabric_mod_port +
↪"_wwn;\n"
        buf += "}\n\n"
        buf += "static void " + fabric_mod_name + "_drop_" + fabric_mod_port +
↪"(struct se_wwn *wwn)\n"
        buf += "{\n"
        buf += "        struct " + fabric_mod_name + "_" + fabric_mod_port + "␣
↪*" + fabric_mod_port + " = container_of(wwn,\n"
        buf += "                                        struct " + fabric_mod_name + "_
↪" + fabric_mod_port + ", " + fabric_mod_port + "_wwn);\n"
        buf += "        kfree(" + fabric_mod_port + ");\n"
        buf += "}\n\n"

        buf += "static const struct target_core_fabric_ops " + fabric_mod_name␣
↪+ "_ops = {\n"
        buf += "        .module                                 = THIS_MODULE,\n
↪"
        buf += "        .name                                   = \"" + fabric_
↪mod_name + "\",\n"
        buf += "        .get_fabric_name                = " + fabric_mod_name␣
↪+ "_get_fabric_name,\n"
        buf += "        .tpg_get_wwn                            = " + fabric_mod_
↪name + "_get_fabric_wwn,\n"
        buf += "        .tpg_get_tag                            = " + fabric_mod_
↪name + "_get_tag,\n"
        buf += "        .tpg_check_demo_mode                    = " + fabric_mod_
↪name + "_check_false,\n"
        buf += "        .tpg_check_demo_mode_cache        = " + fabric_mod_
↪name + "_check_true,\n"
        buf += "        .tpg_check_demo_mode_write_protect = " + fabric_mod_
↪name + "_check_true,\n"
```

```
        buf += "        .tpg_check_prod_mode_write_protect = " + fabric_mod_
↪name + "_check_false,\n"
        buf += "        .tpg_get_inst_index                = " + fabric_mod_
↪name + "_tpg_get_inst_index,\n"
        buf += "        .release_cmd                       = " + fabric_mod_
↪name + "_release_cmd,\n"
        buf += "        .sess_get_index                    = " + fabric_
↪mod_name + "_sess_get_index,\n"
        buf += "        .sess_get_initiator_sid            = NULL,\n"
        buf += "        .write_pending                     = " + fabric_mod_
↪name + "_write_pending,\n"
        buf += "        .set_default_node_attributes       = " + fabric_mod_
↪name + "_set_default_node_attrs,\n"
        buf += "        .get_cmd_state                     = " + fabric_mod_
↪name + "_get_cmd_state,\n"
        buf += "        .queue_data_in                     = " + fabric_mod_
↪name + "_queue_data_in,\n"
        buf += "        .queue_status                      = " + fabric_mod_
↪name + "_queue_status,\n"
        buf += "        .queue_tm_rsp                      = " + fabric_mod_
↪name + "_queue_tm_rsp,\n"
        buf += "        .aborted_task                      = " + fabric_mod_
↪name + "_aborted_task,\n"
        buf += "        /*\n"
        buf += "         * Setup function pointers for generic logic in target_
↪core_fabric_configfs.c\n"
        buf += "         */\n"
        buf += "        .fabric_make_wwn                = " + fabric_mod_name␣
↪+ "_make_" + fabric_mod_port + ",\n"
        buf += "        .fabric_drop_wwn                = " + fabric_mod_name␣
↪+ "_drop_" + fabric_mod_port + ",\n"
        buf += "        .fabric_make_tpg                = " + fabric_mod_name␣
↪+ "_make_tpg,\n"
        buf += "        .fabric_drop_tpg                = " + fabric_mod_name␣
↪+ "_drop_tpg,\n"
        buf += "};\n\n"

        buf += "static int __init " + fabric_mod_name + "_init(void)\n"
        buf += "{\n"
        buf += "        return target_register_template(&" + fabric_mod_name +
↪"_ops);\n"
        buf += "};\n\n"

        buf += "static void __exit " + fabric_mod_name + "_exit(void)\n"
        buf += "{\n"
        buf += "        target_unregister_template(&" + fabric_mod_name + "_
↪ops);\n"
        buf += "};\n\n"

        buf += "MODULE_DESCRIPTION(\"" + fabric_mod_name.upper() + " series␣
↪fabric driver\");\n"
```

```
        buf += "MODULE_LICENSE(\"GPL\");\n"
        buf += "module_init(" + fabric_mod_name + "_init);\n"
        buf += "module_exit(" + fabric_mod_name + "_exit);\n"

        ret = p.write(buf)
        if ret:
                tcm_mod_err("Unable to write f: " + f)

        p.close()

        return

def tcm_mod_scan_fabric_ops(tcm_dir):

        fabric_ops_api = tcm_dir + "include/target/target_core_fabric.h"

        print "Using tcm_mod_scan_fabric_ops: " + fabric_ops_api
        process_fo = 0;

        p = open(fabric_ops_api, 'r')

        line = p.readline()
        while line:
                if process_fo == 0 and re.search('struct target_core_fabric_
↪ops {', line):
                        line = p.readline()
                        continue

                if process_fo == 0:
                        process_fo = 1;
                        line = p.readline()
                        # Search for function pointer
                        if not re.search('\(\*', line):
                                continue

                        fabric_ops.append(line.rstrip())
                        continue

                line = p.readline()
                # Search for function pointer
                if not re.search('\(\*', line):
                        continue

                fabric_ops.append(line.rstrip())

        p.close()
        return

def tcm_mod_dump_fabric_ops(proto_ident, fabric_mod_dir_var, fabric_mod_name):
        buf = ""
```

```
        bufi = ""

        f = fabric_mod_dir_var + "/" + fabric_mod_name + "_fabric.c"
        print "Writing file: " + f

        p = open(f, 'w')
        if not p:
                tcm_mod_err("Unable to open file: " + f)

        fi = fabric_mod_dir_var + "/" + fabric_mod_name + "_fabric.h"
        print "Writing file: " + fi

        pi = open(fi, 'w')
        if not pi:
                tcm_mod_err("Unable to open file: " + fi)

        buf = "#include <linux/slab.h>\n"
        buf += "#include <linux/kthread.h>\n"
        buf += "#include <linux/types.h>\n"
        buf += "#include <linux/list.h>\n"
        buf += "#include <linux/types.h>\n"
        buf += "#include <linux/string.h>\n"
        buf += "#include <linux/ctype.h>\n"
        buf += "#include <asm/unaligned.h>\n"
        buf += "#include <scsi/scsi_common.h>\n"
        buf += "#include <scsi/scsi_proto.h>\n"
        buf += "#include <target/target_core_base.h>\n"
        buf += "#include <target/target_core_fabric.h>\n"
        buf += "#include \"" + fabric_mod_name + "_base.h\"\n"
        buf += "#include \"" + fabric_mod_name + "_fabric.h\"\n\n"

        buf += "int " + fabric_mod_name + "_check_true(struct se_portal_group␣
↪*se_tpg)\n"
        buf += "{\n"
        buf += "        return 1;\n"
        buf += "}\n\n"
        bufi += "int " + fabric_mod_name + "_check_true(struct se_portal_group␣
↪*);\n"

        buf += "int " + fabric_mod_name + "_check_false(struct se_portal_group␣
↪*se_tpg)\n"
        buf += "{\n"
        buf += "        return 0;\n"
        buf += "}\n\n"
        bufi += "int " + fabric_mod_name + "_check_false(struct se_portal_
↪group *);\n"

        total_fabric_ops = len(fabric_ops)
        i = 0
```

```
        while i < total_fabric_ops:
                fo = fabric_ops[i]
                i += 1
#                print "fabric_ops: " + fo

                if re.search('get_fabric_name', fo):
                        buf += "char *" + fabric_mod_name + "_get_fabric_
↪name(void)\n"
                        buf += "{\n"
                        buf += "        return \"" + fabric_mod_name + "\";\n"
                        buf += "}\n\n"
                        bufi += "char *" + fabric_mod_name + "_get_fabric_
↪name(void);\n"
                        continue

                if re.search('get_wwn', fo):
                        buf += "char *" + fabric_mod_name + "_get_fabric_
↪wwn(struct se_portal_group *se_tpg)\n"
                        buf += "{\n"
                        buf += "        struct " + fabric_mod_name + "_tpg␣
↪*tpg = container_of(se_tpg,\n"
                        buf += "                                    struct " +␣
↪fabric_mod_name + "_tpg, se_tpg);\n"
                        buf += "        struct " + fabric_mod_name + "_" +␣
↪fabric_mod_port + " *" + fabric_mod_port + " = tpg->" + fabric_mod_port + ";\
↪n\n"
                        buf += "        return &" + fabric_mod_port + "->" +␣
↪fabric_mod_port + "_name[0];\n"
                        buf += "}\n\n"
                        bufi += "char *" + fabric_mod_name + "_get_fabric_
↪wwn(struct se_portal_group *);\n"

                if re.search('get_tag', fo):
                        buf += "u16 " + fabric_mod_name + "_get_tag(struct se_
↪portal_group *se_tpg)\n"
                        buf += "{\n"
                        buf += "        struct " + fabric_mod_name + "_tpg␣
↪*tpg = container_of(se_tpg,\n"
                        buf += "                                    struct " +␣
↪fabric_mod_name + "_tpg, se_tpg);\n"
                        buf += "        return tpg->" + fabric_mod_port + "_
↪tpgt;\n"
                        buf += "}\n\n"
                        bufi += "u16 " + fabric_mod_name + "_get_tag(struct se_
↪portal_group *);\n"

                if re.search('tpg_get_inst_index\)\(', fo):
                        buf += "u32 " + fabric_mod_name + "_tpg_get_inst_
↪index(struct se_portal_group *se_tpg)\n"
                        buf += "{\n"
```

```
                        buf += "        return 1;\n"
                        buf += "}\n\n"
                        bufi += "u32 " + fabric_mod_name + "_tpg_get_inst_
↪index(struct se_portal_group *);\n"

                if re.search('\*release_cmd\)\(', fo):
                        buf += "void " + fabric_mod_name + "_release_
↪cmd(struct se_cmd *se_cmd)\n"
                        buf += "{\n"
                        buf += "        return;\n"
                        buf += "}\n\n"
                        bufi += "void " + fabric_mod_name + "_release_
↪cmd(struct se_cmd *);\n"

                if re.search('sess_get_index\)\(', fo):
                        buf += "u32 " + fabric_mod_name + "_sess_get_
↪index(struct se_session *se_sess)\n"
                        buf += "{\n"
                        buf += "        return 0;\n"
                        buf += "}\n\n"
                        bufi += "u32 " + fabric_mod_name + "_sess_get_
↪index(struct se_session *);\n"

                if re.search('write_pending\)\(', fo):
                        buf += "int " + fabric_mod_name + "_write_
↪pending(struct se_cmd *se_cmd)\n"
                        buf += "{\n"
                        buf += "        return 0;\n"
                        buf += "}\n\n"
                        bufi += "int " + fabric_mod_name + "_write_
↪pending(struct se_cmd *);\n"

                if re.search('set_default_node_attributes\)\(', fo):
                        buf += "void " + fabric_mod_name + "_set_default_node_
↪attrs(struct se_node_acl *nacl)\n"
                        buf += "{\n"
                        buf += "        return;\n"
                        buf += "}\n\n"
                        bufi += "void " + fabric_mod_name + "_set_default_node_
↪attrs(struct se_node_acl *);\n"

                if re.search('get_cmd_state\)\(', fo):
                        buf += "int " + fabric_mod_name + "_get_cmd_
↪state(struct se_cmd *se_cmd)\n"
                        buf += "{\n"
                        buf += "        return 0;\n"
                        buf += "}\n\n"
                        bufi += "int " + fabric_mod_name + "_get_cmd_
↪state(struct se_cmd *);\n"
```

```
                if re.search('queue_data_in\)\(', fo):
                        buf += "int " + fabric_mod_name + "_queue_data_
↪in(struct se_cmd *se_cmd)\n"
                        buf += "{\n"
                        buf += "        return 0;\n"
                        buf += "}\n\n"
                        bufi += "int " + fabric_mod_name + "_queue_data_
↪in(struct se_cmd *);\n"

                if re.search('queue_status\)\(', fo):
                        buf += "int " + fabric_mod_name + "_queue_
↪status(struct se_cmd *se_cmd)\n"
                        buf += "{\n"
                        buf += "        return 0;\n"
                        buf += "}\n\n"
                        bufi += "int " + fabric_mod_name + "_queue_
↪status(struct se_cmd *);\n"

                if re.search('queue_tm_rsp\)\(', fo):
                        buf += "void " + fabric_mod_name + "_queue_tm_
↪rsp(struct se_cmd *se_cmd)\n"
                        buf += "{\n"
                        buf += "        return;\n"
                        buf += "}\n\n"
                        bufi += "void " + fabric_mod_name + "_queue_tm_
↪rsp(struct se_cmd *);\n"

                if re.search('aborted_task\)\(', fo):
                        buf += "void " + fabric_mod_name + "_aborted_
↪task(struct se_cmd *se_cmd)\n"
                        buf += "{\n"
                        buf += "        return;\n"
                        buf += "}\n\n"
                        bufi += "void " + fabric_mod_name + "_aborted_
↪task(struct se_cmd *);\n"

        ret = p.write(buf)
        if ret:
                tcm_mod_err("Unable to write f: " + f)

        p.close()

        ret = pi.write(bufi)
        if ret:
                tcm_mod_err("Unable to write fi: " + fi)

        pi.close()
        return

def tcm_mod_build_kbuild(fabric_mod_dir_var, fabric_mod_name):
```

```
        buf = ""
        f = fabric_mod_dir_var + "/Makefile"
        print "Writing file: " + f

        p = open(f, 'w')
        if not p:
                tcm_mod_err("Unable to open file: " + f)

        buf += fabric_mod_name + "-objs                         := " + fabric_
↪mod_name + "_fabric.o \\\n"
        buf += "                                              " + fabric_mod_name␣
↪+ "_configfs.o\n"
        buf += "obj-$(CONFIG_" + fabric_mod_name.upper() +
↪")                 += " + fabric_mod_name + ".o\n"

        ret = p.write(buf)
        if ret:
                tcm_mod_err("Unable to write f: " + f)

        p.close()
        return

def tcm_mod_build_kconfig(fabric_mod_dir_var, fabric_mod_name):

        buf = ""
        f = fabric_mod_dir_var + "/Kconfig"
        print "Writing file: " + f

        p = open(f, 'w')
        if not p:
                tcm_mod_err("Unable to open file: " + f)

        buf = "config " + fabric_mod_name.upper() + "\n"
        buf += "        tristate \"" + fabric_mod_name.upper() + " fabric␣
↪module\"\n"
        buf += "        depends on TARGET_CORE && CONFIGFS_FS\n"
        buf += "        default n\n"
        buf += "        help\n"
        buf += "          Say Y here to enable the " + fabric_mod_name.upper()␣
↪+ " fabric module\n"

        ret = p.write(buf)
        if ret:
                tcm_mod_err("Unable to write f: " + f)

        p.close()
        return

def tcm_mod_add_kbuild(tcm_dir, fabric_mod_name):
```

```
        buf = "obj-$(CONFIG_" + fabric_mod_name.upper() + ")         += " +␣
↪fabric_mod_name.lower() + "/\n"
        kbuild = tcm_dir + "/drivers/target/Makefile"

        f = open(kbuild, 'a')
        f.write(buf)
        f.close()
        return

def tcm_mod_add_kconfig(tcm_dir, fabric_mod_name):
        buf = "source \"drivers/target/" + fabric_mod_name.lower() + "/Kconfig\
↪"\n"
        kconfig = tcm_dir + "/drivers/target/Kconfig"

        f = open(kconfig, 'a')
        f.write(buf)
        f.close()
        return

def main(modname, proto_ident):
#        proto_ident = "FC"
#        proto_ident = "SAS"
#        proto_ident = "iSCSI"

        tcm_dir = os.getcwd();
        tcm_dir += "/../../"
        print "tcm_dir: " + tcm_dir
        fabric_mod_name = modname
        fabric_mod_dir = tcm_dir + "drivers/target/" + fabric_mod_name
        print "Set fabric_mod_name: " + fabric_mod_name
        print "Set fabric_mod_dir: " + fabric_mod_dir
        print "Using proto_ident: " + proto_ident

        if proto_ident != "FC" and proto_ident != "SAS" and proto_ident !=
↪"iSCSI":
                print "Unsupported proto_ident: " + proto_ident
                sys.exit(1)

        ret = tcm_mod_create_module_subdir(fabric_mod_dir)
        if ret:
                print "tcm_mod_create_module_subdir() failed because module␣
↪already exists!"
                sys.exit(1)

        tcm_mod_build_base_includes(proto_ident, fabric_mod_dir, fabric_mod_
↪name)
        tcm_mod_scan_fabric_ops(tcm_dir)
        tcm_mod_dump_fabric_ops(proto_ident, fabric_mod_dir, fabric_mod_name)
        tcm_mod_build_configfs(proto_ident, fabric_mod_dir, fabric_mod_name)
        tcm_mod_build_kbuild(fabric_mod_dir, fabric_mod_name)
```

```
        tcm_mod_build_kconfig(fabric_mod_dir, fabric_mod_name)

        input = raw_input("Would you like to add " + fabric_mod_name + " to␣
↪drivers/target/Makefile..? [yes,no]: ")
        if input == "yes" or input == "y":
                tcm_mod_add_kbuild(tcm_dir, fabric_mod_name)

        input = raw_input("Would you like to add " + fabric_mod_name + " to␣
↪drivers/target/Kconfig..? [yes,no]: ")
        if input == "yes" or input == "y":
                tcm_mod_add_kconfig(tcm_dir, fabric_mod_name)

        return

parser = optparse.OptionParser()
parser.add_option('-m', '--modulename', help='Module name', dest='modname',
                action='store', nargs=1, type='string')
parser.add_option('-p', '--protoident', help='Protocol Ident', dest='protoident
↪',
                action='store', nargs=1, type='string')

(opts, args) = parser.parse_args()

mandatories = ['modname', 'protoident']
for m in mandatories:
        if not opts.__dict__[m]:
                print "mandatory option is missing\n"
                parser.print_help()
                exit(-1)

if __name__ == "__main__":

        main(str(opts.modname), opts.protoident)
```

CHAPTER

FOUR

TARGET EXPORT DEVICE SCRIPT

```
#!/bin/sh
#
# This script illustrates the sequence of operations in configfs to
# create a very simple LIO iSCSI target with a file or block device
# backstore.
#
# (C) Copyright 2014 Christophe Vu-Brugier <cvubrugier@fastmail.fm>
#

print_usage() {
    cat <<EOF
Usage: $(basename $0) [-p PORTAL] DEVICE|FILE
Export a block device or a file as an iSCSI target with a single LUN
EOF
}

die() {
    echo $1
    exit 1
}

while getopts "hp:" arg; do
    case $arg in
        h) print_usage; exit 0;;
        p) PORTAL=${OPTARG};;
    esac
done
shift $(($OPTIND - 1))

DEVICE=$1
[ -n "$DEVICE" ] || die "Missing device or file argument"
[ -b $DEVICE -o -f $DEVICE ] || die "Invalid device or file: ${DEVICE}"
IQN="iqn.2003-01.org.linux-iscsi.$(hostname):$(basename $DEVICE)"
[ -n "$PORTAL" ] || PORTAL="0.0.0.0:3260"

CONFIGFS=/sys/kernel/config
CORE_DIR=$CONFIGFS/target/core
ISCSI_DIR=$CONFIGFS/target/iscsi
```

```
# Load the target modules and mount the config file system
lsmod | grep -q configfs || modprobe configfs
lsmod | grep -q target_core_mod || modprobe target_core_mod
mount | grep -q ^configfs || mount -t configfs none $CONFIGFS
mkdir -p $ISCSI_DIR

# Create a backstore
if [ -b $DEVICE ]; then
    BACKSTORE_DIR=$CORE_DIR/iblock_0/data
    mkdir -p $BACKSTORE_DIR
    echo "udev_path=${DEVICE}" > $BACKSTORE_DIR/control
else
    BACKSTORE_DIR=$CORE_DIR/fileio_0/data
    mkdir -p $BACKSTORE_DIR
    DEVICE_SIZE=$(du -b $DEVICE | cut -f1)
    echo "fd_dev_name=${DEVICE}" > $BACKSTORE_DIR/control
    echo "fd_dev_size=${DEVICE_SIZE}" > $BACKSTORE_DIR/control
    echo 1 > $BACKSTORE_DIR/attrib/emulate_write_cache
fi
echo 1 > $BACKSTORE_DIR/enable

# Create an iSCSI target and a target portal group (TPG)
mkdir $ISCSI_DIR/$IQN
mkdir $ISCSI_DIR/$IQN/tpgt_1/

# Create a LUN
mkdir $ISCSI_DIR/$IQN/tpgt_1/lun/lun_0
ln -s $BACKSTORE_DIR $ISCSI_DIR/$IQN/tpgt_1/lun/lun_0/data
echo 1 > $ISCSI_DIR/$IQN/tpgt_1/enable

# Create a network portal
mkdir $ISCSI_DIR/$IQN/tpgt_1/np/$PORTAL

# Disable authentication
echo 0 > $ISCSI_DIR/$IQN/tpgt_1/attrib/authentication
echo 1 > $ISCSI_DIR/$IQN/tpgt_1/attrib/generate_node_acls

# Allow write access for non authenticated initiators
echo 0 > $ISCSI_DIR/$IQN/tpgt_1/attrib/demo_mode_write_protect

echo "Target ${IQN}, portal ${PORTAL} has been created"
```