Detailed Level Design for Tdinal

Version 4.0 April 18, 2005

February 6, 2008

Tdinal is ksocknal porting to windows nt/xp system. NT kernel has no kernel socket interfaces for network operations. Instead it provides TDI (transport driver interface) to the tdi clients locating at the upper edge of the transport protocol stacks. The tdi inerfaces have different features/protocols to the linux kernel sockets, so we need implement a new tdinal module to fully use the tdi features and make a compact implementation upon tdi and lustre nal without any extra layers (such as socket emulator layer).

1 System Overview:

1.1 Tdinal components:

Windows tidnal module consists of several parts:

- Tdi client objects management
- Tdi connections management
- Tdi data r/t managment
- Common helper routines (buffer /io vector/ MDL/ management ...)
- ksocknal routines

1.1.1 tdi client objects managment routines

Tdi has 3 types of network communication entities, which are managed via FileObjects.

- Transport Address
- Connection endpoints
- Transport provider control channel

We need create the corresponding FileObjects to manage the node and connection.

1.1.2 Tdi connections management

This part handles the connection creation/destruciton and the interactions between ksocknal routines and tdi connecitons. It's a asynchronous process for TDI client to handle incoming connecting requests. And the same to the abnormal disconnection handling.

1.1.3 Tdi data r/t managment

Data process is the core part of tidnal. For receiving, the tdi receive callbacks will be triggerred and we need collect all the data being stored in system Tsdu into our own Tsdu list. Then the ksocknal will get the data from the Tsdu list and release them after using.

For sending, it's a different procedure to receiving. It calls the tdi routine to issue the sending Irp to the underlying transport driver. The latter will do the real data sending. Because tcp transport driver does not support buffer-sending, i.e. the asymchronous mechansisym of ClientEventSendPossible is not implemented. We have to use the asynchronous feature of nt system I/O processing model, because ksocknal must do the data sending/receiving asynchronously.

1.1.4 Common helper routines

ksocknal has I/O vectors and kiov queues to manage the payload. Windows tdi uses the MDL to manage the TSDU.

The main task of this section is to provide the buffer management routines to handle the conversion/transfer between these structures.

1.2 Tdinal sources layout:

There are many routines to achieve the tdinal. The are not arranged exactly following it's componment, because there are other factors which should be considered, such as coding style.

- socknal.h : ksocknal definitions (It's the orignal file of ksocknal module.)
- tdinal.h: the tdinal specified definitions, structures
- daemon.c: the kernel acceptor related routines, including the connections management for a daemon
- debug.c: provides debug helper routines (message printing / status decoding)
- strusup.c: the common helper routines about memory /tdi objects.
- tconn.c: it mainly contains the llifecycle control and data process routines related to tconn
- tcp.c: mainly the tdi event callbacks and tcp related routines

- tdi.c: the tdi objects management routines
- socknal.c: ksocknal files
- socknal_cb.c: ksocknal files

1.3 1.3 Coding/Naming conventions

Here's the ksocknal conventions: (by eeb)

1) use the same typographical and naming conventions (i.e. more like linux; ksocknal_procedure_name(), ksock_struct_type_t, ksnc_structure_member etc), and not mixedCaseNames, CAPITAL_TYPEDEFS, bboolean etc.
2) "typedef" is normally used for two cases:
(a) __u23, __u64 etc.
(b) typedef struct ksock_structure { int ksnc_member_one; int ksnc_member_two; } ksock_structure_t
where (a) is restricted to "well known" types and (b) is ONLY for structures (not scalars) and the type name is the structure name with '_t' added.
3) avoid the define the PXXX types (the pointer defintion). for some cases the '*' in front of a name is needed to tell us it is declaring a pointer.

We should keep an identical coding/naming convention to ksocknal sytel. The codes related to nal part should follow this rule. Windows kernel sources have completely different conventions. Bur for some pure windows tdi routines, I still keep them as windows programming sytle. It saves some time from changing all the codes and also owns respect to the windows one, after all, they are for windows programmers to develop and maintain..

2 Tdinal Implemention

2.1 Tdinal structures

2.1.1 ksock_tconn_t

definition:

```
types of tdi connections:
typedef enum {
    kstt_sender = 0,    // normal sending connection type, it's active
                        // connection, while child tconn is for passive
                        // connection.
    kstt_listener,      // listener daemon type, it just acts as a
                        // daemon, and it does not have real connection.
                        // It manages children tcons to accept or refuse
                        // the connecting request from remote peers.
    kstt_child,         // accepted child connection type, it's parent
```

```
                        // must be Listener
    kstt_lasttype
} ksock_tconn_type;
states of the tdi conneciton:
typedef enum {
    ksts_uninited = 0,  // tconn is just allocated (zero values),
                        // not initialized yet
    ksts_inited,        // tconn structure initialized: so it now
                        // can be identified as a sender, listener
                        // or a child
    ksts_bind,          // tconn is bound: the local address object
                        // (ip/port) is created. after being bound,
                        // we must call ksocknal_put_tconn to release
                        // the tconn objects, it's not safe just to
                        // free the memory of tconn.
    ksts_associated,    // the connection object is created and
                        // associated with the address object. so
                        // it's ready for connection. only for child
                        //  and sender.
    ksts_connecting,    // only used by child tconn: in the ConnectEvent
                        // handler routine, it indicts the child tconn
                        // is busy to be connected to the peer.
    ksts_connected,     // the connection is built already: for
                        // sender and child tconn
    ksts_listening,     // listener daemon is working, only for
                        // listener tconn
    ksts_disconnected,  // disconnected by user
    ksts_aborted,       // un-exptected broken status
    ksts_last           // total number of tconn statuses
} ksock_tconn_state;
Flags of tdi connnections:
#define KS_TCONN_MAGIC              'KSTM'
#define KS_TCONN_HANDLERS_SET 0x00000010  // Conection handlers are set.
#define KS_TCONN_DISCONNECT_BUSY  0x00010000 // Disconnect Workitem
                                             // is queued ...
#define KS_TCONN_DESTROY_BUSY    0x00020000  // Destory Workitem
                                             //is queued ...
struct ksock_tconn {
  unsigned long         kstc_magic;      /* Magic & Flags */
  unsigned long         kstc_flags;
  spinlock_t            kstc_lock;       /* serialise lock*/
  struct ksock_conn *   kstc_conn;       /* ksock_conn_t */
  ksock_tconn_type      kstc_type;       /* tdi connection Type */
  ksock_tconn_state     kstc_state;      /* tdi connection state flag */
  ksock_unicode_name_t  kstc_dev;        /* tcp transport device name */
  ksock_tdi_addr_t      kstc_addr;       /* local address handlers /
```

```
                                        Objects */
    atomic_t              kstc_refcount;   /* reference count of
                                        ksock_tconn */
    struct list_head      kstc_list;       /* linked to global
                                        ksocknal_data */
    union {
      struct {
        ksock_backlogs_t  kstc_listening;  /* listeing backlog child
                                        connections */
        ksock_backlogs_t  kstc_accepted;   /* connected backlog child
                                        connections */
        event_t           kstc_accept_event;   /* Signaled by
            AccepteIrpCompletion, ksocknal_wait_accpeted_conns waits on */
        event_t           kstc_destroy_event;  /* Signaled when accepted
                                            child is released */
      } listener;
      struct  {
        ksock_tconn_info_t  kstc_info;  /* Connection Info if Connected */
        int                 kstc_queued;/* Attached to Parent->ChildList*/
        int                 kstc_queueno; /*0: Attached to Listening list
                                          1: Attached to Accepted list*/
        int                 kstc_busy;     /* referred by
                                        ConnectEventCallback ? */
        int                 kstc_accepted;  /* the connection is ready ?*/
        struct list_head    kstc_link; /*linked to parent tdi connection*/
        ksock_tconn_t   *   kstc_parent;     /* points to it's parent */
        ksock_chain_t       kstc_recv; /* tsdu engine for data receiving*/
      } child;
      struct {
        ksock_tconn_info_t  kstc_info; /* Connection Info if Connected */
        ksock_chain_t       kstc_recv; /* tsdu engine for data receiving*/
      } sender;
    };
    ksock_workitem_t        kstc_destroy; /* tconn destruction workitem */
    ksock_disconnect_workitem_t    kstc_disconnect;  /* connection
                                                disconnect workitem */
};
```

lifecycle & state:

Different type of tdi connection has differnt lifecycle. We need describe them seperately.

sender tconn:

Creation: This type of tconn is created via ksocknal_autoconnect (ksocknal_connect_peer) or ksocknal_cmd (NAL_CMD_CONNECT_PEER / ksocknal_build_conn). And both the two cases will ultimately call the following routines in sequence:

1. ksocknal_create_tconn: allocate the tconn structure from memory and initialize it (set refercount to 1 and no kstc_state set, i.e. it's state is ksts_uninited (0)). It's safe to call ksocknal_free_tconn to free it in failure cases.

2. ksocknal_init_sender: initialize the sender related structures and set kstc_type to kstt_sender / kstc_state to ksts_inited. It's safe to be freed via ksocknal_free_tconn in failure cases.

3. ksocknal_bind_tconn: Here we create the address object to stand for it's local network address entity and set it's state to ksts_bind. After binding, we could not just call ksocknal_free_tconn to free the tconn strcture, becasue it owns tdi address objects. We must close it before freeing or call ksocknal_put_tconn to release it.

4. ksocknal_build_tconn: this routine does several jobs to make the tdi connection ready. a), create a connection FileObject and associate it with the local address FileObject to make the tconn ready to perform the real network conneciton. If it succeeds, we need change the kstc_state to ksts_associated. b), set the corresponding tdi event callbacks (disconnect / data recv & send event callbacks) to response the tdi transport driver's requests. c), issue a TdiConnect Irp to the transport driver to build the connection with the remote peer. If the connection is built successfully, the kstc_state will be changed to ksts_connected, or in failure case we need do some cleanup and abort the tconn, then the tconn will be destroied later via the extra call of ksocknal_put_tconn out side.

5. ksocknal_create_conn: attach the tconn to the newly created ksocknal conn object and schedule data receiving & transmitting on the connection. And inside this routine it will grab another refercount of tconn to keep it alive.

6. ksocknal_put_tconn: drop it's initial reference of the tconn to transfer the ownership to ksocnal conn object. If ksocknal_create_conn fails to grab the refer count (ex: step 4 or 5 get errors), here the tconn will be disconnected (if conected already) and destroied finally. For a connected tconn (sender / child), the destruction is a two phase process. a) it will queue a workitem to abort/disconnect the connection. ksocknal_disconnect_tconn is to be called to do the final disconnection job: issue tdi disconnect irp to the transport driver and disassociate the connecton file object. b) after ksocknal_disconnect_tconn, another ksocknal_put_tconn will be called to performe the second stage of the destruction: disassociate the connection if needed, then release all the fileobjects

of address/connection entities. Since now everything is cleaned up, it's safe to free the memory of the tconn structure via ksocknal_free_tconn.

Destruction: For a sender tconn which has been built ready for ksock_conn_t data processing, there are two cases to abort/disconnect it:

normal case:

ksocknal issues the ksocknal_terminate_conn to terminate the ksocknal connection. In turn a ksocknal_put_tconn will be called and the tconn will destroyed after all the outstanding references drop to zero.

abnormal case:

in case of tdi internal errors, the disconnect event callback will triggerred to shutdown the tdi connection. Here we just abort the tconn and set it's kstc_state to ksts_aborted without dropping the refer count .

Then later the ksocknal data processing routines will get error (because the connection is already aborted.), then it will call ksocknal_terminate_conn to drop the refercount of the tconn, and then the tconn will be destroyed.

We can see that ksocknal_terminate_conn is the ulitimate call to trigger the destruction of sneder / child tconns . And in the end the last caller of ksocknal_put_tconn is to destroy and destroy the whole tconn structure.

listener tconn: listener tconn is a special type of tconn. It could not handle data directly. What it does is only to accept incoming connecting requests and create backlog child tconn to accept the connecting request. The child tconn will perform the data process as sender tconn does.

The listener tconn has two ksock_backlogs_t queues:

```
typedef struct ksock_backlogs {
  struct list_head    list;   /* list to link the backlog connections */
  int                 num;    /* number of backlogs in the list */
} ksock_backlogs_t;
```

1. kstc_listening: contains all the backlog tconns ready to accept the incoming connection requests.

2. kstc_accepted: contains the backlog tconns already connected (accepted) to the peers.

All the children tconns will grab the refercount of the parent/listener tconn, becasue they are sharing the same FileObject of local address, even if the connected tconn are bound to different network address/cards.

Here we must state a rule about the lock acquirision order on the listener and it's children: we just acquire the parent's lock first, then the child tconn. Among children tconns, we must avoid lock acquirision nesting.

The listener tconn involves daemon structure closely. So we describe the daemon and tconn here to clearify the relations of them.

Creation: There's only one path to create listener tconn: ksocknal_cmd / NAL_CMD_START_DAEMON.

This command is executed via ioctl from user mode acceptor program. The acceptor will indict us the ip address and port number to be bound. Then ksocknal_cmd will call ksocknal_start_daemon to start a daemon thread (the listener tconn will be created in the daemon thread):

1. ksocknal_start_daemon: it allocates a new daemon sturcture and execute the ksocknal_daemon in a system thread
2. ksocknal_daemon thread will be starated and in turn it will call ksocknal_create_tconn, ksocknal_init_listener and ksocknal_bind_tconn.
3. ksocknal_create_tconn: just allocate the memory ot tconn structure for the daemon.
4. ksocknal_init_listener: initializes it as a listener: set the type to kstt_listener and state to ksts_inited.
5. ksocknal_bind_tconn: bind it with the user (acceptor) specified ip address / port number
6. ksocknal_start_listener: this routine does several jobs: a) ksocknal_replenish_backlogs to create listening child tdi connections. Every child also will grab a refercount of the parent listener tconn and is to be attached to the daemon tconn's kstc_listening queue. b) set the tdi event callbacks to answer the incoming connection requests from remote peers. c) now the daemon is ready and working , we just set the state to ksts_listening.
7. ksocknal_wait_child_tconn: It waits until the tdi connect event (KsConnectEventHandler) callback approves an incoming connection request with a listening child tconn. Then returns the connected child tconn to ksocknal_daemon thread.
8. ksocknal_daemon will call ksocknal_create_conn to schedule the connected child tconn for data processing by nal and portals.
9. ksocknal_put_tconn: give the handler of the tconn off to the ksocknal conn object.
10. then it will continue the loop from step 7 to wait for next connection request.

Destruction: The destruction of the listener is triggerred by ksocknal_cmd / NAL_CMD_STOP_DAEMON. In response, ksocknal_cmd will call ksocknal_stop_daemon. ksocnal_stop_daemon just sets the shutdown flags and notify the ksocknal_daemon thread, then return back. (It's the ksocknal_daemon's duty to destory the daemon structure and the listener tconn.)

ksocknal_daemon will exit the loop from step 7 to 10. And it will call ksocknal_shut_daemon in the end to release the listener tconn and the daemon structure. The following is the flow of ksocknal_shut_daemon:

1. it will call ksocknal_reset_handlers to NULL all the event handlers. then the transport driver won't bother us any more.

2. cleanup all the listening child tconns in the kstc_listening queue. it calls ksocknal_put_tconn to free the child tconns.

3. then call ksocknal_put_tconn on the listener daemon tconn: daemon->tconn. At the monment, there might be accepted child tconns busy with data traffic (owned by ksock_conn_t objects.) Until all these accepted child tocnns are release, the listener tconn will be destoried automatically when all the references are released. Please refer the "child tconn" section.

4. ksocknal_free_daemon is called to free the memory of the daemon structure.

child tconn: As described in the section of listener tconn, child tconn is created by it's parent tconn. All of them are sharing the same address FileObject of their parent's. The child acts the same to sender tconn on disconnection and data traffic process.

Creation: In the section of ksocknal_start_listener, we can see that in ksocknal_start_listen, the listener daemon will create backlog children tconns in advance. Because the ClientEventConnect callback runs at DISPATCH_LEVEL and could not be blocked. The job of connection object creation and association must be done at IRQL $<$ DISPATCH_LEVEL outside of the event callback routines. In the event handler, we need the pre-prepared tdi connection object to build the ACCEPT Irp and return it to the transport driver to complete building the connection. So we must create the backlog listening child tconns in advance.

The creation of the children tconns are done by ksocknal_replenish_child_tconn. This routine will call ksocknal_create_child_tconn to create the child tconn one by one and attach it to the kstc_listening queue. The following is the flow of ksocknal_create_child_tconn:

1. ksocknal_create_tconn: create the meory structure of tconn

2. ksocknal_init_child: initialize it as a child (set type to kstt_child and set state to ksts_inited)

3. ksocknal_bind_tconn: grab the refercount of it's parent tconn and the address FileObject

4. KsOpenConnection/KsAssociateAddress: create the connection object and associate it with the address object. Then set the state to ksts_associated.

Now the child tconn is ready for the ClientEvnetConnect handler to accept incoming connection request.

Approval for connections: This job is done by the KsConnectEventHandler and ksocknal_wait_child_tconn.

1. KsConnectEventHandler will be called by the transport dirver to response the incoming connection request from remote peers.
2. ksocknal_get_vacancy_backlog: it will search the backlog list of the parent tconn and get a free one. then change it's member kstc_busy to TRUE.
3. Then it will change the tconn's state to ksts_connecting and build Accept Irp upon the vacancy backlog, then return the setuped acception Irp to system.
4. the Irp completion rutine of KsAcceptCompletionRoutine will be triggerred later. If the connection is built successfully, then chage the state of the tconn to ksts_connected, set the child.kstc_accepted to TRUE, and signal the acceptevent. (ksocknal_daemon thread waits on it. It will be waken up.)
5. now it's the job of ksocknal_daemon / ksocknal_wait_child_tconn to find the connected child tconn (kstc_busy is TRUE and kstc_accepted is TRUE) from the kstc_listening list, if find it, then move it to kstc_accepted queue (the child's kstc_queueno will be set to 1) and call ksocknal_create_conn to schedule the data processing.

this part was also described in the previous section of listener tconn.

Destruction: the same in the flow to the destruction of the sender tconns.

concurrency and locks

tconn is the core structure of ksocknal. It will be referred in several contexts, we must use lock to serial the concurrnet access.

1, Irp completion routines
2, tdi event handlers
3, ksocknal threads

2.1.2 daemon

definition

```
typedef struct ksock_daemon {
  ksock_tconn_t *   tconn;    /* the listener connection object */
  unsigned short    nbacklogs;/* number of listening backlog conns */
```

```
  unsigned short    port;     /* listening port number */
  int               shutdown; /* daemon threads is to exit */
  struct list_head  list;     /* to be attached into ksock_nal_data_t*/
} ksock_daemon_t ;
```

state

The logic is described together with the listener tconn, see the section of "listener tconn".

routines

- ksocknal_start_daemon: it allocates a new daemon structure and start the ksocknal_daemon thrad.
- ksocknal_stop_daemon: it will set the shutdown flag of the daemon and active ksocknal_daemon thread
- ksocknal_daemon: it will free the daemon structure before leavingconcurrency and locks

concurrency & locks

For daemon sturcture, there are about 2 possible concurrency races:
1, ksocknal_daemon thread
2, ksocknal_cmd: ksocknal_start/stop_daemon

2.1.3 tsdu

The tsdu is not a queue, but in a buffer format. It's just a block of memory with a header describes some common information, the rest part of the memory block is to store different tsdu slots: KS_TSDU_DAT or KS_TSDU_MDL

definition

```
#define KS_TSDU_MAGIC       'KSTD'
#define KS_TSDU_ATTACHED    0x00000001  // Attached to the socket
                                        // receive tsdu list
typedef struct _KS_TSDU {
    ULONG               Magic;
    ULONG               Flags;
    struct list_head    Link;           // To be linked to KsTsduMgr or
                                        // ksocknal_data.freetsdus
    ULONG               TotalLength;    // Total size of KS_TSDU
    ULONG               StartOffset;    // Start offset of the first Tsdu unit
    ULONG               LastOffset;     // End offset of the last Tsdu unit
/*
    union {
```

```
        KS_TSDU_DAT[];
        KS_TSDU_BUF[];
        KS_TSDU_MDL[];
    }
*/
} KS_TSDU, *PKS_TSDU;
#define TSDU_TYPE_DAT   ((USHORT)0x54000002)
#define TSDU_TYPE_MDL   ((USHORT)0x54000003)
typedef struct _KS_TSDU_DAT {
    USHORT              TsduType;
    USHORT              TsduFlags;
    ULONG               TotalLength;
    ULONG               DataLength;
    ULONG               StartOffset;
    UCHAR               Data[1];
} KS_TSDU_DAT, *PKS_TSDU_DAT;
#define KS_QUADWORD_ALIGN(x)   ((x + 0x0F) & (~(0x0F)))
#define KS_TSDU_STRU_SIZE(Len) (KS_QUADWORD_ALIGN(Len + FIELD_OFFSET(KS_TSDU_DAT, Data)
typedef struct _KS_TSDU_MDL {
    USHORT              TsduType;
    USHORT              TsduFlags;
    ULONG               DataLength;
    ULONG               StartOffset;
    PMDL                Mdl;
    PVOID               Descriptor;
} KS_TSDU_MDL, *PKS_TSDU_MDL;
```

state & logic

Tsdu buffer is only used for data receiving. We need collection all the incoming data into the tsdu list in the tdi receive event handlers, then ksocknal will issue ksocknal_recv_mdl will retrieve the data from the tsdu list and release the corresponding resources to system (transport driver).

First, KsTcpChainedReceiveEventHandler or KsTcpReceiveEventHandler will allocate the tsdu buffer from system memory via KsAllocateKsTsdu, then move data or mdl from system Tsdu to the our own tsdu buffer, and queue the tsdu buffer to the tsdu list manager: TsduMgr.

Later the ksocknal_recv_mdl will lookup the tsdu list and move the data from the Tsdu buffer to portals memory descriptors. Then ksocknal_recv_mdl will put the empty tsdu to a free list (ksocknal_data.freetsdus) for later re-use (KsAllocateKsTsdu will check the freelist first, then try the system memory if the list is empty).

concurrency & lock

There's possible race between ksocknal_recv_mdl and KsTcpXXXReceiveEventHandler routines. We should pay attention in these routines when accessing the tsdu lists.

routines

see the implemention section 2.2

2.1.4 ksock_conn_t

definition

```
typedef struct ksock_conn
{
        ......
        ksock_tconn_t      *ksnc_tconn;         /* tdi internal */
        ......
} ksock_conn_t;
```

state

refer the lustre ksocknal documents.

routines

see section 2.9. There describes the modificaton of the ksocknal routines to work with tdi.

2.1.5 ksock_nal_data_t

This sturcture defines the global information of the ksocknal_data. We add the tdinal related global structures in the tail of this structure.

There are about 3 groups:
1, Tdsu related queues / cache slab/ spinlock
2, tconn related queues / cache slab / spinlock
3, daemon related queues / spinlock

definition:

```
typedef struct
{
        ......
        /*
         *  Tdinal internal defintions
```

```
         */
        TDI_PROVIDER_INFO ksnd_provider;   /* tdi tcp/ip provider's information */
        spinlock_t        ksnd_tconn_lock; /* tdi connections access serialise */
        int               ksnd_ntconns;    /* number of tconns attached in list */
        struct list_head  ksnd_tconns;          /* tdi connections list */
        cfs_mem_cache_t * ksnd_tconn_slab;      /* slabs for ksock_tconn_t allocations
        event_t           ksnd_tconn_exit;      /* exit event to be signaled by the las
        spinlock_t        ksnd_tsdu_lock;       /* tsdu access serialise */

        int               ksnd_ntsdus;          /* number of tsdu buffers allocated */
        unsigned long     ksnd_tsdu_size;       /* the size of a signel tsdu buffer */
        cfs_mem_cache_t * ksnd_tsdu_slab;       /* slab cache for tsdu buffer allocatio
        int               ksnd_nfreetsdus;      /* number of tsdu buffers in the freed
        struct list_head  ksnd_freetsdus;          /* List of the freed Tsdu buffer. */
        spinlock_t        ksnd_daemon_lock;     /* stabilize daemon ops */
        int               ksnd_ndaemons;        /* number of listening daemons */
        struct list_head  ksnd_daemons;         /* listening daemon list */
        event_t           ksnd_daemon_exit;     /* the last daemon quiting should singa
} ksock_nal_data_t;
```

routines: Please see section 2.9 for the logic details of these routines:

- ksocknal_init_tdi_data: initialize these members in ksocknal_data
- ksocknal_fini_tdi_data: finialzie/cleanup these members
- ksocknal_api_startup: it nees call ksocknal_init_tdi_data
- ksocknal_api_shutdown: it needs call ksocknal_fini_tdi_data

2.2 Data process logic

2.2.1 Receiving

The data receiving part is a completely asynchronous process. We need implemented the following 4 receive event calbacks to process the incoming data:

```
+--------------------------------+-----------------------------------+
| Callback                       | Description                       |
+--------------------------------+-----------------------------------+
+--------------------------------+-----------------------------------+
| ReceiveEventHandler            | Normally small buffer transfer    |
+--------------------------------+-----------------------------------+
| ChainedReceiveHandler          | Bulk buffer transfer (via MDL )   |
+--------------------------------+-----------------------------------+
| ReceiveExpeditedHandler        | Out-of-band: normal small buffer  | |
+--------------------------------+------ ----------------------------+
```

```
|ChainedReceiveExpeditedHandler   | Out-of-band: bulk buffer transfer  | |
+-----------------------+-------------------------------------------+
```

in tdinal, they are KsTcpReceiveEventHandler, KsTcpReceiveExpeditedEventHandler, KsTcpChainedReceiveEventHandler, KsTcpChainedReceiveExpeditedEventHandler. The expedited handlers are calling the KsTcpReceiveEventHandler and KsTcpChainedReceiveEventHandler.

Logic of data receiving:

1. tconn is created and initialized, connected. Please refer the section 2.1.1 for the details of this process.

2. then socknal connetion is to be built and initialized via ksocknal_create_conn, the routine will call ksocknal_new_packet to schedule a receive request of portals message header in size_of(ptl_hdr_t) bytes.

3. in case that the remote peer sends data, the tdi callbacks TDI_EVENT_RECEIVE / TDI_EVENT_RECEIVE_EXPEDITED / TDI_EVENT_CHAINED_RECEIVE will be triggerred. Which callback will be triggerred depends on the size of the tsdu data and the options. For a bulked data traffic, the bulk transfer callback TDI_EVENT_CHAINED_RECEIVE / ChainedReceiveHandler will be actived.

4. then the tdi callback (KsTcpReceiveEventHandler / KsTcpChainedReceiveEventHandler) will receive all the incoming data in the system tsdu and queue it into the our own tsdu buffer queue (TsduMgr). It also needs to wake up the scheduler thread via ksocknal_sched_conn(tconn->kstc_conn, xxxx).

5. the scheduler thread now is woken up and processes the receive request of ptl_hdr_t via ksocknal_process_receive

6. ksocknal_process_receive will call ksocknal_receive to read payload into the pre-allocated buffer.

7. similarly, ksocknal_receive calls ksocknal_recv_iov or ksocknal_recv_kiov. And ksocknal_recv_iov or ksocknal_recv_kiov will move the data from tsdu buffers to the the connection data buffer.

8. after the data is received, we are back to ksocknal_process_receive. ksocknal_process_receive is to process the SOCKNAL_RX_HEADER case and call lib_parse with the received ptl_hdr_t.

9. then potoals lib_parse routine will prepare memory buffer and issue a new receive request to read all the payload into the prepared buffer. lib_parse calls ksocknal_recv or ksocknal_recv_pages to schedule the new receive request.

10. then it starts a new loop of these steps, but the state for ksocknal_process_receive is changed to SOCKNAL_RX_BODY now. Steps of 3,4 might be running at the concurrent time while ksocknal_scheduler are busy with data process.

2.2.2 Sending

Since tcp/ip transport does not support buffer-sending, i.e. asynchronous sending, we have to use the windows asynchronous mechanisym to implement the asychronour sending.

Logic of data transmission:

1. portals will call the ksocknal callbacks: ksocknal_send or ksocknal_send_pages to process a sending request.
2. then ksocknal_send or ksocknal_send_pages will call ksocknal_sendmsg.
3. ksocknal_sendmsg will allocate a ltx structure to encapsule all the information inside, then call ksocknal_launch_packet.
4. ksocknal_launch_packet will queue the ltx to the corresponding connection, then wake up the corresponding scheduler thread (ksocknal_queue_tx_locked is to do this job).
5. the scheduler thread will be woken up and call ksocknal_process_transmit.
6. 6. Then ksocknal_transmit will be called by ksocknal_process_transmit.
7. ksocknal_transmit will call ksocknal_send_iov or ksocknal_send_kiov to process the transmission request.
8. ksocknal_send_iov or ksocknal_send_kiov will collect all the buffers and lock them into MDL to make them paged-in, then call ksocknal_send_mdl to send the whole mdl buffer.
9. ksocknal_send_mdl will create the TdiSend Irp and issue it to the transport driver. The squence of IoCallDriver's return and the calling of Irp completion routine (KsTcpCompleitonRoutine) is not uncertain, so we introduce the context->refercount ot detimine whether ksocknal_send_mdl or the completion routine to do the cleanup. The latter will do the cleanup job, such as update the tx iov or kiov offset, wake up the ksocknal_scheduler thread, free the Irp. (in HLD document, there's a detailed explaination on it.)
10. If ksocknal_send_mdl returns STATUS_PENDING, (-EAGAIN) will be returned to ksocknal_scheduler. Then the tx will be re-queued to the conn->ksnc_tx_queue, but conn->ksnc_tx_ready will be kept as 0, so the conn structure won't be queued to sched->kss_tx_conns. It's the duty of the Irp completion routine to re-active the scheduler to process the conn's requests.

11. If step10 succeeds with STATUS_SUCCESS, that means the payload is fullly sent out, now if the caller does not drops the last reference of the completion context, it will do the same to step 10. Otherwise, it will return with (rc $>$ 0). Then ksocknal_transmit will loop until all the payload is sent out, and in trun return to ksocknal_process_transmit. ksocknal_process_transmit will finialize the tx and return to ksocknal_scheduler with (rc $>$ 0).

2.3 Tsdu routines

2.3.1 KsAllocateKsTsdu

Logic:

```
/*
 * KsAllocateKsTsdu
 *   Reuse a Tsdu from the freelist or allocate a new Tsdu
 *   from the LookAsideList table or the NonPagedPool
 *
 * Arguments:
 *   N/A
 *
 * Return Value:
 *   PKS_Tsdu: the new Tsdu or NULL if it fails
 *
 * Notes:
 *   N/A
 */
PKS_TSDU
KsAllocateKsTsdu()
{
    PKS_TSDU    KsTsdu = NULL;
    spin_lock(&(ksocknal_data.ksnd_tsdu_lock));
    if (!list_empty (&(ksocknal_data.ksnd_freetsdus))) {
        LASSERT(ksocknal_data.ksnd_nfreetsdus > 0);
        KsTsdu = list_entry(ksocknal_data.ksnd_freetsdus.next, KS_TSDU, Link);
        list_del(&(KsTsdu->Link));
        ksocknal_data.ksnd_nfreetsdus--;
    } else {
        KsTsdu = (PKS_TSDU) cfs_mem_cache_alloc(
                        ksocknal_data.ksnd_tsdu_slab, 0);
        if (KsTsdu) {
            RtlZeroMemory(KsTsdu, ksocknal_data.ksnd_tsdu_size);
        }
    }
    spin_unlock(&(ksocknal_data.ksnd_tsdu_lock));
    if (NULL != KsTsdu) {
```

```
        KsInitializeKsTsdu(KsTsdu, ksocknal_data.ksnd_tsdu_size);
    }
    return (KsTsdu);
}
```

Use case:

```
Please refer KsTcpReceiveEventHandler:a
        /* allocate the tsdu buffer from NPList or Pool */
        KsTsdu = KsAllocateKsTsdu();
        if (NULL == KsTsdu) {
            goto errorout;
        } else {
            bNewTsdu = TRUE;
        }
```

2.3.2 KsPutKsTsdu

Logic:

```
/*
 * KsPutKsTsdu
 *   Move the Tsdu to the free tsdu list in ksocknal_data.
 *
 * Arguments:
 *   KsTsdu: Tsdu to be moved.
 *
 * Return Value:
 *   N/A
 *
 * Notes:
 *   N/A
 */
VOID
KsPutKsTsdu(
    PKS_TSDU  KsTsdu
    )
{
    spin_lock(&(ksocknal_data.ksnd_tsdu_lock));
    list_add( &(KsTsdu->Link), &(ksocknal_data.ksnd_freetsdus));
    ksocknal_data.ksnd_nfreetsdus++;
    spin_unlock(&(ksocknal_data.ksnd_tsdu_lock));
}
```

Use case:

```
see ksocknal_recv_mdl:
```

```
    /* KsTsdu is empty now, we need put it onto the free list ... */
    list_del(&(KsTsdu->Link));
    KsTsduMgr->NumOfTsdu--;
    KsPutKsTsdu(KsTsdu);
```

2.3.3 KsFreeKsTsdu

Logic:

```
/*
 * KsFreeKsTsdu
 *   Release a Tsdu: uninitialize then free it.
 *
 * Arguments:
 *   KsTsdu: Tsdu to be freed.
 *
 * Return Value:
 *   N/A
 *
 * Notes:
 *   N/A
 */
VOID
KsFreeKsTsdu(
    PKS_TSDU  KsTsdu
    )
{
    spin_lock(&(ksocknal_data.ksnd_tsdu_lock));
    cfs_mem_cache_free(
           ksocknal_data.ksnd_tsdu_slab,
           KsTsdu );

    spin_unlock(&(ksocknal_data.ksnd_tsdu_lock));
}
```

2.3.4 KsInitializeTsdu

Logic:

```
/*
 * KsInitializeKsTsdu
 *   Initialize the Tsdu buffer header
 *
 * Arguments:
 *   KsTsdu: the Tsdu to be initialized
 *   Length: the total length of the Tsdu
 *
```

```
 * Return Value:
 *   VOID
 *
 * NOTES:
 *   N/A
 */
VOID
KsInitializeKsTsdu(
    PKS_TSDU    KsTsdu,
    ULONG       Length
    )
{
    KsTsdu->Magic = KS_TSDU_MAGIC;
    KsTsdu->TotalLength = Length;
    KsTsdu->StartOffset = KsTsdu->LastOffset =
    KS_QUADWORD_ALIGN(sizeof(KS_TSDU));
}
```

Use case:

```
N/A
```

2.3.5 KsInitializeKsTsduMgr

Logic:

```
/*
 * KsInitializeKsTsduMgr
 *   Initialize the management structure of
 *   Tsdu buffers
 *
 * Arguments:
 *   TsduMgr: the TsduMgr to be initialized
 *
 * Return Value:
 *   VOID
 *
 * NOTES:
 *   N/A
 */
VOID
KsInitializeKsTsduMgr(
    PKS_TSDUMGR     TsduMgr
    )
{
    KeInitializeEvent(
            &(TsduMgr->Event),
```

```
            NotificationEvent,
            FALSE
            );
    CFS_INIT_LIST_HEAD(
            &(TsduMgr->TsduList)
            );
    TsduMgr->NumOfTsdu  = 0;
    TsduMgr->TotalBytes = 0;
}
```

Use case:

```
N/A
```

2.3.6 KsInitializeKsChain

Logic:

```
/*
 * KsInitializeKsChain
 *   Initialize the China structure for receiving
 *   or transmitting
 *
 * Arguments:
 *   KsChain: the KsChain to be initialized
 *
 * Return Value:
 *   VOID
 *
 * NOTES:
 *   N/A
 */
VOID
KsInitializeKsChain(
    PKS_CHAIN       KsChain
    )
{
    KsInitializeKsTsduMgr(&(KsChain->Normal));
    KsInitializeKsTsduMgr(&(KsChain->Expedited));
}
```

Use case:

2.3.7 KsCleanupTsduMgr

Logic:

```
/*
```

```
 * KsCleanupTsduMgr
 *   Clean up all the Tsdus in the TsduMgr list
 *
 * Arguments:
 *   KsTsduMgr: the Tsdu list manager
 *
 * Return Value:
 *   NTSTATUS:  nt status code
 *
 * NOTES:
 *   N/A
 */
NTSTATUS
KsCleanupTsduMgr(
    PKS_TSDUMGR     KsTsduMgr
    )
{
    PKS_TSDU         KsTsdu;
    PKS_TSDU_DAT    KsTsduDat;
    PKS_TSDU_MDL    KsTsduMdl;
    LASSERT(NULL != KsTsduMgr);
    KeSetEvent(&(KsTsduMgr->Event), 0, FALSE);
    while (!list_empty(&KsTsduMgr->TsduList)) {
        KsTsdu = list_entry(KsTsduMgr->TsduList.next, KS_TSDU, Link);
        LASSERT(KsTsdu->Magic == KS_TSDU_MAGIC);
        KsTsduDat = (PKS_TSDU_DAT)((PUCHAR)KsTsdu + KsTsdu->StartOffset);
        KsTsduMdl = (PKS_TSDU_MDL)((PUCHAR)KsTsdu + KsTsdu->StartOffset);

        if (TSDU_TYPE_DAT == KsTsduDat->TsduType) {
            KsTsdu->StartOffset += KsTsduDat->TotalLength;
            if (KsTsdu->StartOffset == KsTsdu->LastOffset) {
                //
                // KsTsdu is empty now, we need free it ...
                //
                list_del(&(KsTsdu->Link));
                KsTsduMgr->NumOfTsdu--;
                KsFreeKsTsdu(KsTsdu);
            }
        } else {
            //
            // MDL Tsdu Unit ...
            //
            LASSERT(TSDU_TYPE_MDL == KsTsduMdl->TsduType);
            TdiReturnChainedReceives(
                &(KsTsduMdl->Descriptor),
                1 );
```

```
                KsTsdu->StartOffset += sizeof(KS_TSDU_MDL);
                if (KsTsdu->StartOffset == KsTsdu->LastOffset) {
                    //
                    // KsTsdu is empty now, we need free it ...
                    //
                    list_del(&(KsTsdu->Link));
                    KsTsduMgr->NumOfTsdu--;
                    KsFreeKsTsdu(KsTsdu);
                }
            }
        }
        return STATUS_SUCCESS;
    }
```

Use case:

```
    This routine will be called when disconnecting a tconn to cleanup all the tsdu entries
```

2.3.8 KsCleanupKsChain

Logic:

```
    /*
     * KsCleanupKsChain
     *   Clean up the TsduMgrs of the KsChain
     *
     * Arguments:
     *   KsChain: the chain managing TsduMgr
     *
     * Return Value:
     *   NTSTATUS:  nt status code
     *
     * NOTES:
     *   N/A
     */
    NTSTATUS
    KsCleanupKsChain(
        PKS_CHAIN   KsChain
        )
    {
        NTSTATUS    Status;
        LASSERT(NULL != KsChain);
        Status = KsCleanupTsduMgr(
                    &(KsChain->Normal)
                    );
        if (!NT_SUCCESS(Status)) {
```

```
        cfs_enter_debugger();
        goto errorout;
    }
    Status = KsCleanupTsduMgr(
                &(KsChain->Expedited)
                );
    if (!NT_SUCCESS(Status)) {
        cfs_enter_debugger();
        goto errorout;
   }
errorout:
    return Status;
}
```

Use case:

To be called by KsCleanupTsdu.

2.3.9 KsCleanupTsdu

Logic:

```
/*
 * KsCleanupTsdu
 *   Clean up all the Tsdus of a tdi connected object
 *
 * Arguments:
 *   tconn: the tdi connection which is connected already.
 *
 * Return Value:
 *   Nt status code
 *
 * NOTES:
 *   N/A
 */
NTSTATUS
KsCleanupTsdu(
    ksock_tconn_t * tconn
    )
{
    NTSTATUS        Status = STATUS_SUCCESS;
    if (tconn->kstc_type != kstt_sender &&
        tconn->kstc_type != kstt_child ) {
        goto errorout;
    }
    if (tconn->kstc_type != kstt_sender) {
        Status = KsCleanupKsChain(
```

```
                    &(tconn->sender.kstc_recv)
                    );
            if (!NT_SUCCESS(Status)) {
                cfs_enter_debugger();
                goto errorout;
            }
        } else {
            Status = KsCleanupKsChain(
                    &(tconn->child.kstc_recv)
                    );
            if (!NT_SUCCESS(Status)) {
                cfs_enter_debugger();
                goto errorout;
            }
        }
    errorout:
        return (Status);
    }
```

Use case:

ksocknal_disconnect_tconn will call it to cleanup all the tsdu lists.

```
            /* cleanup the tsdumgr Lists */
            KsCleanupTsdu (tconn);
```

2.4 Mdl routines

2.4.1 KsCopyMdlChainToMdlChain

Logic:

```
    /*
     * KsCopyMdlChainToMdlChain
     *   Copy data from  a [chained] Mdl to anther [chained] Mdl.
     *   Tdi library does not provide this function. We have to
     *   realize it ourselives.
     *
     * Arguments:
     *   SourceMdlChain: the source mdl
     *   SourceOffset:   start offset of the source
     *   DestinationMdlChain: the dst mdl
     *   DestinationOffset: the offset where data are to be copied.
     *   BytesTobecopied:   the expteced bytes to be copied
     *   BytesCopied:    to store the really copied data length
     *
     * Return Value:
```

```
 *   NTSTATUS: STATUS_SUCCESS or other error code
 *
 * NOTES:
 *   The length of source mdl must be >= SourceOffset + BytesTobecopied
 */
NTSTATUS
KsCopyMdlChainToMdlChain(
    IN PMDL     SourceMdlChain,
    IN ULONG    SourceOffset,
    IN PMDL     DestinationMdlChain,
    IN ULONG    DestinationOffset,
    IN ULONG    BytesTobecopied,
    OUT PULONG  BytesCopied
    )
{
    ......
}
```

Use case:

```
see ksocknal_recv_mdl:
//
// Recvmsg requst could be statisfied ...
// We just copy the data from kernel Mdl to
// user specifid Mdl
//
ULONG   BytesCopied = size - BytesRecved;
Status = KsCopyMdlChainToMdlChain(
            KsTsduMdl->Mdl,
            KsTsduMdl->StartOffset,
            mdl,
            BytesRecved,
            BytesCopied,
            &BytesCopied
            );
if (NT_SUCCESS(Status)) {
    ......
}
```

2.4.2 KsQueryMdlSize

Logic:

```
/*
 * KsQueryMdlSize
 *   Query the whole size of a MDL (may be chained)
 *
```

```
 * Arguments:
 *   Mdl:  the Mdl to be queried
 *
 * Return Value:
 *   ULONG: the total size of the mdl
 *
 * NOTES:
 *   N/A
 */
ULONG
KsQueryMdlsSize (PMDL Mdl)
{
    PMDL    Next = Mdl;
    ULONG   Length = 0;
    //
    // Walking the MDL Chain ...
    //
    while (Next) {
        Length += MmGetMdlByteCount(Next);
        Next = Next->Next;
    }
    return (Length);
}
```

Use case:

```
N/A
```

2.4.3 KsLocalUserBuffer

Logic:

```
/*
 * KsLockUserBuffer
 *   Allocate MDL for the buffer and lock the pages into
 *   nonpaged pool
 *
 * Arguments:
 *   UserBuffer:  the user buffer to be locked
 *   Length:      length in bytes of the buffer
 *   Operation:   read or write access
 *   pMdl:        the result of the created mdl
 *
 * Return Value:
 *   NTSTATUS:     kernel status code (STATUS_SUCCESS
 *                 or other error code)
 *
```

```
 * NOTES:
 *   N/A
 */
NTSTATUS
KsLockUserBuffer (
    IN PVOID            UserBuffer,
    IN ULONG            Length,
    IN LOCK_OPERATION   Operation,
    OUT PMDL *          pMdl
    )
{
    NTSTATUS    Status;
    PMDL        Mdl = NULL;

    LASSERT(UserBuffer != NULL);
    *pMdl = NULL;

    Mdl = IoAllocateMdl(
                UserBuffer,
                Length,
                FALSE,
                FALSE,
                NULL
                );

    if (Mdl == NULL) {
        Status = STATUS_INSUFFICIENT_RESOURCES;
    } else {

        __try {
            MmProbeAndLockPages(
                    Mdl,
                    KernelMode,
                    Operation
                    );

            Status = STATUS_SUCCESS;
            *pMdl = Mdl;
        } __except (EXCEPTION_EXECUTE_HANDLER) {
            IoFreeMdl(Mdl);

            Mdl = NULL;
            cfs_enter_debugger();

            Status = STATUS_INVALID_USER_BUFFER;
        }
```

```
    }

    return Status;
}
```

Use case:

```
see it's wrapper function: ksocknal_lock_buffer
```

2.4.4 KsMapMdlBuffer

Logic:

```
/*
 * KsMapMdlBuffer
 *   Map the mdl into a buffer in kernel space
 *
 * Arguments:
 *   Mdl:  the mdl to be mapped
 *
 * Return Value:
 *   PVOID: the buffer mapped or NULL in failure
 *
 * NOTES:
 *   N/A
 */
PVOID
KsMapMdlBuffer (PMDL    Mdl)
{
    LASSERT(Mdl != NULL);

    return MmGetSystemAddressForMdlSafe(
                Mdl,
                NormalPagePriority
                );
}
```

Use case:

```
see it's wrapper function: ksocknal_mal_mdl
```

2.4.5 KsReleaseMdl

Logic:

```
/*
 * KsReleaseMdl
 *   Unlock all the pages in the mdl
```

```
 *
 * Arguments:
 *   Mdl:  memory description list to be released
 *
 * Return Value:
 *   N/A
 *
 * NOTES:
 *   N/A
 */
VOID
KsReleaseMdl (IN PMDL   Mdl)
{
    LASSERT(Mdl != NULL);
    while (Mdl) {
        PMDL    Next;
        Next = Mdl->Next;
        MmUnlockPages(Mdl);
        IoFreeMdl(Mdl);
        Mdl = Next;
    }
}
```

Use case:

```
see it's wrapper function: ksocknal_release_mdl
```

2.4.6 ksocknal_lock_buffer

Logic:

```
/*
 * ksocknal_lock_buffer
 *   allocate MDL for the user spepcified buffer and lock (paging-in)
 *   all the pages of the buffer into system memory
 *
 * Arguments:
 *   buffer:  the user buffer to be locked
 *   length:  length in bytes of the buffer
 *   access:  read or write access
 *   mdl:     the result of the created mdl
 *
 * Return Value:
 *   int:     the ksocknal error code: 0: success / -x: failture
 *
 * Notes:
 *   N/A
```

```
 */
int
ksocknal_lock_buffer (
    void *            buffer,
    int               length,
    LOCK_OPERATION    access,
    ksock_mdl_t **    kmdl
    )
{
    NTSTATUS        status;
    status = KsLockUserBuffer(
                    buffer,
                    length,
                    access,
                    kmdl
                    );
    return cfs_error_code(status);
}
```

Use case:

```
see ksocknal_lock_iovs and ksocknal_lock_kiovs
   ......
    rc = ksocknal_lock_buffer(
                iov[i].iov_base,
                iov[i].iov_len,
                IoReadAccess | IoWriteAccess,
                &Iovec );
    if (rc < 0)
            break;
    }
    ......
```

2.4.7 ksocknal_map_mdl

Logic:

```
/*
 * ksocknal_map_mdl
 *   Map the mdl pages into kernel space
 *
 * Arguments:
 *   mdl:  the mdl to be mapped
 *
 * Return Value:
 *   void *: the buffer mapped or NULL in failure
 *
```

```
 * Notes:
 *   N/A
 */
void *
ksocknal_map_mdl (ksock_mdl_t * mdl)
{
    LASSERT(mdl != NULL);
    return KsMapMdlBuffer(mdl);
}
```

Use case:

```
N/A
```

2.4.8 ksocknal_release_mdl

Logic:

```
/*
 *  ksocknal_release_mdl
 *   Unlock all the pages in the mdl and release the mdl
 *
 * Arguments:
 *   mdl:  memory description list to be released
 *
 * Return Value:
 *   N/A
 *
 * Notes:
 *   N/A
 */
void
ksocknal_release_mdl (ksock_mdl_t *mdl)
{
    LASSERT(mdl != NULL);
    KsReleaseMdl(mdl);
}
```

Use case:

```
see ksocknal_recv_kiov
    ......
    /* release the chained mdl */
    ksocknal_release_mdl(mdl);
    ......
```

2.4.9 ksocknal_lock_iovs

Logic:

```
/*
 * ksocknal_lock_iovs
 *   Lock the i/o vector buffers into MDL structure
 *
 * Arguments:
 *   iov:  the array of i/o vectors
 *   niov: number of i/o vectors to be locked
 *   len:  the real length of the iov vectors
 *
 * Return Value:
 *   ksock_mdl_t *: the Mdl of the locked buffers or
 *         NULL pointer in failure case
 *
 * Notes:
 *   N/A
 */
ksock_mdl_t *
ksocknal_lock_iovs(
    IN struct iovec  *iov,
    IN int            niov,
    IN int *          len )
{
    int             rc;
    int             i = 0;
    int             total = 0;
    ksock_mdl_t *   mdl = NULL;
    ksock_mdl_t *   tail = NULL;
    LASSERT(iov != NULL);
    LASSERT(niov > 0);
    LASSERT(len != NULL);
    for (i=0; i < niov; i++) {
        ksock_mdl_t * Iovec = NULL;

        rc = ksocknal_lock_buffer(
                iov[i].iov_base,
                iov[i].iov_len,
                IoReadAccess | IoWriteAccess,
                &Iovec );
        if (rc < 0)
            break;
        }
        if (tail) {
```

```
            tail->Next = Iovec;
        } else {
            mdl = Iovec;
        }
        tail = Iovec;
        total +=iov[i].iov_len;
    }
    if (rc >= 0) {
        *len = total;
    } else {
        if (mdl) {
            ksock_release_mdl(mdl);
            mdl = NULL;
        }
    }
    return mdl;
}
```

Use case:

```
see ksocknal_recv_iov:
    /* lock the whole tx iovs into a single mdl chain */
    mdl = ksocknal_lock_iovs(iov, conn->ksnc_rx_niov, &size);
    if (!mdl) {
        rc = -ENOMEM;
        return (rc);
    }
```

2.4.10 ksocknal_lock_kiovs

Logic:

```
/*
 * ksocknal_lock_kiovs
 *   Lock the kiov pages into MDL structure
 *
 * Arguments:
 *   kiov:  the array of kiov pages
 *   niov:  number of kiov to be locked
 *   len:   the real length of the kiov arrary
 *
 * Return Value:
 *   PMDL: the Mdl of the locked buffers or NULL
 *         pointer in failure case
 *
 * Notes:
 *   N/A
```

```
 */
ksock_mdl_t *
ksocknal_lock_kiovs(
    IN ptl_kiov_t *   kiov,
    IN int            nkiov,
    IN int *          len )
{
    int             rc;
    int             i = 0;
    int             total = 0;
    ksock_mdl_t *   mdl = NULL;
    ksock_mdl_t *   tail = NULL;
    LASSERT(kiov != NULL);
    LASSERT(nkiov > 0);
    LASSERT(len != NULL);
    for (i=0; i < niov; i++) {
        ksock_mdl_t *        Iovec = NULL;
        //
        //  Lock the kiov page into Iovec ¡
        //
        rc = ksocknal_lock_buffer(
                (PUCHAR)kiov[i].kiov_page->addr +
                     kiov[i].kiov_offset,
                iov[i].kiov_len,
                IoReadAccess | IoWriteAccess,
                &Iovec
            );
        if (rc < 0) {
            break;
        }
        //
        // Attach the Iovec to the mdl chain
        //
        if (tail) {
            tail->Next = Iovec;
        } else {
            mdl = Iovec;
        }
        tail = Iovec;
        total += kiov[i].kiov_len;
    }
    if (rc >= 0) {
        *len = total;
    } else {
        if (mdl) {
            ksocknal_release_mdl(mdl);
```

```
            mdl = NULL;
        }
    }
    return mdl;
}
```

Use case:

```
see ksocknal_recv_kiov:
    /* lock the whole tx kiovs into a single mdl chain */
    mdl = ksocknal_lock_kiovs(kiov, conn->ksnc_rx_nkiov, &size);
    if (!mdl) {
            rc = -ENMEM;
            return (rc);
    }
    .....
```

2.5 Irp manipulation routines

2.5.1 KsBuildTdiIrp

Logic:

```
/*
 * KsBuildTdiIrp
 *   Allocate a new IRP and initialize it to be issued to tdi
 *
 * Arguments:
 *   DeviceObject:  device object created by the underlying
 *                  TDI transport driver
 *
 * Return Value:
 *   PRIP:   the allocated Irp in success or NULL in failure.
 *
 * NOTES:
 *   N/A
 */
PIRP
KsBuildTdiIrp(
    IN PDEVICE_OBJECT    DeviceObject
    )
{
    PIRP                Irp;
    PIO_STACK_LOCATION  IrpSp;
    //
    // Allocating the IRP ...
    //
```

```
    Irp = IoAllocateIrp(DeviceObject->StackSize, FALSE);
    if (NULL != Irp) {
        //
        // Getting the Next Stack Location ...
        //
        IrpSp = IoGetNextIrpStackLocation(Irp);
        //
        // Initializing Irp ...
        //
        IrpSp->MajorFunction = IRP_MJ_INTERNAL_DEVICE_CONTROL;
        IrpSp->Parameters.DeviceIoControl.IoControlCode = 0;
    }
    return Irp;
}
```

Use case:

see KsSetEventHandlers:

```
        PIRP            Irp;
        //
        // Building Tdi Internal Irp ...
        //
        Irp = KsBuildTdiIrp(DeviceObject);
        if (NULL == Irp) {
            Status = STATUS_INSUFFICIENT_RESOURCES;
        } else {
            ......
        }
```

2.5.2 KsSubmitTdiIrp

Logic:

```
/*
 * KsSubmitTdiIrp
 *   Issue the Irp to the underlying tdi driver
 *
 * Arguments:
 *   DeviceObject:  the device object created by TDI driver
 *   Irp:           the I/O request packet to be processed
 *   bSynchronous:  synchronous or not. If true, we need wait
 *                  until the process is finished.
 *   Information:   returned info
 *
 * Return Value:
 *   NTSTATUS:      kernel status code
```

```
 *
 * NOTES:
 *   N/A
 */
NTSTATUS
KsSubmitTdiIrp(
    IN PDEVICE_OBJECT   DeviceObject,
    IN PIRP             Irp,
    IN BOOLEAN          bSynchronous,
    OUT PULONG          Information
    )
{
    NTSTATUS            Status;
    KEVENT              Event;
    if (bSynchronous) {
        KeInitializeEvent(
            &Event,
            SynchronizationEvent,
            FALSE
            );
        IoSetCompletionRoutine(
            Irp,
            KsIrpCompletionRoutine,
            &Event,
            TRUE,
            TRUE,
            TRUE
            );
    }
    Status = IoCallDriver(DeviceObject, Irp);
    if (bSynchronous) {
        if (STATUS_PENDING == Status) {
            Status = KeWaitForSingleObject(
                        &Event,
                        Executive,
                        KernelMode,
                        FALSE,
                        NULL
                        );
        }
        Status = Irp->IoStatus.Status;
        if (Information) {
            *Information = Irp->IoStatus.Information;
        }
        IoFreeIrp(Irp);
    }
```

```
    if (!NT_SUCCESS(Status)) {
        KsPrint((2, "KsSubmitTdiIrp: Error when submitting the Irp: Status = %xh (%s) .
                    Status, KsNtStatusToString(Status)));
    }
    return (Status);
}
Use case:
    see KsQueryConnectionInfo:
        ......
        Status = KsSubmitTdiIrp(
                    DeviceObject,
                    Irp,
                    TRUE,
                    ConnectionSize
                    );
        ......
```

2.6 Tdi client objects

The routines of this section are to manipulate the tdi objects: creation/destrution, set/query attributes, etc. The logics of these functions are decided and described by NTDDK. There's little things flexible.

2.6.1 KsOpenControl

Logic description:

```
/*
* KsOpenControl
* Open the Control Channel Object ...
*
* Arguments:
*   DeviceName: the transport device name to be opened * Handle: to contain the opened
*   FileObject: to contain the FileObject of the device
*
* Return Value:
*   NTSTATUS: kernel status code (STATUS_SUCCESS or other error code)
*
* Notes:
*   N/A
*/
NTSTATUS
KsOpenControl(
    IN PUNICODE_STRING      DeviceName,
    OUT HANDLE *            Handle,
    OUT PFILE_OBJECT *      FileObject
```

```
    )
{
    NTSTATUS          Status = STATUS_SUCCESS;
    OBJECT_ATTRIBUTES ObjectAttributes;
    IO_STATUS_BLOCK   IoStatus;
    LASSERT( KeGetCurrentIrql() < DISPATCH_LEVEL );
    //
    // Initializing ...
    //
    InitializeObjectAttributes(
        &ObjectAttributes,
        DeviceName,
        OBJ_CASE_INSENSITIVE,
        NULL,
        NULL
        );
    LASSERT( KeGetCurrentIrql() < DISPATCH_LEVEL );
    //
    // Creating the Transport Address Object ...
    //
    Status = ZwCreateFile(
                Handle,
                GENERIC_READ | GENERIC_WRITE,
                &ObjectAttributes,
                &IoStatus,
                0,
                FILE_ATTRIBUTE_NORMAL,
                FILE_SHARE_READ,
                FILE_OPEN_IF,
                0,
                NULL,
                0
                );
    if (NT_SUCCESS(Status)) {
        //
        // Now Obtaining the FileObject of the Transport Address ...
        //
        Status = ObReferenceObjectByHandle(
                    *Handle,
                    FILE_ANY_ACCESS,
                    NULL,
                    KernelMode,
                    FileObject,
                    NULL
                    );
        if (!NT_SUCCESS(Status)) {
```

```
            cfs_enter_debugger();
            ZwClose(*Handle);
        }
    } else {
        cfs_enter_debugger();
    }
    return (Status);
}
```

Use case:

```
Please refer function: KsQueryProviderInfo:
......
RtlInitUnicodeString(&ControlName, TdiDeviceName);
    //
    // Open the Tdi Control Channel
    //
    Status = KsOpenControl(
                &ControlName,
                &Handle,
                &FileObject
                );
    if (!NT_SUCCESS(Status)) {
        KsPrint((2, "KsQueryProviderInfo: Fail to open the tdi control channel.\n"));
        return (Status);
   ......
```

2.6.2 KsCloseControl

Logic description:

```
/*
 * KsCloseControl
 *   Release the Control Channel Handle and FileObject
 *
 * Arguments:
 *   Handle:       the channel handle to be released
 *   FileObject:   the fileobject to be released
 *
 * Return Value:
 *   NTSTATUS:     kernel status code (STATUS_SUCCESS
 *                 or other error code)
 *
 * Notes:
 *   N/A
 */
NTSTATUS
```

```
KsCloseControl(
    IN HANDLE             Handle,
    IN PFILE_OBJECT       FileObject
   )
{
    NTSTATUS  Status = STATUS_SUCCESS;
    LASSERT( KeGetCurrentIrql() < DISPATCH_LEVEL );
    if (FileObject) {
        ObDereferenceObject(FileObject);
    }
    if (Handle) {
        Status = ZwClose(Handle);
    }
    ASSERT(NT_SUCCESS(Status));
    return (Status);
}
```

Use case:

```
Refer KsQueryProviderInfo:
......
KsCloseControl(Handle, FileObject);
......
```

2.6.3 KsOpenAddress

Logic description:

```
/*
 * KsOpenAddress
 *   Open the tdi address object
 *
 * Arguments:
 *   DeviceName:   device name of the address object
 *   pAddress:     tdi address of the address object
 *   AddressLength: length in bytes of the tdi address
 *   Handle:       the newly opened handle
 *   FileObject:   the newly opened fileobject
 *
 * Return Value:
 *   NTSTATUS:     kernel status code (STATUS_SUCCESS
 *                 or other error code)
 *
 * Notes:
 *   N/A
 */
NTSTATUS
```

```
KsOpenAddress(
    IN PUNICODE_STRING      DeviceName,
    IN PTRANSPORT_ADDRESS   pAddress,
    IN ULONG                AddressLength,
    OUT HANDLE *            Handle,
    OUT PFILE_OBJECT *      FileObject
   )
{
    NTSTATUS          Status = STATUS_SUCCESS;
    PFILE_FULL_EA_INFORMATION Ea = NULL;
    ULONG             EaLength;
    UCHAR             EaBuffer[EA_MAX_LENGTH];
    OBJECT_ATTRIBUTES ObjectAttributes;
    IO_STATUS_BLOCK   IoStatus;
    //
    // Building EA for the Address Object to be Opened ...
    //
    Ea = (PFILE_FULL_EA_INFORMATION)EaBuffer;
    Ea->NextEntryOffset = 0;
    Ea->Flags = 0;
    Ea->EaNameLength = TDI_TRANSPORT_ADDRESS_LENGTH;
    Ea->EaValueLength = (USHORT)AddressLength;
    RtlCopyMemory(
        &(Ea->EaName),
        TdiTransportAddress,
        Ea->EaNameLength + 1
        );
    RtlMoveMemory(
        &(Ea->EaName[Ea->EaNameLength + 1]),
        pAddress,
        AddressLength
        );
    EaLength =  sizeof(FILE_FULL_EA_INFORMATION) +
                Ea->EaNameLength + AddressLength;
    LASSERT( KeGetCurrentIrql() < DISPATCH_LEVEL );
    //
    // Initializing ...
    //
    InitializeObjectAttributes(
        &ObjectAttributes,
        DeviceName,
        OBJ_CASE_INSENSITIVE,
        NULL,
        NULL
        );
    LASSERT( KeGetCurrentIrql() < DISPATCH_LEVEL );
```

```
    //
    // Creating the Transport Address Object ...
    //
    Status = ZwCreateFile(
                Handle,
                GENERIC_READ | GENERIC_WRITE | SYNCHRONIZE,
                &ObjectAttributes,
                &IoStatus,
                0,
                FILE_ATTRIBUTE_NORMAL,
                FILE_SHARE_READ,
                FILE_OPEN_IF,
                0,
                Ea,
                EaLength
                );
    if (NT_SUCCESS(Status)) {
        //
        // Now Obtaining the FileObject of the Transport Address ...
        //
        Status = ObReferenceObjectByHandle(
                    *Handle,
                    FILE_ANY_ACCESS,
                    NULL,
                    KernelMode,
                    FileObject,
                    NULL
                    );
        if (!NT_SUCCESS(Status)) {
            cfs_enter_debugger();
            ZwClose(*Handle);
        }
    } else {
        cfs_enter_debugger();
    }
    return (Status);
}
```

Use case:

```
Refer ksocknal_bind_tconn:
......
status = KsOpenAddress(
                    &(tconn->kstc_dev),
                    &(taddr.Tdi),
                    AddrLength,
```

```
                    &(KsAddress.Handle),
                    &(KsAddress.FileObject)
                    );
if (!NT_SUCCESS(Status)) {
    rc = cfs_error_code(status);
    goto errorout;
}
```

2.6.4 KsCloseAddress

Logic description:

```
/*
 * KsCloseAddress
 *   Release the Hanlde and FileObject of an opened tdi
 *   address object
 *
 * Arguments:
 *   Handle:       the handle to be released
 *   FileObject:   the fileobject to be released
 *
 * Return Value:
 *   NTSTATUS:     kernel status code (STATUS_SUCCESS
 *                 or other error code)
 *
 * Notes:
 *   N/A
 */
NTSTATUS
KsCloseAddress(
    IN HANDLE             Handle,
    IN PFILE_OBJECT       FileObject
)
{
    NTSTATUS  Status = STATUS_SUCCESS;
    LASSERT( KeGetCurrentIrql() < DISPATCH_LEVEL );
    if (FileObject) {
        ObDereferenceObject(FileObject);
    }
    if (Handle) {
        Status = ZwClose(Handle);
    }
    ASSERT(NT_SUCCESS(Status));
    return (Status);
}
```

Use case:

```
Please refer ksocknal_destroy_tconn:
KsCloseAddress(
            tconn->kstc_addr.Handle,
            tconn->kstc_addr.FileObject
            );
```

2.6.5 KsOpenConnection

Logic description:

```
/*
 * KsOpenConnection
 *   Open a tdi connection object
 *
 * Arguments:
 *   DeviceName:   device name of the connection object
 *   ConnectionContext: the connection context
 *   Handle:       the newly opened handle
 *   FileObject:   the newly opened fileobject
 *
 * Return Value:
 *   NTSTATUS:     kernel status code (STATUS_SUCCESS
 *                 or other error code)
 *
 * Notes:
 *   N/A
 */
NTSTATUS
KsOpenConnection(
    IN PUNICODE_STRING      DeviceName,
    IN PVOID                ConnectionContext,
    OUT HANDLE *            Handle,
    OUT PFILE_OBJECT *      FileObject
   )
{
    NTSTATUS            Status = STATUS_SUCCESS;
    PFILE_FULL_EA_INFORMATION Ea = NULL;
    ULONG               EaLength;
    UCHAR               EaBuffer[EA_MAX_LENGTH];
    OBJECT_ATTRIBUTES   ObjectAttributes;
    IO_STATUS_BLOCK     IoStatus;
    //
    // Building EA for the Address Object to be Opened ...
    //
    Ea = (PFILE_FULL_EA_INFORMATION)EaBuffer;
```

```
Ea->NextEntryOffset = 0;
Ea->Flags = 0;
Ea->EaNameLength = TDI_CONNECTION_CONTEXT_LENGTH;
Ea->EaValueLength = (USHORT)sizeof(PVOID);
RtlCopyMemory(
    &(Ea->EaName),
    TdiConnectionContext,
    Ea->EaNameLength + 1
    );
RtlMoveMemory(
    &(Ea->EaName[Ea->EaNameLength + 1]),
    &(ConnectionContext),
    sizeof(PVOID)
    );
EaLength =  sizeof(FILE_FULL_EA_INFORMATION) - 1 +
                              Ea->EaNameLength + 1 + sizeof(PVOID);
LASSERT( KeGetCurrentIrql() < DISPATCH_LEVEL );
//
// Initializing ...
//
InitializeObjectAttributes(
    &ObjectAttributes,
    DeviceName,
    OBJ_CASE_INSENSITIVE,
    NULL,
    NULL
    );
LASSERT( KeGetCurrentIrql() < DISPATCH_LEVEL );
//
// Creating the Connection Object ...
//
Status = ZwCreateFile(
            Handle,
            GENERIC_READ | GENERIC_WRITE | SYNCHRONIZE,
            &ObjectAttributes,
            &IoStatus,
            0,
            FILE_ATTRIBUTE_NORMAL,
            FILE_SHARE_READ,
            FILE_OPEN_IF,
            0,
            Ea,
            EaLength
            );
if (NT_SUCCESS(Status)) {
    //
```

```
            // Now Obtaining the FileObject of the Transport Address ...
            //
            Status = ObReferenceObjectByHandle(
                        *Handle,
                        FILE_ANY_ACCESS,
                        NULL,
                        KernelMode,
                        FileObject,
                        NULL
                        );
            if (!NT_SUCCESS(Status)) {
                cfs_enter_debugger();
                ZwClose(*Handle);
            }
        } else {
            cfs_enter_debugger();
        }
        return (Status);
    }
```

Use case:

```
    Please refer ksocknal_build_tconn:
    /* create the connection file handle / object  */
    status =  KsOpenConnection(
                    &(tconn->kstc_dev),
                    (PVOID)tconn,
                    &(tconn->sender.kstc_conn.Handle),
                    &(tconn->sender.kstc_conn.FileObject)
                    );
```

2.6.6 KsCloseConnection

Logic:

```
    /*
     * KsCloseConnection
     *   Release the Hanlde and FileObject of an opened tdi
     *   connection object
     *
     * Arguments:
     *   Handle:       the handle to be released
     *   FileObject:   the fileobject to be released
     *
     * Return Value:
```

```
 *   NTSTATUS:     kernel status code (STATUS_SUCCESS
 *                 or other error code)
 *
 * Notes:
 *   N/A
 */
NTSTATUS
KsCloseConnection(
    IN HANDLE             Handle,
    IN PFILE_OBJECT       FileObject
    )
{
    NTSTATUS  Status = STATUS_SUCCESS;
    LASSERT( KeGetCurrentIrql() < DISPATCH_LEVEL );
    if (FileObject) {
        ObDereferenceObject(FileObject);
    }
    if (Handle) {
        Status = ZwClose(Handle);
    }
    ASSERT(NT_SUCCESS(Status));
    return (Status);
}
```

Use case:

```
Plese refer ksocknal_destroy_tconn
    /* release the connection object */
    KsCloseConnection(
            tconn->child.kstc_info.Handle,
            tconn->child.kstc_info.FileObject
           );
```

2.6.7 KsAssociateAddress

Logic:

```
/*
 * KsAssociateAddress
 *   Associate an address object with a connection object
 *
 * Arguments:
 *   AddressHandle:  the handle of the address object
 *   ConnectionObject:  the FileObject of the connection
 *
 * Return Value:
 *   NTSTATUS:     kernel status code (STATUS_SUCCESS
```

```
 *                 or other error code)
 *
 * Notes:
 *   N/A
 */
NTSTATUS
KsAssociateAddress(
    IN HANDLE           AddressHandle,
    IN PFILE_OBJECT     ConnectionObject
    )
{
    NTSTATUS            Status;
    PDEVICE_OBJECT      DeviceObject;
    PIRP                Irp;
    //
    // Getting the DeviceObject from Connection FileObject
    //
    DeviceObject = IoGetRelatedDeviceObject(ConnectionObject);
    //
    // Building Tdi Internal Irp ...
    //
    Irp = KsBuildTdiIrp(DeviceObject);
    if (NULL == Irp) {
        Status = STATUS_INSUFFICIENT_RESOURCES;
    } else {
        //
        // Assocating the Address Object with the Connection Object
        //
        TdiBuildAssociateAddress(
            Irp,
            DeviceObject,
            ConnectionObject,
            NULL,
            NULL,
            AddressHandle
            );
        //
        // Calling the Transprot Driver with the Prepared Irp
        //
        Status = KsSubmitTdiIrp(DeviceObject, Irp, TRUE, NULL);
    }

    return (Status);
}
```

Use case:

```
Please refer ksocknal_build_tconn:
/* associdate the the connection with the adress object of the tconn */
status = KsAssociateAddress(
            tconn->kstc_addr.Handle,
            tcon->sender.kstc_conn.FileObject
            );
```

2.6.8 KsDisassociateAddress

Logic:

```
/*
 * KsDisassociateAddress
 *   Disassociate the connection object (the relationship will
 *   the corresponding address object will be dismissed. )
 *
 * Arguments:
 *   ConnectionObject:  the FileObject of the connection
 *
 * Return Value:
 *   NTSTATUS:     kernel status code (STATUS_SUCCESS
 *                 or other error code)
 *
 * Notes:
 *   N/A
 */
NTSTATUS
KsDisassociateAddress(
    IN PFILE_OBJECT     ConnectionObject
    )
{
    NTSTATUS            Status;
    PDEVICE_OBJECT      DeviceObject;
    PIRP                   Irp;
    //
    // Getting the DeviceObject from Connection FileObject
    //
    DeviceObject = IoGetRelatedDeviceObject(ConnectionObject);
    //
    // Building Tdi Internal Irp ...
    //
    Irp = KsBuildTdiIrp(DeviceObject);
    if (NULL == Irp) {
        Status = STATUS_INSUFFICIENT_RESOURCES;
    } else {
```

```
        //
        // Disassocating the Address Object with the Connection Object
        //
        TdiBuildDisassociateAddress(
            Irp,
            DeviceObject,
            ConnectionObject,
            NULL,
            NULL
            );
        //
        // Calling the Transprot Driver with the Prepared Irp
        //
        Status = KsSubmitTdiIrp(DeviceObject, Irp, TRUE, NULL);
    }
    return (Status);
}
```

Use case:

```
Please refer ksocknal_destroy_tconn:
    /* disassociate the relation between it's connection object
       and the address object */
    if (tconn->kstc_state == ksts_associated) {
        KsDisassociateAddress(
                tconn->child.kstc_info.FileObject
                );
    }
    ......
```

2.6.9 KsSetEventHandlers

Logic:

```
/*
//
// Connection Control Event Callbacks
//
TDI_EVENT_CONNECT
TDI_EVENT_DISCONNECT
TDI_EVENT_ERROR
//
// Tcp Event Callbacks
//
TDI_EVENT_RECEIVE
TDI_EVENT_RECEIVE_EXPEDITED
TDI_EVENT_CHAINED_RECEIVE
```

```
TDI_EVENT_CHAINED_RECEIVE_EXPEDITED
//
// Udp Event Callbacks
//
TDI_EVENT_RECEIVE_DATAGRAM
TDI_EVENT_CHAINED_RECEIVE_DATAGRAM
*/
/*
 * KsSetEventHandlers
 *   Set the tdi event callbacks with an address object
 *
 * Arguments:
 *   AddressObject: the FileObject of the address object
 *   EventContext:  the parameter for the callbacks
 *   Flags: indicts which callback is to be set
 *   ConnectEventHandler: to be called when an connection
 *                        is offered by a remote-node peer
 *   DisconnectEventHandler: connection break event callback
 *   ErrorEventHandler: error process callback
 *   ErrorExEventHandler: ...
 *   SendPossibleHandler: Kernel socket spack is available
 *   ReceiveEventHandler: callback for normal receives
 *   ReceiveExpeditedEventHandler: for expedited (OOB) recvs
 *   ChainedReceiveEventHandler: bulk & normal receives
 *   ChainedReceiveExpeditedEventHandler: buld & OOB recvs
 *   ReceiveDatagramEventHandler: udp receives
 *   ChainedReceiveDatagramEventHandler: udp buld receives
 *
 * Return Value:
 *   NTSTATUS:     kernel status code (STATUS_SUCCESS
 *                 or other error code)
 *
 * NOTES:
 *   N/A
 */
NTSTATUS
KsSetEventHandlers(
    IN PFILE_OBJECT     AddressObject,  // Address File Object
    IN PVOID            EventContext,   // Context for Handlers
    IN ULONG            Flags,          // Handler Indict Bits
    IN PTDI_IND_CONNECT                     ConnectEventHandler,
    IN PTDI_IND_DISCONNECT                  DisconnectEventHandler,
    IN PTDI_IND_ERROR                       ErrorEventHandler,
    IN PTDI_IND_ERROR_EX                    ErrorExEventHandler,
    IN PTDI_IND_SEND_POSSIBLE               SendPossibleHandler,
    IN PTDI_IND_RECEIVE                     ReceiveEventHandler,
```

```
    IN PTDI_IND_RECEIVE_EXPEDITED           ReceiveExpeditedEventHandler,
    IN PTDI_IND_CHAINED_RECEIVE             ChainedReceiveEventHandler,
    IN PTDI_IND_CHAINED_RECEIVE_EXPEDITED   ChainedReceiveExpeditedEventHandler,
    IN PTDI_IND_RECEIVE_DATAGRAM            ReceiveDatagramEventHandler,
    IN PTDI_IND_CHAINED_RECEIVE_DATAGRAM    ChainedReceiveDatagramEventHandler
   )
{
    NTSTATUS             Status = STATUS_SUCCESS;
    PDEVICE_OBJECT       DeviceObject;
    DeviceObject = IoGetRelatedDeviceObject(AddressObject);
    //
    // Setting the Connect Event Handler
    //
    if (IsHandlerSet(Flags, CB_CONNECT))
    {
        PIRP            Irp;
        //
        // Building Tdi Internal Irp ...
        //
        Irp = KsBuildTdiIrp(DeviceObject);
        if (NULL == Irp) {
            Status = STATUS_INSUFFICIENT_RESOURCES;
        } else {
            //
            // Building the Irp to Set the Event Handler ...
            //
            TdiBuildSetEventHandler(
                Irp,
                DeviceObject,
                AddressObject,
                NULL,
                NULL,
                TDI_EVENT_CONNECT,
                ConnectEventHandler,
                EventContext
                );
            //
            // Calling the Transprot Driver with the Prepared Irp
            //
            Status = KsSubmitTdiIrp(DeviceObject, Irp, TRUE, NULL);
        }

        if (!NT_SUCCESS(Status))
        {
            cfs_enter_debugger();
            goto errorout;
```

```
        }
    }
    ......
errorout:
    if (!NT_SUCCESS(Status)) {
        KsPrint((2, "KsSetEventHandlers: Error Status = %xh (%s)\n",
                    Status, KsNtStatusToString(Status) ));
    }
    return (Status);
}
```

Use case:

```
Please refer ksocknal_set_handlers:
/* set all the event callbacks to NULL */
status = KsSetEventHandlers(
  tconn->kstc_addr.FileObject,   /* Address File Object  */
  tconn,                         /* Event Context */
  flags,                         /* Handler Setting Flags*/
  KsConnectEventHandler,         /* ConnectEventHandler */
  KsDisconnectEventHandler,      /* DisconnectEventHandler */
  KsErrorEventHandler,           /* Error Handler */
  NULL,                          /* ERROR_EX */
  NULL,                          /* SendPossibleHandler */
  KsTcpReceiveEventHandler,      /* ReceiveEventHandler */
  KsTcpReceiveExpeditedEventHandler,/* ReceiveExpedited ... */
  KsTcpChainedReceiveEventHandler,  /* ChainedReceive ... */
  KsTcpChainedReceiveExpeditedEventHandler,   /* ChainedReceiveExpedited... */
  NULL,                          /* Udp ... */
  NULL
  );
```

2.6.10 KsQueryAddressInfo

Logic:

```
/*
 * KsQueryAddressInfo
 *   Query the address of the FileObject specified
 *
 * Arguments:
 *   FileObject:  the FileObject to be queried
 *   AddressInfo: buffer to contain the address info
 *   AddressSize: length of the AddressInfo buffer
 *
 * Return Value:
 *   NTSTATUS:     kernel status code (STATUS_SUCCESS
```

```
 *                 or other error code)
 *
 * Notes:
 *   N/A
 */
NTSTATUS
KsQueryAddressInfo(
    PFILE_OBJECT            FileObject,
    PTDI_ADDRESS_INFO       AddressInfo,
    PULONG                  AddressSize
   )
{
    NTSTATUS          Status = STATUS_UNSUCCESSFUL;
    PIRP              Irp = NULL;
    PMDL              Mdl;
    PDEVICE_OBJECT    DeviceObject;
    LASSERT( KeGetCurrentIrql() < DISPATCH_LEVEL );
    DeviceObject = IoGetRelatedDeviceObject(FileObject);
    RtlZeroMemory(AddressInfo, *(AddressSize));
    //
    // Allocating the Tdi Setting Irp ...
    //
    Irp = KsBuildTdiIrp(DeviceObject);
    if (NULL == Irp) {
        Status = STATUS_INSUFFICIENT_RESOURCES;
    } else {
        //
        // Locking the User Buffer / Allocating a MDL for it
        //
        Status = KsLockUserBuffer(
                    AddressInfo,
                    *(AddressSize),
                    IoModifyAccess,
                    &Mdl
                    );
        if (!NT_SUCCESS(Status)) {
            IoFreeIrp(Irp);
            Irp = NULL;
        }
    }
    if (Irp) {
        LASSERT(NT_SUCCESS(Status));
        TdiBuildQueryInformation(
                    Irp,
                    DeviceObject,
                    FileObject,
```

```
                    NULL,
                    NULL,
                    TDI_QUERY_ADDRESS_INFO,
                    Mdl
                    );
        Status = KsSubmitTdiIrp(
                    DeviceObject,
                    Irp,
                    TRUE,
                    AddressSize
                    );
        KsReleaseMdl(Mdl);
    }
    if (!NT_SUCCESS(Status)) {
        cfs_enter_debugger();
        //TDI_BUFFER_OVERFLOW
    }
    return (Status);
}
```

Use case:

```
Please refer KsQueryIpAddress:
    Status = KsQueryAddressInfo(
        FileObject,
        TdiAddressInfo,
        &Length
        );
    ......
```

2.6.11 KsQueryProviderInfo

Logic:

```
/*
 * KsQueryProviderInfo
 *   Query the underlying transport device's information
 *
 * Arguments:
 *   TdiDeviceName:  the transport device's name string
 *   ProviderInfo:   TDI_PROVIDER_INFO struncture
 *
 * Return Value:
 *   NTSTATUS:       Nt system status code
  *
 * NOTES:
 *   N/A
```

```
 */
NTSTATUS
KsQueryProviderInfo(
    PWSTR               TdiDeviceName,
    PTDI_PROVIDER_INFO  ProviderInfo
   )
{
    NTSTATUS            Status = STATUS_SUCCESS;
    PIRP                Irp = NULL;
    PMDL                Mdl = NULL;
    UNICODE_STRING      ControlName;
    HANDLE              Handle;
    PFILE_OBJECT        FileObject;
    PDEVICE_OBJECT      DeviceObject;
    ULONG               ProviderSize = 0;
    RtlInitUnicodeString(&ControlName, TdiDeviceName);
    //
    // Open the Tdi Control Channel
    //
    Status = KsOpenControl(
                &ControlName,
                &Handle,
                &FileObject
                );
    if (!NT_SUCCESS(Status)) {
        KsPrint((2, "KsQueryProviderInfo: Fail to open the tdi control channel.\n"));
        return (Status);
    }
    //
    // Obtain The Related Device Object
    //
    DeviceObject = IoGetRelatedDeviceObject(FileObject);
    ProviderSize = sizeof(TDI_PROVIDER_INFO);
    RtlZeroMemory(ProviderInfo, ProviderSize);
    //
    // Allocating the Tdi Setting Irp ...
    //
    Irp = KsBuildTdiIrp(DeviceObject);
    if (NULL == Irp) {
        Status = STATUS_INSUFFICIENT_RESOURCES;
    } else {
        //
        // Locking the User Buffer / Allocating a MDL for it
        //
        Status = KsLockUserBuffer(
                    ProviderInfo,
```

```
                    ProviderSize,
                    IoModifyAccess,
                    &Mdl
                    );
        if (!NT_SUCCESS(Status)) {
            IoFreeIrp(Irp);
            Irp = NULL;
        }
    }
    if (Irp) {
        LASSERT(NT_SUCCESS(Status));
        TdiBuildQueryInformation(
                    Irp,
                    DeviceObject,
                    FileObject,
                    NULL,
                    NULL,
                    TDI_QUERY_PROVIDER_INFO,
                    Mdl
                    );
        Status = KsSubmitTdiIrp(
                    DeviceObject,
                    Irp,
                    TRUE,
                    &ProviderSize
                    );
        KsReleaseMdl(Mdl);
    }
    if (!NT_SUCCESS(Status)) {
        cfs_enter_debugger();
        //TDI_BUFFER_OVERFLOW
    }
    KsCloseControl(Handle, FileObject);
    return (Status);
}
```

Use case:

```
TDI_PROVIDER_INFO   ProviderInfo;
/* query the tcp transport provider's info */
Status = KsQueryProviderInfo(
             TCP_DEV_NAME,
             &ProviderInfo
         );
```

2.6.12 KsQueryConnectionInfo

Logic:

```
/*
 * KsQueryConnectionInfo
 *   Query the connection info of the FileObject specified
 *   (some statics data of the traffic)
 *
 * Arguments:
 *   FileObject:     the FileObject to be queried
 *   ConnectionInfo: buffer to contain the connection info
 *   ConnectionSize: length of the ConnectionInfo buffer
 *
 * Return Value:
 *   NTSTATUS:     kernel status code (STATUS_SUCCESS
 *                 or other error code)
 *
 * NOTES:
 *   N/A
 */
NTSTATUS
KsQueryConnectionInfo(
    PFILE_OBJECT            ConnectionObject,
    PTDI_CONNECTION_INFO    ConnectionInfo,
    PULONG                  ConnectionSize
   )
{
    NTSTATUS          Status = STATUS_UNSUCCESSFUL;
    PIRP              Irp = NULL;
    PMDL              Mdl;
    PDEVICE_OBJECT    DeviceObject;
    LASSERT( KeGetCurrentIrql() < DISPATCH_LEVEL );
    DeviceObject = IoGetRelatedDeviceObject(ConnectionObject);
    RtlZeroMemory(ConnectionInfo, *(ConnectionSize));
    //
    // Allocating the Tdi Query Irp ...
    //
    Irp = KsBuildTdiIrp(DeviceObject);
    if (NULL == Irp) {
        Status = STATUS_INSUFFICIENT_RESOURCES;
    } else {
        //
        // Locking the User Buffer / Allocating a MDL for it
        //
        Status = KsLockUserBuffer(
```

```
                    ConnectionInfo,
                    *(ConnectionSize),
                    IoModifyAccess,
                    &Mdl
                    );
            if (NT_SUCCESS(Status)) {
                IoFreeIrp(Irp);
                Irp = NULL;
            }
        }
        if (Irp) {
            LASSERT(NT_SUCCESS(Status));
            TdiBuildQueryInformation(
                    Irp,
                    DeviceObject,
                    ConnectionObject,
                    NULL,
                    NULL,
                    TDI_QUERY_CONNECTION_INFO,
                    Mdl
                    );
            Status = KsSubmitTdiIrp(
                    DeviceObject,
                    Irp,
                    TRUE,
                    ConnectionSize
                    );
            KsReleaseMdl(Mdl);
        }
        return (Status);
    }
    Use case:
    This routine is just implemented, but not used in tdinal.
    TDI_CONNECTION_INFO ConnectionInfo;
    ULONG               ConnectionSize;
    Status  = KsQueryConnectionInfo(
                 ConnectionObject,
                 &ConnectionInfo,
                 &ConnectionSize
                 );
    ......
```

2.6.13 KsInitializeTdiAddress

Logic:

```
/*
 * KsInitializeTdiAddress
 *   Initialize the tdi addresss
 *
 * Arguments:
 *   pTransportAddress: tdi address to be initialized
 *   IpAddress:         the ip address of object
 *   IpPort:            the ip port of the object
 *
 * Return Value:
 *   ULONG: the total size of the tdi address
 *
 * NOTES:
 *   N/A
 */
ULONG
KsInitializeTdiAddress(
    IN OUT PTA_IP_ADDRESS   pTransportAddress,
    IN ULONG                IpAddress,
    IN USHORT               IpPort
    )
{
    pTransportAddress->TAAddressCount = 1;
    pTransportAddress->Address[ 0 ].AddressLength = TDI_ADDRESS_LENGTH_IP;
    pTransportAddress->Address[ 0 ].AddressType   = TDI_ADDRESS_TYPE_IP;
    pTransportAddress->Address[ 0 ].Address[ 0 ].sin_port = IpPort;
    pTransportAddress->Address[ 0 ].Address[ 0 ].in_addr  = IpAddress;
    return (FIELD_OFFSET(TRANSPORT_ADDRESS, Address->Address) + TDI_ADDRESS_LENGTH_IP);
}
```

Use case:

```
N/A
```

2.6.14 KsQueryTdiAddressLength

Logic:

```
/*
 * KsQueryTdiAddressLength
 *   Query the total size of the tdi address
 *
 * Arguments:
 *   pTransportAddress: tdi address to be queried
```

```
 *
 * Return Value:
 *   ULONG: the total size of the tdi address
 *
 * NOTES:
 *   N/A
 */
ULONG
KsQueryTdiAddressLength(
    PTRANSPORT_ADDRESS      pTransportAddress
    )
{
    ULONG                   TotalLength = 0;
    LONG                    i;
    PTA_ADDRESS UNALIGNED   pTaAddress = NULL;
    ASSERT (NULL != pTransportAddress);
    TotalLength  = FIELD_OFFSET(TRANSPORT_ADDRESS, Address) +
                   FIELD_OFFSET(TA_ADDRESS, Address) * pTransportAddress->TAAddressCour
    pTaAddress = (TA_ADDRESS UNALIGNED *)pTransportAddress->Address;
    for (i = 0; i < pTransportAddress->TAAddressCount; i++)
    {
        TotalLength += pTaAddress->AddressLength;
        pTaAddress = (TA_ADDRESS UNALIGNED *)((PCHAR)pTaAddress +
                                              FIELD_OFFSET(TA_ADDRESS,Address) +
                                              pTaAddress->AddressLength );
    }
    return (TotalLength);
}
```

Use case:

```
N/A
```

2.6.15 KsQueryIpAddress

Logic:

```
/*
 * KsQueryIpAddress
 *   Query the ip address of the tdi object
 *
 * Arguments:
 *   FileObject: tdi object to be queried
 *   TdiAddress: TdiAddress buffer, to store the queried
 *               tdi ip address
 *   AddressLength: buffer length of the TdiAddress
 *
```

```
 * Return Value:
 *   ULONG: the total size of the tdi ip address
 *
 * NOTES:
 *   N/A
 */
NTSTATUS
KsQueryIpAddress(
    PFILE_OBJECT    FileObject,
    PVOID           TdiAddress,
    ULONG*          AddressLength
    )
{
    NTSTATUS        Status;
    PTDI_ADDRESS_INFO   TdiAddressInfo;
    ULONG               Length;
    //
    // Maximum length of TDI_ADDRESSS_INFO with one TRANSPORT_ADDRESS
    //
    Length = MAX_ADDRESS_LENGTH;
    TdiAddressInfo = (PTDI_ADDRESS_INFO)
                        ExAllocatePoolWithTag(
                            NonPagedPool,
                            Length,
                            'KSAI' );
    if (NULL == TdiAddressInfo) {
        Status = STATUS_INSUFFICIENT_RESOURCES;
        goto errorout;
    }
    Status = KsQueryAddressInfo(
        FileObject,
        TdiAddressInfo,
        &Length
        );
errorout:
    if (NT_SUCCESS(Status))
    {
        if (*AddressLength < Length) {
            Status = STATUS_BUFFER_TOO_SMALL;
        } else {
            *AddressLength = Length;
            RtlCopyMemory(
                TdiAddress,
                &(TdiAddressInfo->Address),
                Length
                );
```

```
            Status = STATUS_SUCCESS;
        }
    } else {
    }
    if (NULL != TdiAddressInfo) {
        ExFreePool(TdiAddressInfo);
    }
    return Status;
}
```

Use case:

```
Refer ksocknal_query_local_ipaddr:
    TdiAddress = &(tconn->kstc_addr.Tdi);
    AddressLength = MAX_ADDRESS_LENGTH;
    status = KsQueryIpAddress(FileObject, TdiAddress, &AddressLength);
    if (NT_SUCCESS(status)) {
        ......
    }
    ......
```

2.6.16 KsErrorEventHandler

Logic:

```
/*
 * KsErrorEventHandler
 *   the common error event handler callback
 *
 * Arguments:
 *   TdiEventContext: should be the socket
 *   Status: the error code
 *
 * Return Value:
 *   Status: STATS_SUCCESS
 *
 * NOTES:
 *   We need not do anything in such a severe
 *   error case. System will process it for us.
 */
NTSTATUS
KsErrorEventHandler(
    IN PVOID        TdiEventContext,
    IN NTSTATUS     Status
   )
{
    KsPrint((2, "KsErrorEventHandler called at Irql = %xh ...\n",
```

```
                KeGetCurrentIrql()));
    cfs_enter_debugger();
    return (STATUS_SUCCESS);
}
```

Use case:

```
It's the tdi event callback routine.
```

2.6.17 ksocknal_set_handlers

Logic:

```
/*
 * ksocknal_set_handlers
 *   setup all the event handler callbacks
 *
 * Arguments:
 *   tconn: the tdi connecton object
 *
 * Return Value:
 *   int: ksocknal error code
 *
 * NOTES:
 *   N/A
 */
int
ksocknal_set_handlers(
    ksock_tconn_t *     tconn
    )
{
    NTSTATUS        status = STATUS_SUCCESS;
    unsigned long   flags = 0;
    if (tconn->kstc_addr.FileObject == NULL) {
        goto errorout;
    }
    /* for sender and listenr, there are different set of callbacks */
    if (tconn->kstc_type == kstt_sender) {
        SetHandler(flags, CB_ERROR);
        SetHandler(flags, CB_DISCONNECT);
        SetHandler(flags, CB_RECEIVE);
        SetHandler(flags, CB_RECEIVE_EXPEDITED);
        SetHandler(flags, CB_CHAINED_RECEIVE);
        SetHandler(flags, CB_CHAINED_RECEIVE_EXPEDITED);
    } else if (tconn->kstc_type == kstt_listener) {
        SetHandler(flags, CB_ERROR);
        SetHandler(flags, CB_CONNECT);
```

```
            SetHandler(flags, CB_DISCONNECT);
            SetHandler(flags, CB_RECEIVE);
            SetHandler(flags, CB_RECEIVE_EXPEDITED);
            SetHandler(flags, CB_CHAINED_RECEIVE);
            SetHandler(flags, CB_CHAINED_RECEIVE_EXPEDITED);
        } else {
            goto errorout;
        }
        /* set all the event callbacks to NULL */
        status = KsSetEventHandlers(
                    tconn->kstc_addr.FileObject, /* Address File Object  */
                    tconn,                   /* Event Context */
                    flags,                   /* Handler Setting Flags */
                    KsConnectEventHandler,   /* ConnectEventHandler */
                    KsDisconnectEventHandler,/* DisconnectEventHandler */
                    KsErrorEventHandler,     /* Error Handler */
                    NULL,                    /* ERROR_EX */
                    NULL,                    /* SendPossibleHandler */
                    KsTcpReceiveEventHandler,/* ReceiveEventHandler */
                    KsTcpReceiveExpeditedEventHandler, /* ReceiveExpedited ... */
                    KsTcpChainedReceiveEventHandler, /* ChainedReceive ... */
                    KsTcpChainedReceiveExpeditedEventHandler,   /* ChainedReceiveExpedited.
                    NULL,                    /* Udp ... */
                    NULL
                    );
    errorout:
        return cfs_error_code(status);
    }
```

Use case:

```
    Please refer function: ksocknal_build_tconn:

    /* set the event callbacks */
    rc = ksocknal_set_handlers(tconn);
    if (rc < 0) {
        cfs_enter_debugger();
        goto errorout;
    }
```

2.6.18 ksocknal_reset_handlers

Logic:

```
    /*
     * ksocknal_reset_handlers
     *   disable all the event handler callbacks (set to NULL)
```

```
 *
 * Arguments:
 *   tconn: the tdi connecton object
 *
 * Return Value:
 *   int: ksocknal error code
 *
 * NOTES:
 *   N/A
 */
int
ksocknal_reset_handlers(
    ksock_tconn_t *     tconn
    )
{
    NTSTATUS        status = STATUS_SUCCESS;
    unsigned long   flags = 0;
    if (tconn->kstc_addr.FileObject == NULL) {
        goto errorout;
    }
    /* for sender and listenr, there are different set of callbacks */
    if (tconn->kstc_type == kstt_sender) {
        SetHandler(flags, CB_ERROR);
        SetHandler(flags, CB_DISCONNECT);
        SetHandler(flags, CB_RECEIVE);
        SetHandler(flags, CB_RECEIVE_EXPEDITED);
        SetHandler(flags, CB_CHAINED_RECEIVE);
        SetHandler(flags, CB_CHAINED_RECEIVE_EXPEDITED);
    } else if (tconn->kstc_type == kstt_listener) {
        SetHandler(flags, CB_ERROR);
        SetHandler(flags, CB_CONNECT);
        SetHandler(flags, CB_DISCONNECT);
        SetHandler(flags, CB_RECEIVE);
        SetHandler(flags, CB_RECEIVE_EXPEDITED);
        SetHandler(flags, CB_CHAINED_RECEIVE);
        SetHandler(flags, CB_CHAINED_RECEIVE_EXPEDITED);
    } else {
        goto errorout;
    }
    /* set all the event callbacks to NULL */
    status = KsSetEventHandlers(
                tconn->kstc_addr.FileObject, /* Address File Object */
                tconn,    /* Event Context */
                flags,    /* Handler Setting Flags */
                NULL,     /* ConnectEventHandler */
                NULL,     /* DisconnectEventHandler */
```

```
                NULL,     /* Error Handler */
                NULL,     /* ERROR_EX */
                NULL,
                NULL,     /* ReceiveEventHandler */
                NULL,     /* ReceiveExpedited ... */
                NULL,     /* ChainedReceive ... */
                NULL,     /* ChainedReceiveExpedited... */
                NULL,     /* Udp ... */
                NULL
                );
errorout:
    return cfs_error_code(status);
}
```

Use case:

```
Rerfer ksocknal_disconnect_tconn:
    /* reset all the event handlers to NULL */
    ksocknal_reset_handlers (tconn);
```

2.7 Tconn routines

2.7.1 ksocknal_create_tconn

Logic:

```
/*
 * ksocknal_create_tconn
 *   allocate a new tconn structure from the SLAB cache or
 *   NonPaged sysetm pool
 *
 * Arguments:
 *   N/A
 *
 * Return Value:
 *   ksock_tconn_t *: the address of tconn or NULL if it fails
 *
 * NOTES:
 *   N/A
 */
ksock_tconn_t *
ksocknal_create_tconn()
{
    ksock_tconn_t * tconn = NULL;
    spin_lock(&(ksocknal_data.ksnd_tconn_lock));
    /* allocate ksoc_tconn_t from the slab cache memory */
    tconn = (ksock_tconn_t *)cfs_mem_cache_alloc(
```

```
                ksocknal_data.ksnd_tconn_slab, 0 );
        spin_unlock(&(ksocknal_data.ksnd_tconn_lock));
        if (tconn) {
            /* initialize the tconn ... */
            tconn->kstc_magic = KS_TCONN_MAGIC;

            ExInitializeWorkItem(
                &(tconn->kstc_disconnect.WorkItem),
                KsDisconnectHelper,
                &(tconn->kstc_disconnect)
                );
            KeInitializeEvent(
                    &(tconn->kstc_disconnect.Event),
                    SynchronizationEvent,
                    FALSE );
            ExInitializeWorkItem(
                &(tconn->kstc_destroy),
                ksocknal_destroy_tconn,
                tconn
                );
            spin_lock_init(&(tconn->kstc_lock));
            ksocknal_get_tconn(tconn);
            spin_lock(&(ksocknal_data.ksnd_tconn_lock));
            /* attach it into global list in ksocknal_data */
            list_add(&(tconn->kstc_list), &(ksocknal_data.ksnd_tconns));
            ksocknal_data.ksnd_ntconns++;
            spin_unlock(&(ksocknal_data.ksnd_tconn_lock));
        }
        return (tconn);
    }
```

Use case:

```
    see ksocknal_build_tconn:
    ksock_tconn_t * tconn;
    /* create the tdi connecion structure */
    tconn = ksocknal_create_tconn();
    if (!tconn) {
        return -ENOMEM;
    }
    ......
```

2.7.2 ksocknal_free_tconn

Logic:

```
/*
```

```
 * ksocknal_free_tconn
 *   free the tconn structure to the SLAB cache or NonPaged
 *   sysetm pool
 *
 * Arguments:
 *   tconn:  the tcon is to be freed
 *
 * Return Value:
 *   N/A
 *
 * Notes:
 *   N/A
 */
void
ksocknal_free_tconn(ksock_tconn_t * tconn)
{
    LASSERT(atomic_read(tconn->kstc_refcount) == 0);
    spin_lock(&(ksocknal_data.ksnd_tconn_lock));
    /* remove it from the global list */
    list_del(&tconn->kstc_list);
    ksocknal_data.ksnd_ntconns--;
    /* free the structure memory */
    cfs_mem_cache_free(ksocknal_data.ksnd_tconn_slab, tconn);
   /* if this is the last tconn, it would be safe for
       ksocknal_tdi_fini_data to quit ... */
    if (ksocknal_data.ksnd_ntconns == 0) {
        cfs_wake_event(&ksocknal_data.ksnd_tconn_exit);
    }
    spin_unlock(&(ksocknal_data.ksnd_tconn_lock));
}
```

Use case:

```
see ksocknal_destroy_tconn:
    ......
    /* free the tconn structure ... */
    ksocknal_free_tconn(tconn);
    ......
```

2.7.3 ksocknal_init_listener

Logic:

```
/*
 * ksocknal_init_listener
 *   Initialize the tconn as a listener (daemon)
 *
```

```
 * Arguments:
 *   tconn: the listener tconn
 *
 * Return Value:
 *   N/A
 *
 * Notes:
 *   N/A
 */
void
ksocknal_init_listener(
    ksock_tconn_t * tconn
    )
{
    /* preparation: intialize the tconn members */
    tconn->kstc_type = kstt_listener;
    RtlInitUnicodeString(&(tconn->kstc_dev), TCP_DEVICE_NAME);
    CFS_INIT_LIST_HEAD(&(tconn->listener.kstc_listening.list));
    CFS_INIT_LIST_HEAD(&(tconn->listener.kstc_accepted.list));
    init_event( &(tconn->listener.kstc_accept_event),
                TRUE,
                FALSE );
    init_event( &(tconn->listener.kstc_destroy_event),
                TRUE,
                FALSE );
    tconn->kstc_type = ksts_inited;
}
Use case:
N/A
```

2.7.4 ksocknal_init_sender

Logic:

```
/*
 * ksocknal_init_sender
 *   Initialize the tconn as a sender
 *
 * Arguments:
 *   tconn: the sender tconn
 *
 * Return Value:
 *   N/A
 *
 * Notes:
 *   N/A
```

```
 */
void
ksocknal_init_sender(
    ksock_tconn_t * tconn
    )
{
    tconn->kstc_type = kstt_sender;
    RtlInitUnicodeString(&(tconn->kstc_dev), TCP_DEVICE_NAME);
    KsInitializeKsChain(&(tconn->sender.kstc_recv));
    tconn->kstc_type = ksts_inited;
}
```

Use case:

```
N/A
```

2.7.5 ksocknal_init_child

Logic:

```
/*
 * ksocknal_init_child
 *   Initialize the tconn as a child
 *
 * Arguments:
 *   tconn: the child tconn
 *
 * Return Value:
 *   N/A
 *
 * NOTES:
 *   N/A
 */
void
ksocknal_init_child(
    ksock_tconn_t * tconn
    )
{
    tconn->kstc_type = kstt_child;
    RtlInitUnicodeString(&(tconn->kstc_dev), TCP_DEVICE_NAME);
    KsInitializeKsChain(&(tconn->child.kstc_recv));
    tconn->kstc_type = ksts_inited;
}
```

Use case:

```
N/A
```

2.7.6 ksocknal_get_tconn

Logic:

```
/*
 * ksocknal_get_tconn
 *   increase the reference count of the tconn with 1
 *
 * Arguments:
 *   tconn: the tdi connection to be referred
 *
 * Return Value:
 *   N/A
 *
 * NOTES:
 *   N/A
 */

void
ksocknal_get_tconn(
    ksock_tconn_t * tconn
    )
{
    atomic_inc(&(tconn->kstc_refcount));
}
```

Use case:

```
N/A
```

2.7.7 ksocknal_put_tconn

Logic:

```
/*
 * ksocknal_put_tconn
 *   decrease the reference count of the tconn and destroy
 *   it if the refercount becomes 0.
 *
 * Arguments:
 *   tconn: the tdi connection to be dereferred
 *
 * Return Value:
 *   N/A
 *
 * NOTES:
 *   N/A
 */
```

```
void
ksocknal_put_tconn(
    ksock_tconn_t *tconn
    )
{
    if (atomic_dec_and_test(&(tconn->kstc_refcount))) {
        spin_lock(&(tconn->kstc_lock));
        if ( ( tconn->kstc_type == kstt_child ||
               tconn->kstc_type == kstt_sender ) &&
             ( tconn->kstc_state == ksts_connected ) ) {
            spin_unlock(&(tconn->kstc_lock));
            ksocknal_abort_tconn(tconn);
        } else {
          if (cfs_is_flag_set(tconn->kstc_flags, KS_..._DESTROY_BUSY)) {
                cfs_enter_debugger();
            } else {
                ExQueueWorkItem(
                        &(tconn->kstc_destory),
                        DelayedWorkQueue
                        );
                cfs_set_flag(tconn->kstc_flags, KS_TCONN_DESTROY_BUSY);
            }
            spin_unlock(&(tconn->kstc_lock));
        }
    }
}
```

Use case:

```
N/A
```

2.7.8 ksocknal_destroy_tconn

Logic:

```
/*
 * ksocknal_destroy_tconn
 *   cleanup the tdi connection and free it
 *
 * Arguments:
 *   tconn: the tdi connection to be cleaned.
 *
 * Return Value:
 *   N/A
 *
 * NOTES:
 *   N/A
```

```
 */
void
ksocknal_destroy_tconn(
    ksock_tconn_t *     tconn
    )
{
    LASSERT(tconn->kstc_refcount.counter == 0);
    ksocknal_reset_handlers(tconn);
    if (tconn->kstc_type == kstt_listener) {
        /* for listener, we just need to close the address object */
        KsCloseAddress(
                tconn->kstc_addr.Handle,
                tconn->kstc_addr.FileObject
                );
        spin_lock(&tconn->kstc_lock);
        tconn->kstc_state = ksts_inited;
        spin_unlock(&tconn->kstc_lock);
    } else if (tconn->kstc_type == kstt_child) {
        /* for child tdi conections */
        /* disassociate the relation between it's connection object
           and the address object */
        if (tconn->kstc_state == ksts_associated) {
            KsDisassociateAddress(
                tconn->child.kstc_info.FileObject
                );
        }
        /* release the connection object */
        KsCloseConnection(
                tconn->child.kstc_info.Handle,
                tconn->child.kstc_info.FileObject
                );
        /* release it's refer of it's parent's address object */
        KsCloseAddress(
                NULL,
                tconn->kstc_addr.FileObject
                );
        spin_lock(&tconn->child.kstc_parent->kstc_lock);
        spin_lock(&tconn->kstc_lock);
        tconn->kstc_state = ksts_inited;
        /* remove it frome it's parent's queues */
        if (tconn->child.kstc_queued) {
            list_del(&(tconn->child.kstc_link));
            if (tconn->child.kstc_queueno) {
                LASSERT(tconn->child.kstc_parent->listener.kstc_accepted.num > 0);
            tconn->child.kstc_parent->listener.kstc_accepted.num -= 1;
            } else {
```

```
                LASSERT(tconn->child.kstc_parent->listener.kstc_listening.num > 0);
            tconn->child.kstc_parent->listener.kstc_listening.num -= 1;
            }
            tconn->child.kstc_queued = FALSE;
        }
        spin_unlock(&tconn->kstc_lock);
        spin_unlock(&tconn->child.kstc_parent->kstc_lock);
        /* drop the reference of the parent tconn */
        ksocknal_put_tconn(tconn->child.kstc_parent);
    } else if (tconn->kstc_type == kstt_sender) {
        /* release the connection object */
        KsCloseConnection(
                tconn->sender.kstc_info.Handle,
                tconn->sender.kstc_info.FileObject
                );
        /* release it's refer of it's parent's address object */
        KsCloseAddress(
                tconn->kstc_addr.Handle,
                tconn->kstc_addr.FileObject
                );
        spin_lock(&tconn->kstc_lock);
        tconn->kstc_state = ksts_inited;
        spin_unlock(&tconn->kstc_lock);
    } else {
        cfs_enter_debugger();
    }
    /* free the tconn structure ... */
    ksocknal_free_tconn(tconn);
}
```

Use case:

```
N/A
```

2.7.9 ksocknal_bind_tconn

Logic:

```
/*
 * ksocknal_bind_tconn
 *   bind the tdi connection object with an address
 *
 * Arguments:
 *   tconn:    tconn to be bound
 *   parent:   the parent tconn object
 *   ipaddr:   the ip address
 *   port:     the port number
```

```
 *
 * Return Value:
 *   int:   0 for success or ksocknal error codes.
 *
 * NOTES:
 *   N/A
 */
int
ksocknal_bind_tconn (
    ksock_tconn_t * tconn,
    ksock_tconn_t * parent,
    unsigned long   addr,
    unsigned short  port
    )
{
    NTSTATUS            status;
    int                 rc = 0;
    ksock_tdi_addr_t    taddr;
    memset(&taddr, 0, sizeof(ksock_tdi_addr_t));
    if (tconn->kstc_state != ksts_inited) {
        status = STATUS_INVALID_PARAMETER;
        rc = cfs_error_code(status);
        goto errorout;
    } else if (tconn->kstc_type == kstt_child) {
        if (NULL == parent) {
            status = STATUS_INVALID_PARAMETER;
            rc = cfs_error_code(status);
            goto errorout;
        }
        /* refer it's parent's address object */
        taddr = parent->kstc_addr;
        ObReferenceObject(taddr.FileObject);
        ksocknal_get_tconn(parent);
    } else {
        PTRANSPORT_ADDRESS TdiAddress = &(taddr.Tdi);
        ULONG              AddrLen = 0;
        /* intialize the tdi address*/
        TdiAddress->TAAddressCount = 1;
        TdiAddress->Address[0].AddressLength = TDI_ADDRESS_LENGTH_IP;
        TdiAddress->Address[0].AddressType   = TDI_ADDRESS_TYPE_IP;
        ((PTDI_ADDRESS_IP)&(TdiAddress->Address[0].Address))->sin_port = port;
        ((PTDI_ADDRESS_IP)&(TdiAddress->Address[0].Address))->in_addr = addr;
        memset(&(((PTDI_ADDRESS_IP)&(TdiAddress->Address[0].Address))->sin_zero[0]),0,8
        /* open the transport address object */
        AddrLen = FIELD_OFFSET(TRANSPORT_ADDRESS, Address->Address) +
                  TDI_ADDRESS_LENGTH_IP;
```

```
        status = KsOpenAddress(
                    &(tconn->kstc_dev),
                    &(taddr.Tdi),
                    AddrLen,
                    &(taddr.Handle),
                    &(taddr.FileObject)
                    );
        if (!NT_SUCCESS(status)) {
            rc = cfs_error_code(status);
            goto errorout;
        }
    }
    spin_lock(&(tconn->kstc_lock));
    if (tconn->kstc_type == kstt_child) {
        tconn->child.kstc_parent = parent;
    }
    tconn->kstc_state = ksts_bind;
    tconn->kstc_addr  = taddr;
    spin_unlock(&(tconn->kstc_lock));
errorout:
    return (rc);
}
```

Use case:

```
N/A
```

2.7.10 ksocknal_build_tconn

Logic:

```
/*
 * ksocknal_build_tconn
 *  build tcp/streaming connection to remote peer
 *
 * Arguments:
 *   tconn:    tconn to be connected to the peer
 *   addr:     the peer's ip address
 *   port:     the peer's port number
 *
 * Return Value:
 *   int:   0 for success or ksocknal error codes.
 *
 * Notes:
 *   N/A
 */
int
```

```
ksocknal_build_tconn(
    ksock_tconn_t *                 tconn,
    unsigned long                   addr,
    unsigned short                  port
    )
{
    int                             rc = 0;
    NTSTATUS                        status = STATUS_SUCCESS;
    PFILE_OBJECT                    ConnectionObject = NULL;
    PDEVICE_OBJECT                  DeviceObject = NULL;
    PTDI_CONNECTION_INFORMATION     ConnectionInfo = NULL;
    ULONG                           AddrLength;
    PIRP                            Irp = NULL;
    LASSERT(tconn->kstc_type == kstt_sender);
    LASSERT(tconn->kstc_state == ksts_bind);
    ksocknal_get_tconn(tconn);
    /* create the connection file handle / object  */
    status = KsOpenConnection(
                &(tconn->kstc_dev),
                (PVOID)tconn,
                &(tconn->sender.kstc_info.Handle),
                &(tconn->sender.kstc_info.FileObject)
                );
    if (!NT_SUCCESS(status)) {
        rc = cfs_error_code(status);
        cfs_enter_debugger();
        goto errorout;
    }
    /* associdate the the connection with the adress object of the tconn */
    status = KsAssociateAddress(
                tconn->kstc_addr.Handle,
                tconn->sender.kstc_info.FileObject
                );
    if (!NT_SUCCESS(status)) {
        rc = cfs_error_code(status);
        cfs_enter_debugger();
        goto errorout;
    }
    spin_lock(&(tconn->kstc_lock));
    tconn->kstc_state = ksts_associated;
    spin_unlock(&(tconn->kstc_lock));
    /* set the event callbacks */
    rc = ksocknal_set_handlers(tconn);
    if (rc < 0) {
        cfs_enter_debugger();
        goto errorout;
```

```
}
/* Allocating Connection Info Together with the Address */
AddrLength = FIELD_OFFSET(TRANSPORT_ADDRESS, Address->Address)
            + TDI_ADDRESS_LENGTH_IP;
ConnectionInfo = (PTDI_CONNECTION_INFORMATION)ExAllocatePoolWithTag(
NonPagedPool, sizeof(TDI_CONNECTION_INFORMATION) + AddrLength, 'iCsK');
if (NULL == ConnectionInfo) {
    status = STATUS_INSUFFICIENT_RESOURCES;
    rc = cfs_error_code(status);
    cfs_enter_debugger();
    goto errorout;
}
/* Initializing ConnectionInfo ... */
{
    PTRANSPORT_ADDRESS TdiAddress;
    /* ConnectionInfo settings */
    ConnectionInfo->UserDataLength = 0;
    ConnectionInfo->UserData = NULL;
    ConnectionInfo->OptionsLength = 0;
    ConnectionInfo->Options = NULL;
    ConnectionInfo->RemoteAddressLength = AddrLength;
    ConnectionInfo->RemoteAddress = ConnectionInfo + 1;
    /* intialize the tdi address*/
    TdiAddress = ConnectionInfo->RemoteAddress;
    TdiAddress->TAAddressCount = 1;
    TdiAddress->Address[0].AddressLength = TDI_ADDRESS_LENGTH_IP;
    TdiAddress->Address[0].AddressType   = TDI_ADDRESS_TYPE_IP;
    ((PTDI_ADDRESS_IP)&(TdiAddress->Address[0].Address))->sin_port = port;
    ((PTDI_ADDRESS_IP)&(TdiAddress->Address[0].Address))->in_addr = addr;
    memset(&(((PTDI_ADDRESS_IP)&(TdiAddress->Address[0].Address))->sin_zero[0]),0,8
}
/* Now prepare to connect the remote peer ... */
ConnectionObject = tconn->sender.kstc_info.FileObject;
DeviceObject = IoGetRelatedDeviceObject(ConnectionObject);
/* allocate a new Irp */
Irp = KsBuildTdiIrp(DeviceObject);
if (NULL == Irp) {
    status = STATUS_INSUFFICIENT_RESOURCES;
    rc = cfs_error_code(status);
    cfs_enter_debugger();
    goto errorout;
}
/* setup the Irp */
TdiBuildConnect(
        Irp,
        DeviceObject,
```

```
            ConnectionObject,
            NULL,
            NULL,
            NULL,
            ConnectionInfo,
            NULL
            );
        /* sumbit the Irp to the underlying transport driver */
        status = KsSubmitTdiIrp(
                        DeviceObject,
                        Irp,
                        TRUE,
                        NULL
                        );
        spin_lock(&(tconn->kstc_lock));
        if (NT_SUCCESS(status)) {
            /* Connected! the conneciton is built successfully. */
            tconn->kstc_state = ksts_connected;
            tconn->sender.kstc_info.ConnectionInfo = ConnectionInfo;
            tconn->sender.kstc_info.Remote         = ConnectionInfo->RemoteAddress;
            spin_unlock(&(tconn->kstc_lock));
        } else {
            /* Not connected! Abort it ... */
            Irp = NULL;
            rc = cfs_error_code(status);
            tconn->kstc_state = ksts_associated;
            spin_unlock(&(tconn->kstc_lock));
            /* reset all the event handlers */
            ksocknal_reset_handlers(tconn);
            /* disassocidate the connection and the address object,
               after cleanup,  it's safe to set the state to abort ... */
            if ( NT_SUCCESS(KsDisassociateAddress(
                            tconn->sender.kstc_info.FileObject)) {
                tconn->kstc_state = ksts_aborted;
            }

            goto errorout;
        }
    errorout:
        if (NT_SUCCESS(status)) {
            ksocknal_query_local_ipaddr(tconn);
        } else {
            if (ConnectionInfo) {
                ExFreePool(ConnectionInfo);
            }
            if (Irp) {
```

```
            IoFreeIrp(Irp);
        }
    }
    ksocknal_put_tconn(tconn);
    return (rc);
}
```

Use case:

```
N/A
```

2.7.11 ksocknal_disconnect_tconn

Logic:

```
/*
 * ksocknal_disconnect_tconn
 *   disconnect the tconn from a connection
 *
 * Arguments:
 *   tconn: the tdi connecton object connected already
 *   flags: flags & options for disconnecting
 *
 * Return Value:
 *   int: ksocknal error code
 *
 * Notes:
 *   N/A
 */
int
ksocknal_disconnect_tconn(
    ksock_tconn_t *     tconn,
    unsigned long       flags
    )
{
    NTSTATUS            status = STATUS_SUCCESS;
    ksock_tconn_info_t * info;

    PFILE_OBJECT        ConnectionObject;
    PDEVICE_OBJECT      DeviceObject = NULL;
    PIRP                Irp = NULL;
    KEVENT              Event;
    ksocknal_get_tconn(tconn);
    /* make sure tt's connected already and it
       must be a sender or a child ...       */
    LASSERT(tconn->kstc_state == ksts_connected);
    LASSERT( tconn->kstc_type == kstt_sender ||
```

```
        tconn->kstc_type == kstt_child);
/* reset all the event handlers to NULL */
if (tconn->kstc_type != kstt_child) {
    ksocknal_reset_handlers (tconn);
}
/* Disconnecting to the remote peer ... */
if (tconn->kstc_type == kstt_sender) {
    info = &(tconn->sender.kstc_info);
} else {
    info = &(tconn->child.kstc_info);
}
ConnectionObject = info->FileObject;
DeviceObject = IoGetRelatedDeviceObject(ConnectionObject);
/* allocate an Irp and setup it */
Irp = KsBuildTdiIrp(DeviceObject);
if (NULL == Irp) {
    status = STATUS_INSUFFICIENT_RESOURCES;
    cfs_enter_debugger();
    goto errorout;
}
KeInitializeEvent(
        &Event,
        SynchronizationEvent,
        FALSE
        );
TdiBuildDisconnect(
        Irp,
        DeviceObject,
        ConnectionObject,
        KsDisconectCompletionRoutine,
        &Event,
        NULL,
        flags,
        NULL,
        NULL
        );
/* issue the Irp to the underlying transport
   driver to disconnect the connection    */
status = IoCallDriver(DeviceObject, Irp);
if (STATUS_PENDING == status) {
    status = KeWaitForSingleObject(
                &Event,
                Executive,
                KernelMode,
                FALSE,
                NULL
```

```
                    );
        status = Irp->IoStatus.Status;
    }
    KsPrint((2, "KsDisconnect: Disconnection is done with Status = %xh (%s) ...\n",
                status, KsNtStatusToString(status)));
    IoFreeIrp(Irp);
    if (info->ConnectionInfo) {
        /* disassociate the association between connection/address objects */
        status = KsDisassociateAddress(ConnectionObject);
        if (!NT_SUCCESS(status)) {
            cfs_enter_debugger();
        }
        spin_lock(&(tconn->kstc_lock));
        /* cleanup the tsdumgr Lists */
        KsCleanupTsdu (tconn);
        /* set the state of the tconn */
        if (NT_SUCCESS(status)) {
            tconn->kstc_state = ksts_disconnected;
        } else {
            tconn->kstc_state = ksts_associated;
        }
        /* free  the connection info to system pool*/
        ExFreePool(info->ConnectionInfo);
        info->ConnectionInfo = NULL;
        info->Remote = NULL;
        spin_unlock(&(tconn->kstc_lock));
    }
    status = STATUS_SUCCESS;
errorout:
    ksocknal_put_tconn(tconn);
    return cfs_error_code(status);
}
```

Use case:

```
N/A
```

2.7.12 ksocknal_abort_tconn

Logic:

```
/*
 * ksocknal_abort_tconn
 *   The connection is broken un-expectedly. We need do
 *   some cleanup.
 *
 * Arguments:
```

```
 *   tconn: the tdi connection
 *
 * Return Value:
 *   N/A
 *
 * Notes:
 *   N/A
 */
void
ksocknal_abort_tconn(
    ksock_tconn_t *     tconn
    )
{
    PKS_DISCONNECT_WORKITEM WorkItem = NULL;
    WorkItem = &(tconn->kstc_disconnect);
    ksocknal_get_tconn(tconn);
    spin_lock(&(tconn->kstc_lock));
    if (tconn->kstc_state != ksts_connected) {
    } else {
        if (!cfs_is_flag_set(tconn->kstc_flags, KS_TCONN_DISCONNECT_BUSY)) {
            WorkItem->Flags = TDI_DISCONNECT_ABORT;
            WorkItem->tconn = tconn;
            cfs_set_flag(tconn->kstc_flags, KS_TCONN_DISCONNECT_BUSY);
            ExQueueWorkItem(
                    &(WorkItem->WorkItem),
                    DelayedWorkQueue
                    );
            ksocknal_get_tconn(tconn);
        }
    }
    spin_unlock(&(tconn->kstc_lock));
    ksocknal_put_tconn(tconn);
}
```

Use case:

```
N/A
```

2.7.13 ksocknal_query_local_ipaddr

Logic:

```
/*
 * ksocknal_query_local_ipaddr
 *   query the local connection ip address
 *
 * Arguments:
```

```
 *   tconn:  the tconn which is connected
 *
 * Return Value:
 *   int: ksocknal error code
 *
 * Notes:
 *   N/A
 */
int
ksocknal_query_local_ipaddr(
    ksock_tconn_t *     tconn
    )
{
    PFILE_OBJECT    FileObject = NULL;
    NTSTATUS        status;

    PTRANSPORT_ADDRESS TdiAddress;
    ULONG              AddressLength;
    if (tconn->kstc_type == kstt_sender) {
        FileObject = tconn->sender.kstc_info.FileObject;
    } else if (tconn->kstc_type == kstt_child) {
        FileObject = tconn->child.kstc_info.FileObject;
    } else {
        status = STATUS_INVALID_PARAMETER;
        goto errorout;
    }
    TdiAddress = &(tconn->kstc_addr.Tdi);
    AddressLength = MAX_ADDRESS_LENGTH;
    status =  KsQueryIpAddress(FileObject, TdiAddress, &AddressLength);
    if (NT_SUCCESS(status)) {
        KsPrint((0, "ksocknal_query_local_ipaddr: Local ip address = %xh port = %xh\n",
                ((PTDI_ADDRESS_IP)(&(TdiAddress->Address[0].Address)))->in_addr,
                ((PTDI_ADDRESS_IP)(&(TdiAddress->Address[0].Address)))->sin_port ));
    } else {
        KsPrint((0, "KsQueryonnectionIpAddress: Failed to query the connection local ip
    }
errorout:
    return cfs_error_code(status);
}
```

2.8 daemon routines

2.8.1 ksocknal_alloc_daemon

Logic:

```
/*
```

```
 * ksocknal_alloc_daemon
 *   allocate a ksock_daemon_t structure
 *
 * Arguments:
 *   port:      the port number to listen on
 *   backlog:   the number of backlog children connections
 *
 * Return Value:
 *   the adress of the alocate ksoc_daemon or NULL in failure.
 *
 * Notes:
 *   N/A
 */
struct ksock_daemon *
ksocknal_alloc_daemon(unsigned short port, int backlog)
{
    struct ksock_daemon * daemon = (struct ksock_daemon *)
            cfs_alloc(sizeof(struct ksock_daemon), CFS_ALLOC_ZERO);
    if (daemon) {
        daemon->port = port;
        if (backlog == 0) {
            backlog = MAX_CHILD_LISTENERS;
        }
        daemon->nbacklogs = (unsigned short)backlog;
    }
    return daemon;
}
```

Use case:

```
    see ksocknal_start_daemon:
    /* Allocate a KS_DAEMON structure and initialize it. */
    daemon = ksocknal_alloc_daemon(port, backlog);
```

2.8.2 ksocknal_free_daemon

Logic:

```
/*
 * ksocknal_free_daemon
 *   free the ksock_daemon_t structure
 *
 * Arguments:
 *   daemon: the ksocknal daemon structure to be freed
 *
 * Return Value:
 *   N/A
```

```
 *
 * Notes:
 *   N/A
 */
void
ksocknal_free_daemon(struct ksock_daemon * daemon)
{
    LASSERT(daemon);
    cfs_free(daemon);
}
```

Use case:

```
    see ksocknal_shut_daemon:
    ......
    /* 5, now it's time to free the daemon */
    ksocknal_free_daemon(daemon);
    .......
```

2.8.3 ksocknal_daemon

Logic:

```
/*
 * ksocknal_daemon
 *   This is the listening daemon thread for ksocknal
 *
 * Arguments:
 *   Context - Should be the address of the ksocknal_daemon
 *
 * Return Value:
 *   N/A
 *
 * Notes:
 *   N/A
 */
int
ksocknal_daemon(void * context)
{
    int                     rc = 0;
    int                     blink = FALSE;
    struct ksock_daemon *   daemon;
    ksock_tconn_t *         parent;
    ksock_tconn_t *         child;
    NTSTATUS                status;
    KsPrint((2, "ksocknal_daemon: tcp acceptor thread is started !\n"));
    __try {
```

```
daemon = (struct ksock_daemon *) context;
parent = ksocknal_create_tconn();
if (!parent) {
    status = STATUS_INSUFFICIENT_RESOURCES;
    __leave;
}
/* initialize the tconn as a listener */
ksocknal_init_listener(parent);
daemon->tconn = parent;
/* bind the daemon->tconn */
rc = ksocknal_bind_tconn(parent, NULL, 0, daemon->port);
if (rc < 0) {
    ksocknal_free_tconn(parent);
    daemon->tconn = NULL;
    __leave;
}
/* create listening children and make it to listen state*/
rc = ksocknal_start_listen(daemon);
if (rc < 0) {
    /* the job of freeing tconn is done by ksocknal_shut_daemon */
    __leave;
}
spin_lock(&ksocknal_data.ksnd_daemon_lock);
/* add it to the global list */
list_add(&(daemon->list), &(ksocknal_data.ksnd_daemons));
ksocknal_data.ksnd_ndaemons++;
blink = TRUE;
while (!daemon->shutdown) {
    spin_unlock(&ksocknal_data.ksnd_daemon_lock);
    /* wait for incoming connecitons */
    rc = ksocknal_wait_child_tconn(daemon, &child);
    if (rc < 0) {
        __leave;
    }
    if (child) {
        /* craete socknal connection ... */
        rc = ksocknal_create_conn(NULL, child, SOCKNAL_CONN_NONE);
        if (rc < 0) {
            cfs_enter_debugger();
        }
        /* release the reference of the child tdi connection */
        ksocknal_put_tconn(child);
    }
    spin_lock(&ksocknal_data.ksnd_daemon_lock);
}
spin_unlock(&ksocknal_data.ksnd_daemon_lock);
```

```
    }
    __finally {
        if (daemon) {
            spin_lock(&ksocknal_data.ksnd_daemon_lock);
            /* remove the daemon from the ksocknal_data lsit */
            if (blink) {
                list_del(&(daemon->list));
                ksocknal_data.ksnd_ndaemons--;
            }
            /* we need signal the event, or ksocnal_fini_tdi_data
               will never get chance to quit */
            if (ksocknal_data.ksnd_ndaemons == 0) {
                cfs_wake_event(&ksocknal_data.ksnd_daemon_exit);
            }
            spin_unlock(&ksocknal_data.ksnd_daemon_lock);
            /* cleanup the daemon and drop the refer */
            ksocknal_shut_daemon(daemon);
        }
    }
    return rc;
}
```

Use case:

```
N/A (it's an independent thread procedure.)
```

2.8.4 ksocknal_shut_daemon

Logic

```
/*
 * ksocknal_shut_daemon
 *   This is the listening daemon thread for ksocknal
 *
 * Arguments:
 *   Context - Should be the address of the ksocknal_daemon
 *
 * Return Value:
 *   N/A
 *
 * Notes:
 *   N/A
 */
void
ksocknal_shut_daemon(struct ksock_daemon *daemon)
{
    struct list_head *      list;
```

```
    ksock_tconn_t *         backlog;
    if (daemon->tconn) {
        /* 1, Reset all tdi event callbacks to NULL */
        ksocknal_reset_handlers (daemon->tconn);
        spin_lock(&daemon->tconn->kstc_lock);
#if 0   /* let's ksocknal release all the tconns connected (maybe busy for data) */
        /* the connected tdi connecton might be busy with network traffic.
           so we just leave the tdi connecton to be released by ksocknal. */
        /* 2, cleanup all the connected backlog child connections */
        list_for_each (list, &(daemon->tconn->listener.kstc_accepted.list)) {
            backlog = list_entry(list, ksock_tconn_t, child.kstc_link);
            /* disconnect, destroy and free it */
            ksocknal_put_tconn(backlog);
        }
#endif
        /* 3, cleanup all the listening backlog child connections */
        list_for_each (list, &(daemon->tconn->listener.kstc_listening.list)) {
            backlog = list_entry(list, ksock_tconn_t, child.kstc_link);
            /* 3.1, destory and free it */
            ksocknal_put_tconn(backlog);
        }

        spin_unlock(&daemon->tconn->kstc_lock);
        /* 4, free the listening deaemon connection ... */
        ksocknal_put_tconn(daemon->tconn);
    }
    /* 5, now it's time to free the daemon */
    ksocknal_free_daemon(daemon);
}
```

Use case:

```
        /* cleanup the daemon and drop the refer */
        ksocknal_shut_daemon(&daemon);
```

2.8.5 ksocknal_start_daemon

Logic:

```
/*
 * ksocknal_start_daemon
 *   start a new deamon thread
 *
 * Arguments:
 *   port:  the prot number to listen on
 *   backlog: number of backlog children connections
 *
```

```
 * Return Value:
 *   >=0: Success or failure
 *
 * Notes:
 *   N/A
 */
int
ksocknal_start_daemon(unsigned short port, int backlog)
{
    int    rc;
    struct list_head *tmp;
    struct ksock_daemon * daemon = NULL;
    /* 1, check whether the daemon exists in the daemon list in ksocknal_data ? */
    spin_lock(&ksocknal_data.ksnd_daemon_lock);
    list_for_each (tmp, &(ksocknal_data.ksnd_daemons)) {
        daemon = list_entry(tmp, struct ksock_daemon, list);
        if (daemon->port == port) {
            break;
        } else {
            daemon = NULL;
        }
    }
    spin_unlock(&ksocknal_data.ksnd_daemon_lock);
    if (daemon) {
        rc = 0;
        goto errorout;
    }
    /* 2, allocate a KS_DAEMON structure and initialize it. */
    daemon = ksocknal_alloc_daemon(port, backlog);
    if (!daemon) {
        rc = -ENOMEM;
        goto errorout;
    }
    /* 3, start ksocknal_daemon_thread with the allocated ksocknal_daemon */
    rc = cfs_kernel_thread(ksocknal_daemon, daemon, 0);
    if (rc < 0) {
        ksocknal_free_daemon(daemon);
        goto errorout;
    }

    /* 4, now everything is ok. */
errorout:
    return rc;
}
```

Use case:

N/A

2.8.6 ksocknal_stop_daemon

Logic:

```
/*
 * ksocknal_stop_daemon
 *   shut down a running deamon thread
 *
 * Arguments:
 *   port:  the prot number that the daemon is listening on
 *
 * Return Value:
 *   N/A
 *
 * Notes:
 *   N/A
 */
void
ksocknal_stop_daemon(unsigned short port)
{
    struct list_head *tmp;
    struct ksock_daemon * daemon = NULL;
    /* 1, check whether the daemon exists in the daemon list
          in ksocknal_data ? */
    spin_lock(&ksocknal_data.ksnd_daemon_lock);
    list_for_each (tmp, &(ksocknal_data.ksnd_daemons)) {
        daemon = list_entry(tmp, struct ksock_daemon, list);
        if (daemon->port == port) {
            break;
        } else {
            daemon = NULL;
        }
    }
    /* 2, set the shutdown flag and wait for the daemon thread to terminate */
    if (daemon) {
        /* mark the flag to shutdonw it */
        daemon->shutdown = TRUE;
        /* wake up it from the waiting on new incoming connections */
        KeSetEvent(&daemon->tconn->listener.kstc_accept_event, 0, FALSE);
    }

    spin_unlock(&ksocknal_data.ksnd_daemon_lock);
    if (!daemon) {
```

```
        goto errorout;
    }
    /* 3, The daemon thread will do the remained cleanup */
errorout:
    return;
}
```

Use case:

```
called in ksocknal_cmd to response the ioctl cmd: NAL_CMD_STOP_DAEMON.
```

2.8.7 ksocknal_stop_all_daemons

Logic:

```
/*
 * ksocknal_stop_all_daemons
 *   stop all the daemon threads (tdinal is to quit)
 *
 * Arguments:
 *   N/A
 *
 * Return Value:
 *   N/A
 *
 * Notes:
 *   N/A
 */
void
ksocknal_stop_all_daemons()
{
    struct list_head *      tmp;
    struct ksock_daemon *   daemon;
    /* we need query and stop all the daemons in the
       daemon list of ksocknal_data  */
    spin_lock(&ksocknal_data.ksnd_daemon_lock);
    /* in the case of the list is empty, we need signal the
       event here, or we will never get chance to quit */
    if (list_empty(&(ksocknal_data.ksnd_daemons))) {
        cfs_wake_event(&ksocknal_data.ksnd_daemon_exit);
        goto errorout;
    }
    list_for_each (tmp, &(ksocknal_data.ksnd_daemons)) {
        /* get the daemon structure */
        daemon = list_entry(tmp, struct ksock_daemon, list);
        /* mark the flag to shutdonw it */
```

```
        daemon->shutdown = TRUE;
        /* wake up it from the waiting on new incoming connections */
        KeSetEvent(&daemon->tconn->listener.kstc_accept_event, 0, FALSE);
    }
errorout:
    spin_unlock(&ksocknal_data.ksnd_daemon_lock);
    /* now waiting until all the daemon threads exit */
    cfs_wait_event(&ksocknal_data.ksnd_daemon_exit, 0);
}
```

Use case:

called by ksocknal_api_shutdown to shutdown all the daemon threads.

2.8.8 ksocknal_create_child_tconn

Logic:

```
/*
 * ksocknal_create_child_tconn
 *   Create the backlog child connection for a listener
 *
 * Arguments:
 *   parent: the listener daemon connection
 *
 * Return Value:
 *   the child connection or NULL in failure
 *
 * Notes:
 *   N/A
 */
ksock_tconn_t *
ksocknal_create_child_tconn(
    ksock_tconn_t * parent
    )
{
    NTSTATUS            status;
    ksock_tconn_t *     backlog;
    /* allocate the tdi connecton object */
    backlog = ksocknal_create_tconn();
    if (!backlog) {
        goto errorout;
    }
    /* initialize the tconn as a child */
    ksocknal_init_child(parent);
    /* now bind it */
```

```
    if (ksocknal_bind_tconn(backlog, parent, 0, 0) < 0) {
        ksocknal_free_tconn(backlog);
        backlog = NULL;
        goto errorout;
    }
    /* open the connection object */
    status = KsOpenConnection(
                &(backlog->kstc_dev),
                (PVOID)backlog,
                &(backlog->child.kstc_info.Handle),
                &(backlog->child.kstc_info.FileObject)
                );

    if (!NT_SUCCESS(status)) {
        ksocknal_put_tconn(backlog);
        backlog = NULL;
        cfs_enter_debugger();
        goto errorout;
    }
    /* associate it now ... */
    status = KsAssociateAddress(
                backlog->kstc_addr.Handle,
                backlog->child.kstc_info.FileObject
                );
    if (!NT_SUCCESS(status)) {

        ksocknal_put_tconn(backlog);
        backlog = NULL;
        cfs_enter_debugger();
        goto errorout;
    }
    backlog->kstc_state = ksts_associated;
errorout:
    return backlog;
}
```

Use case:

```
called by ksocknal_replenish_backlogs to create a listing child tconn.
```

2.8.9 ksocknal_replenish_backlogs

Logic:

```
/*
 * ksocknal_replenish_backlogs(
```

```
 *   to replenish the backlogs listening...
 *
 * Arguments:
 *   daemon: the listener daemon
 *
 * Return Value:
 *   N/A
 *
 * Notes:
 *   N/A
 */
void
ksocknal_replenish_backlogs(
    ksock_daemon_t * daemon
    )
{
    ksock_tconn_t * tconn = daemon->tconn;
    ksock_tconn_t * backlog;
    int            n = 0;
    spin_lock(&(daemon->tconn->kstc_lock));
    /* calculate how many backlogs needed */
    if ( ( tconn->listener.kstc_listening.num +
           tconn->listener.kstc_accepted.num ) < daemon->nbacklogs ) {
        n = daemon->nbacklogs - ( tconn->listener.kstc_listening.num +
           tconn->listener.kstc_accepted.num );
    } else {
        n = 0;
    }
    while (n--) {
        spin_unlock(&(daemon->tconn->kstc_lock));
        /* create the backlog child tconn */
        backlog = ksocknal_create_child_tconn(tconn);
        spin_lock(&(daemon->tconn->kstc_lock));
        if (backlog) {
            spin_lock(&backlog->kstc_lock);
            /* attch it into the listing list of daemon */
            list_add( &backlog->child.kstc_link,
                      &tconn->listener.kstc_listening.list );
            tconn->listener.kstc_listening.num++;
            backlog->child.kstc_queued = TRUE;
            spin_lock(&backlog->kstc_lock);
        } else {
            cfs_enter_debugger();
        }
    }
    spin_unlock(&(daemon->tconn->kstc_lock));
```

```
}
```

Use case:

```
we need maintain several (daemon->nbacklogs) child tconns for incoming connection reque
```

2.8.10 ksocknal_start_listen

Logic:

```
/*
 * ksocknal_start_listen
 *   setup the listener tdi connection and make it
 *   listen on the user specified ipaddr and port
 *
 * Arguments:
 *   daemon: the listener daemon
 *
 * Return Value:
 *   ksocknal error code >=: success; otherwise error.
 *
 * Notes:
 *   N/A
 */
int
ksocknal_start_listen(struct ksock_daemon * daemon)
{
    int rc = 0;
    /* now replenish the backlogs */
    ksocknal_replenish_backlogs(daemon);
    /* set the event callback handlers */
    rc = ksocknal_set_handlers(daemon->tconn);
    if (rc < 0) {
        return rc;
    }
    spin_lock(&(daemon->tconn->kstc_lock));
    daemon->tconn->kstc_state = ksts_listening;
    spin_unlock(&(daemon->tconn->kstc_lock));
    return rc;
}
```

Use case:

```
called by ksocknal_daemon: to create backlog child tconns and issue TdiListen irp to th
```

2.8.11 ksocknal_wait_child_tconn

Logic:

```
/*
 * ksocknal_wait_child_tconn
 *   accept a child connection from peer
 *
 * Arguments:
 *   daemon:   the daemon structure
 *   child:    to contain the accepted connection
 *
 * Return Value:
 *   ksocknal error code;
 *
 * Notes:
 *   N/A
 */
int
ksocknal_wait_child_tconn(
    struct ksock_daemon * daemon,
    ksock_tconn_t ** child
    )
{
    struct list_head * tmp;
    ksock_tconn_t * backlog = NULL;
    ksocknal_replenish_backlogs(daemon);
    spin_lock(&(daemon->tconn->kstc_lock));
    if (daemon->tconn->listener.kstc_listening.num <=0 ) {
        spin_unlock(&(daemon->tconn->kstc_lock));
        return -1;
    }
again:
    /* check the listening queue and try to search the accepted connecton */
    list_for_each(tmp, &(daemon->tconn->listener.kstc_listening.list)) {
        backlog = list_entry (tmp, ksock_tconn_t, child.kstc_link);
        spin_lock(&(backlog->kstc_lock));
        if (backlog->child.kstc_accepted) {
            LASSERT(backlog->kstc_state == ksts_connected);
            LASSERT(backlog->child.kstc_busy);
            list_del(&(backlog->child.kstc_link));
            list_add(&(backlog->child.kstc_link),
                     &(daemon->tconn->listener.kstc_accepted.list));
            daemon->tconn->listener.kstc_accepted.num++;
            daemon->tconn->listener.kstc_listening.num--;
            backlog->child.kstc_queueno = 1;
```

```
            spin_unlock(&(backlog->kstc_lock));
            break;
        } else {
            spin_unlock(&(backlog->kstc_lock));
            backlog = NULL;
        }
    }
    spin_unlock(&(daemon->tconn->kstc_lock));
    /* we need wait until new incoming connections are requested
       or the case of shuting down the listenig daemon thread  */
    if (backlog == NULL) {
        NTSTATUS    Status;
        Status = KeWaitForSingleObject(
                &(daemon->tconn->listener.kstc_accept_event),
                Executive,
                KernelMode,
                FALSE,
                NULL
                );
        spin_lock(&(daemon->tconn->kstc_lock));
        /* check whether it's exptected to exit ? */
        if (daemon->shutdown) {
            spin_unlock(&(daemon->tconn->kstc_lock));
        } else {
            goto again;
        }
    }
    if (backlog) {
        /* query the local ip address of the connection */
        ksocknal_query_local_ipaddr(backlog);
    }
    *child = backlog;
    return 0;
}
```

Use case:

```
called by ksocknal_daemon to get the accepted connection. This funciton will block the
```

2.8.12 ksocknal_get_vacancy_backlog

Logic:

```
/*
 * ksocknal_get_vacancy_backlog
 *   Get a vacancy listeing child from the backlog list
```

```
 *
 * Arguments:
 *   parent: the listener daemon connection
 *
 * Return Value:
 *   the child listening connection or NULL in failure
 *
 * Notes
 *   Parent's lock should be acquired before calling.
 */
ksock_tconn_t *
ksocknal_get_vacancy_backlog(
    ksock_tconn_t *  parent
    )
{
    ksock_tconn_t * child;
    LASSERT(parent->kstc_type == kstt_listener);
    LASSERT(parent->kstc_state == ksts_listening);
    if (list_empty(&(parent->listener.kstc_listening.list))) {
        child = NULL;
    } else {
        struct list_head * tmp;
        /* check the listening queue and try to get a free connecton */
        list_for_each(tmp, &(parent->listener.kstc_listening.list)) {
            child = list_entry (tmp, ksock_tconn_t, child.kstc_link);
            spin_lock(&(child->kstc_lock));
            if (!child->child.kstc_busy) {
                LASSERT(child->kstc_state == ksts_associated);
                child->child.kstc_busy = TRUE;
                spin_unlock(&(child->kstc_lock));
                break;
            } else {
                spin_unlock(&(child->kstc_lock));
                child = NULL;
            }
        }
    }
    return child;
}
```

Use case:

```
called by tdi connect event handler (KsConnectEventHandler) when approving incoming req
```

2.9 Tcp event callbacks

2.9.1 KsDisconnectCompleitonRoutine

Logic:

```
/*
 * KsDisconnectCompletionRoutine
 *   the Irp completion routine for TdiBuildDisconect
 *
 *   We just signal the event and return MORE_PRO... to
 *   let the caller take the responsibility of the Irp.
 *
 * Arguments:
 *   DeviceObject:  the device object of the transport
 *   Irp:           the Irp is being completed.
 *   Context:       the event specified by the caller
 *
 * Return Value:
 *   Nt status code
 *
 * Notes:
 *   N/A
 */
NTSTATUS
KsDisconectCompletionRoutine (
    IN PDEVICE_OBJECT   DeviceObject,
    IN PIRP             Irp,
    IN PVOID            Context
    )
{
    KeSetEvent((PKEVENT) Context, 0, FALSE);
    return STATUS_MORE_PROCESSING_REQUIRED;
    UNREFERENCED_PARAMETER(DeviceObject);
}
```

Use case:

```
N/A
```

2.9.2 KsAcceptCompletionRoutine

Logic:

```
/*
 * KsAcceptCompletionRoutine
```

```
 *   Irp completion routine for TdiBuildAccept (KsConnectEventHandler)
 *
 *   Here system gives us a chance to check the conneciton is built
 *   ready or not.
 *
 * Arguments:
 *   DeviceObject:  the device object of the transport driver
 *   Irp:           the Irp is being completed.
 *   Context:       the context we specified when issuing the Irp
 *
 * Return Value:
 *   Nt status code
 *
 * Notes:
 *   N/A
 */
NTSTATUS
KsAcceptCompletionRoutine(
    IN PDEVICE_OBJECT   DeviceObject,
    IN PIRP             Irp,
    IN PVOID            Context
    )
{
    ksock_tconn_t * child = (ksock_tconn_t *) Context;
    ksock_tconn_t * parent = child->child.kstc_parent;
    KsPrint((2, "KsAcceptCompletionRoutine: called at Irql: %xh\n",
                KeGetCurrentIrql() ));
    KsPrint((2, "KsAcceptCompletionRoutine: Context = %xh Status = %xh\n",
                 Context, Irp->IoStatus.Status));
    LASSERT(child->kstc_type == kstt_child);
    spin_lock(&(child->kstc_lock));
    LASSERT(parent->kstc_state == ksts_listening);
    LASSERT(child->kstc_state == ksts_connecting);
    if (NT_SUCCESS(Irp->IoStatus.Status)) {
        child->child.kstc_accepted = TRUE;
        child->kstc_state = ksts_connected;
        /* wake up the daemon thread which waits on this event */
        KeSetEvent(
            &(parent->listener.kstc_acceptevent),
            0,
            FALSE
            );
        spin_unlock(&(child->kstc_lock));
        KsPrint((2, "KsAcceptCompletionRoutine: Get %xh now signal the event ...\n", pa
    } else {
        /* re-use this child connecton  */
```

```
        child->child.kstc_accepted = FALSE;
        child->child.kstc_busy = FALSE;
        child->kstc_state = ksts_associated;
        spin_unlock(&(child->kstc_lock));
    }
    /* now free the Irp */
    IoFreeIrp(Irp);
    /* drop the refer count of the child */
    ksocknal_put_tconn(child);
    return (STATUS_MORE_PROCESSING_REQUIRED);
}
```

Use case:

```
N/A
```

2.9.3 KsConnectEventHandler

Logic:

```
/*
 * KsConnectEventHandler
 *   Connect event handler event handler, called by the underlying TDI
 *   transport in response to an incoming request to the listening daemon.
 *
 *   it will grab a vacancy backlog from the children tconn list, and
 *   build an acception Irp with it, then transfer the Irp to TDI driver.
 *
 * Arguments:
 *   TdiEventContext:  the tdi connnection object of the listening daemon
 *   ......
 *
 * Return Value:
 *   Nt kernel status code
 *
 * Notes:
 *   N/A
 */
NTSTATUS
KsConnectEventHandler(
    IN PVOID                    TdiEventContext,
    IN LONG                     RemoteAddressLength,
    IN PVOID                    RemoteAddress,
    IN LONG                     UserDataLength,
    IN PVOID                    UserData,
    IN LONG                     OptionsLength,
    IN PVOID                    Options,
```

```
    OUT CONNECTION_CONTEXT *    ConnectionContext,
    OUT PIRP *                  AcceptIrp
    )
{
    ksock_tconn_t *             parent;
    ksock_tconn_t *             child;
    PFILE_OBJECT                FileObject;
    PDEVICE_OBJECT              DeviceObject;
    NTSTATUS                    Status;
    PIRP                        Irp = NULL;
    PTDI_CONNECTION_INFORMATION ConnectionInfo = NULL;
    KsPrint((2,"KsConnectEventHandler: call at Irql: %u\n", KeGetCurrentIrql()));
    parent = (ksock_tconn_t *) TdiEventContext;
    LASSERT(parent->kstc_typee == kstc_listener);
    spin_lock(&(parent->kstc_lock));
    if (parent->kstc_state == ksts_listening) {
        /* allocate a new ConnectionInfo to backup the peer's info */
        ConnectionInfo = (PTDI_CONNECTION_INFORMATION)ExAllocatePoolWithTag(
                NonPagedPool, sizeof(TDI_CONNECTION_INFORMATION) +
                RemoteAddressLength, 'iCsK' );
        if (NULL == ConnectionInfo) {
            Status = STATUS_INSUFFICIENT_RESOURCES;
            cfs_enter_debugger();
            goto errorout;
        }
        /* initializing ConnectionInfo structure ... */
        ConnectionInfo->UserDataLength = UserDataLength;
        ConnectionInfo->UserData = UserData;
        ConnectionInfo->OptionsLength = OptionsLength;
        ConnectionInfo->Options = Options;
        ConnectionInfo->RemoteAddressLength = RemoteAddressLength;
        ConnectionInfo->RemoteAddress = ConnectionInfo + 1;
        RtlCopyMemory(
                ConnectionInfo->RemoteAddress,
                RemoteAddress,
                RemoteAddressLength
                );
        /* get the vacancy listening child tdi connections */
        child = ksocknal_get_vacancy_backlog(parent);
        if (child) {
            spin_lock(&(child->kstc_lock));
            child->child.kstc_info.ConnectionInfo = ConnectionInfo;
            child->child.kstc_info.Remote = ConnectionInfo->RemoteAddress;
            child->kstc_state = kstc_connecting;
            spin_unlock(&(child->kstc_lock));
        } else {
```

```
            KsPrint((2, "KsConnectEventHandler: No enough backlogs: Refsued the connect
            Status = STATUS_INSUFFICIENT_RESOURCES;
            goto errorout;
        }
        FileObject = child->child.kstc_conn.FileObject;
        DeviceObject = IoGetRelatedDeviceObject (FileObject);

        Irp = KsBuildTdiIrp(DeviceObject);
        TdiBuildAccept(
                Irp,
                DeviceObject,
                FileObject,
                KsAcceptCompletionRoutine,
                child,
                NULL,
                NULL
                );
        IoSetNextIrpStackLocation(Irp);
        /* grap the refer of the child tdi connection */
        ksocknal_get_tconn(child);
        Status = STATUS_MORE_PROCESSING_REQUIRED;
        *AcceptIrp = Irp;
        *ConnectionContext = child;
    } else {
        Status = STATUS_CONNECTION_REFUSED;
        goto errorout;
    }
    spin_unlock(&(parent->kstc_lock));
    return Status;
errorout:
    spin_unlock(&(parent->kstc_lock));
    {
        *AcceptIrp = NULL;
        *ConnectionContext = NULL;
        if (ConnectionInfo) {
            ExFreePool(ConnectionInfo);
        }
        if (Irp) {
            IoFreeIrp (Irp);
        }
    }
    return Status;
}
```

Use case:

```
N/A
```

2.9.4 KsDisconnectHelper

Logic:

```
/*
 * KsDisconnectHelper
 *   the routine to be executed in the WorkItem procedure
 *   this routine is to disconnect a tdi connection
 *
 * Arguments:
 *   Workitem:  the context transferred to the workitem
 *
 * Return Value:
 *   N/A
 *
 * Notes:
 *   tconn is already referred in abort_connecton ...
 */
VOID
KsDisconnectHelper(PKS_DISCONNECT_WORKITEM WorkItem)
{
    ksock_tconn_t * tconn = WorkItem->tconn;
    ksocknal_disconnect_tconn(tconn, WorkItem->Flags);
    KeSetEvent(&(WorkItem->Event), 0, FALSE);
    spin_lock(&(tconn->kstc_lock));
    cfs_clear_flag(tconn->kstc_flags, KS_TCONN_DISCONNECT_BUSY);
    spin_unlock(&(tconn->kstc_lock));
    ksocknal_put_tconn(tconn);
}
```

Use case:

```
N/A
```

2.9.5 KsDisconnectEventHandler

Logic:

```
/*
 * KsDisconnectEventHandler
 *   Disconnect event handler event handler, called by the underlying TDI transport
 *   in response to an incoming disconnection notification from a remote node.
 *
 * Arguments:
```

```
 *   ConnectionContext:  tdi connnection object
 *   DisconnectFlags:    specifies the nature of the disconnection
 *   ......
 *
 * Return Value:
 *   Nt kernel status code
 *
 * Notes:
 *   N/A
 */
NTSTATUS
KsDisconnectEventHandler(
    IN PVOID                TdiEventContext,
    IN CONNECTION_CONTEXT   ConnectionContext,
    IN LONG                 DisconnectDataLength,
    IN PVOID                DisconnectData,
    IN LONG                 DisconnectInformationLength,
    IN PVOID                DisconnectInformation,
    IN ULONG                DisconnectFlags
    )
{
    ksock_tconn_t *         tconn;
    NTSTATUS                Status;
    PKS_DISCONNECT_WORKITEM WorkItem;

    tconn = (ksock_tconn_t *)ConnectionContext;
    KsPrint((2, "KsTcpDisconnectEventHandler: called at Irql: %xh\n",
                KeGetCurrentIrql() ));
    KsPrint((2, "tconn = %x DisconnectFlags= %xh\n",
                 tconn, DisconnectFlags));
    ksocknal_get_tconn(tconn);
    spin_lock(&(tconn->kstc_lock));
    WorkItem = &(tconn->kstc_disconnect);
    if (tconn->kstc_state != ksts_connected) {
        Status = STATUS_SUCCESS;
    } else {
        if (cfs_is_flag_set(DisconnectFlags, TDI_DISCONNECT_ABORT)) {
            Status = STATUS_REMOTE_DISCONNECT;
        } else if (cfs_is_flag_set(DisconnectFlags, TDI_DISCONNECT_RELEASE)) {
            Status = STATUS_GRACEFUL_DISCONNECT;
        }
        if (!cfs_is_flag_set(tconn->kstc_flags, KS_TCONN_DISCONNECT_BUSY)) {
            ksocknal_get_tconn(tconn);
            WorkItem->Flags = DisconnectFlags;
            WorkItem->tconn = tconn;
            cfs_set_flag(tconn->kstc_flags, KS_TCONN_DISCONNECT_BUSY);
```

```
            /* queue the workitem to call */
            ExQueueWorkItem(&(WorkItem->WorkItem), DelayedWorkQueue);
        }
    }
    spin_unlock(&(tconn->kstc_lock));
    ksocknal_put_tconn(tconn);
    return  (Status);
}
```

Use case:

```
N/A
```

2.9.6 KsTcpCompletionRoutine

Logic:

```
/*
 * KsTcpCompletionRoutine
 *   the Irp completion routine for TdiBuildSend and TdiBuildReceive ...
 *   We need call the use's own CompletionRoutine if specified. Or
 *   it's a synchronous case, we need signal the event.
 *
 * Arguments:
 *   DeviceObject:  the device object of the transport
 *   Irp:           the Irp is being completed.
 *   Context:       the context we specified when issuing the Irp
 *
 * Return Value:
 *   Nt status code
 *
 * Notes:
 *   N/A
 */
NTSTATUS
KsTcpCompletionRoutine(
    IN PDEVICE_OBJECT   DeviceObject,
    IN PIRP             Irp,
    IN PVOID            Context
    )
{
    if (Context) {
        PKS_TCP_COMPLETION_CONTEXT  CompletionContext;
        CompletionContext = (PKS_TCP_COMPLETION_CONTEXT) Context;
        if (CompletionContext->CompletionRoutine) {
         if ( CompletionContext->bTransmit &&
         InterlockedDecrement(&CompletionContext->ReferCount) != 0 ) {
```

```
                goto errorout;
            }
            //
            // Giving control to user specified CompletionRoutine ...
            //
            CompletionContext->CompletionRoutine(
                    Irp,
                    CompletionContext
                    );
        } else {
            //
            // Signaling  the Event ...
            //
            KeSetEvent(CompletionContext->Event, 0, FALSE);
        }
    } else {
        cfs_enter_debugger();
    }
errorout:
    /* drop the reference count of the tconn object */
    ksocknal_put_tconn(CompletionContext->tconn);
    return STATUS_MORE_PROCESSING_REQUIRED;
}
```

Use case:

```
N/A (Irp completion routine for receiving/sending
```

2.9.7 KsTcpSendCompletionRoutine

Logic:

```
/*
 * KsTcpSendCompletionRoutine
 *   the user specified Irp completion routine for asynchronous
 *   data transmission requests.
 *
 *   It will do th cleanup job of the ksock_tx_t and wake up the
 *   ksocknal scheduler thread
 *
 * Arguments:
 *   Irp:           the Irp is being completed.
 *   Context:       the context we specified when issuing the Irp
 *
 * Return Value:
 *   Nt status code
 *
```

```
 * Notes:
 *   N/A
 */
NTSTATUS
KsTcpSendCompletionRoutine(
    IN PIRP                         Irp,
    IN PKS_TCP_COMPLETION_CONTEXT   Context
    )
{
    NTSTATUS        Status = Irp->IoStatus.Status;
    ULONG           rc = Irp->IoStatus.Information;
    ksock_tx_t *    tx = (ksock_tx_t *)Context->CompletionContext;
    LASSERT(Context->tconn) ;
    if (NT_SUCCESS(Status)) {
        /*
         *  the transmission was done, we need update the tx
         */
        LASSERT(tx->tx_resid >= rc);
        tx->tx_resid -= rc;
        /*
         *  just partial of tx is sent out, we need update
         *  the fields of tx and schedule later transmission.
         */
        if (tx->tx_resid) {
            if (tx->tx_niov >0) {
                /* if there's iov, we need process iov first */
                while (rc > 0 ) {
                    if (rc < tx->tx_iov->iov_len) {
                        /* didn't send whole iov entry... */
                        tx->tx_iov->iov_base =
                            (char *)(tx->tx_iov->iov_base + rc;
                        tx->tx_iov ->iov_len -= rc;
                        rc = 0;
                     } else {
                        /* the whole of iov was sent out */
                        rc -= tx->tx_iov ->iov_len;
                        tx->tx_iov++;
                        tx->tx_niov--;
                     }
                }
            } else {
                /* it must be processing the kiov queues ... */
                while (rc > 0 ) {
                    if (rc < tx->tx_kiov->kiov_len) {
                        /* didn't send whole kiov entry... */
                        tx->tx_kiov->offset += rc;
```

```
                        tx->tx_kiov->kiov_len -= rc;
                        rc = 0;
                    } else {
                        /* whole kiov was sent out */
                        rc -= tx->tx_kiov->kiov_len;
                        tx->tx_kiov++;
                        tx->tx_nkiov--;
                    }
                }
            }
            /*
            * now it's time to re-queue the conns into the
            * scheduler queue and wake the scheduler thread.
            */
            ksocknal_sched_conn(Context->tconn->kstc_conn, TRUE);
        }

    } else {

        /*
         *  for the case that the transmission is ussuccessful,
         *  we need abort the tdi connection, but not destroy it.
         *  the socknal conn will drop the refer count, then the
         *  tdi connection will be freed.
         */
        ksocknal_abort_tconn(tconn);
    }
    /*
     *  it's our duty to free the Irp.
     */
    if (Irp) {
        IoFreeIrp(Irp);
        Irp = NULL;
    }
    return Status;
}
```

Use case:

```
N/A
```

2.9.8 KsTcpReceiveCompletionRoutine

Logic:

```
NTSTATUS
KsTcpReceiveCompletionRoutine(
```

```
    IN PIRP                         Irp,
    IN PKS_TCP_COMPLETION_CONTEXT   Context
    )
{
    NTSTATUS Status = Irp->IoStatus.Status;
    if (NT_SUCCESS(Status)) {
        DbgPrint( "KsTcpReceiveCompletionRoutine: Total %xh bytes.\n",
                  Context->KsTsduMgr->TotalBytes );
   /* re-active the ksocknal connection and wake up the scheduler */
        if (Context->tconn->kstc_conn) {
            ksocknal_sched_conn(Context->tconn->kstc_conn, FALSE);
        }
        /* wake up the thread waiting for the completion of this Irp */
        KeSetEvent(Context->Event, 0, FALSE);
    } else {
        /* un-expected errors occur, we must abort the connection */
        ksocknal_abort_tconn(Context->tconn);
    }
    if (Context) {
        /* drop the refer count of the tconn */
        if (Context->tconn) {
            ksocknal_put_tconn(Context->tconn);
        }
        /* Freeing the Context structure... */
        ExFreePool(Context);
        Context = NULL;
    }
    /* free the Irp */
    if (Irp) {
        IoFreeIrp(Irp);
    }
    return (Status);
}
```

Use case:

```
N/A
```

2.9.9 KsTcpReceiveEventHandler

Logic:

```
/*
 *  Normal receive event handler
 *
 *  It will move data from system Tsdu to our TsduList
 */
```

```
NTSTATUS
KsTcpReceiveEventHandler(
    IN PVOID                TdiEventContext,
    IN CONNECTION_CONTEXT   ConnectionContext,
    IN ULONG                ReceiveFlags,
    IN ULONG                BytesIndicated,
    IN ULONG                BytesAvailable,
    OUT ULONG *             BytesTaken,
    IN PVOID                Tsdu,
    OUT PIRP *              IoRequestPacket
   )
{
    NTSTATUS            Status;
    ksock_tconn_t *     tconn;
    PKS_CHAIN           KsChain;
    PKS_TSDUMGR         KsTsduMgr;
    PKS_TSDU            KsTsdu;
    PKS_TSDU_DAT        KsTsduDat;
    BOOLEAN             bIsExpedited;
    BOOLEAN             bIsCompleteTsdu;
    BOOLEAN             bNewTsdu = FALSE;
    PIRP                Irp = NULL;
    PMDL                Mdl = NULL;
    PFILE_OBJECT        FileObject;
    PDEVICE_OBJECT      DeviceObject;
    ULONG               BytesReceived = 0;
    PKS_TCP_COMPLETION_CONTEXT context = NULL;
    tconn = (ksock_tconn_t *) ConnectionContext;
    ksocknal_get_tconn(tconn);
    /* check whether the whole body of payload is received or not */
    if ( (cfs_is_flag_set(ReceiveFlags, TDI_RECEIVE_ENTIRE_MESSAGE)) &&
         (BytesIndicated == BytesAvailable) ) {
        bIsCompleteTsdu = TRUE;
    } else {
        bIsCompleteTsdu = FALSE;
    }
    bIsExpedited = cfs_is_flag_set(ReceiveFlags, TDI_RECEIVE_EXPEDITED);
    KsPrint((3, "bIsCompleteTsdu = %d bIsExpedited = %d\n",
                bIsCompleteTsdu, bIsExpedited ));
    spin_lock(&(tconn->kstc_lock));
    /*  check whether we are conntected or not listener ¡-
*/
    if ( !((tconn->kstc_state == ksts_connected) &&
           (tconn->kstc_type == kstt_sender ||
            tconn->kstc_type == kstt_child))) {
        *BytesTaken = BytesIndicated;
```

```
    spin_unlock(&(tconn->kstc_lock));
    ksocknal_put_tconn(tconn);
    return (STATUS_SUCCESS);
}
if (tconn->kstc_type == kstt_sender) {
    KsChain = &(tconn->sender.kstc_recv);
} else {
    LASSERT(tconn->kstc_type == kstt_child);
    KsChain = &(tconn->child.kstc_recv);
}
if (bIsExpedited) {
    KsTsduMgr = &(KsChain->Expedited);
} else {
    KsTsduMgr = &(KsChain->Normal);
}
/* retrieve the latest Tsdu buffer form TsduMgr
   list if the list is not empty. */
if (list_empty(&(KsTsduMgr->TsduList))) {
    LASSERT(KsTsduMgr->NumOfTsdu == 0);
    KsTsdu = NULL;
} else {
    LASSERT(KsTsduMgr->NumOfTsdu > 0);
    KsTsdu = list_entry(KsTsduMgr->TsduList.next, KS_TSDU, Link);
    /* if this Tsdu does not contain enough space,
       we need allocate a new one. */
    if ( KS_TSDU_STRU_SIZE(BytesAvailable) >
         KsTsdu->TotalLength - KsTsdu->LastOffset ) {
        KsTsdu = NULL;
    }
}
/* allocate a new Tsdu in case
   we are not statisfied. */
if (NULL == KsTsdu) {
    KsTsdu = KsAllocateKsTsdu();
    if (NULL == KsTsdu) {
        goto errorout;
    } else {
        bNewTsdu = TRUE;
    }
}
/* setup the KS_TSDU_DATA to contain all the messages */
KsTsduDat = (PKS_TSDU_DAT)((PUCHAR)KsTsdu + KsTsdu->LastOffset);
KsTsduDat->TsduType     =  TSDU_TYPE_DAT;
if ( KsTsdu->TotalLength - KsTsdu->LastOffset >=
    KS_TSDU_STRU_SIZE(BytesAvailable) ) {
    BytesReceived = BytesAvailable;
```

```
} else {
    cfs_enter_debugger();
    BytesReceived = KsTsdu->TotalLength - KsTsdu->LastOffset;
}
KsTsduMgr->TotalBytes  +=  BytesReceived;
KsTsduDat->DataLength   =  BytesReceived;
KsTsduDat->TotalLength  =  KS_TSDU_STRU_SIZE(BytesReceived);
KsTsdu->LastOffset += KsTsduDat->TotalLength;
if (bIsCompleteTsdu) {
    /* It's a complete receive, we just move all
       the data from system to our Tsdu */
    RtlMoveMemory(
        &KsTsduDat->Data[0],
        Tsdu,
        BytesReceived
        );
    *BytesTaken = BytesReceived;
    Status = STATUS_SUCCESS;
    if (bNewTsdu) {
        list_add(&(KsTsdu->Link), &(KsTsduMgr->TsduList));
        KsTsduMgr->NumOfTsdu++;
    }
    KeSetEvent(&(KsTsduMgr->Event), 0, FALSE);
    /* re-active the ksocknal connection and wake up the scheduler */
    if (tconn->kstc_conn) {
        ksocknal_sched_conn(tconn->kstc_conn, FALSE);
    }

} else {
    /* there's still data in tdi internal queue, we need issue a new
       Irp to receive all of them. first allocate the tcp context */
    context = ExAllocatePoolWithTag(
                  NonPagedPool,
                  sizeof(KS_TCP_COMPLETION_CONTEXT),
                  'cTsK');
    if (!context) {
        Status = STATUS_INSUFFICIENT_RESOURCES;
        goto errorout;
    }
    /* setup the context */
    RtlZeroMemory(context, sizeof(KS_TCP_COMPLETION_CONTEXT));
    context->tconn             = tconn;
    context->CompletionRoutine = KsTcpReceiveCompletionRoutine;
    context->CompletionContext = KsTsdu;
    context->CompletionContext = KsTsduDat;
    context->KsTsduMgr         = KsTsduMgr;
```

```
    context->Event             = &(KsTsduMgr->Event);

    if (tconn->kstc_type == kstt_sender) {
        FileObject = tconn->sender.kstc_conn.FileObject;
    } else {
        FileObject = tconn->child.kstc_conn.FileObject;
    }
    DeviceObject = IoGetRelatedDeviceObject(FileObject);
    /* build new tdi Irp and setup it. */
    Irp = KsBuildTdiIrp(DeviceObject);
    if (NULL == Irp) {
        goto errorout;
    }
    Status = KsLockUserBuffer(
                &(KsTsduDat->Data[0]),
                BytesReceived,
                IoModifyAccess,
                &Mdl
                );
    if (!NT_SUCCESS(Status)) {
        goto errorout;
    }
    TdiBuildReceive(
        Irp,
        DeviceObject,
        FileObject,
        KsTcpCompletionRoutine,
        context,
        Mdl,
        ReceiveFlags & (TDI_RECEIVE_NORMAL | TDI_RECEIVE_EXPEDITED),
        BytesReceived
      );

    IoSetNextIrpStackLocation(Irp);
    /* return the newly built Irp to transport driver,
       it will process it to receive all the data */
    *IoRequestPacket = Irp;
    *BytesTaken = 0;
    if (bNewTsdu) {
        list_add(&(KsTsduMgr->TsduList), &(KsTsdu->Link));
        KsTsduMgr->NumOfTsdu++;
    }
    ksocknal_get_tconn(tconn);
    Status = STATUS_MORE_PROCESSING_REQUIRED;
}
spin_unlock(&(tconn->kstc_lock));
```

```
    ksocknal_put_tconn(tconn);
    return (Status);
errorout:
    spin_unlock(&(tconn->kstc_lock));
    if (bNewTsdu && (KsTsdu != NULL)) {
        KsFreeKsTsdu(KsTsdu);
    }
    if (Mdl) {
        KsReleaseMdl(Mdl);
    }
    if (Irp) {
        IoFreeIrp(Irp);
    }
    if (context) {
        ExFreePool(context);
    }
    ksocknal_abort_tconn(tconn);
    ksocknal_put_tconn(tconn);
    *BytesTaken = BytesAvailable;
    Status = STATUS_SUCCESS;
    return (Status);
}
```

Use case:

```
N/A
```

2.9.10 KsTcpReceiveExpeditedEventHandler

Logic:

```
/*
 *  Expedited receive event handler
 */
NTSTATUS
KsTcpReceiveExpeditedEventHandler(
    IN PVOID                TdiEventContext,
    IN CONNECTION_CONTEXT   ConnectionContext,
    IN ULONG                ReceiveFlags,
    IN ULONG                BytesIndicated,
    IN ULONG                BytesAvailable,
    OUT ULONG *             BytesTaken,
    IN PVOID                Tsdu,
    OUT PIRP *              IoRequestPacket
    )
{
    return KsTcpReceiveEventHandler(
```

```
                TdiEventContext,
                ConnectionContext,
                ReceiveFlags | TDI_RECEIVE_EXPEDITED,
                BytesIndicated,
                BytesAvailable,
                BytesTaken,
                Tsdu,
                IoRequestPacket
                );
}
```

Use case:

```
N/A
```

2.9.11 KsTcpChainedReceiveEventHandler

Logic:

```
/*
 *  Bulk receive event handler
 *
 *  It will queue all the system Tsdus to our TsduList.
 *  Then later ksocknal_recv_mdl will release them.
 */
NTSTATUS
KsTcpChainedReceiveEventHandler (
    IN PVOID TdiEventContext,       // the event context
    IN CONNECTION_CONTEXT ConnectionContext,
    IN ULONG ReceiveFlags,
    IN ULONG ReceiveLength,
    IN ULONG StartingOffset,        // offset of start of client data in TSDU
    IN PMDL  Tsdu,                  // TSDU data chain
    IN PVOID TsduDescriptor         // for call to TdiReturnChainedReceives
    )
{
    NTSTATUS            Status;
    Ksock_tconn_t *     tconn;
    PKS_CHAIN           KsChain;
    PKS_TSDUMGR         KsTsduMgr;
    PKS_TSDU            KsTsdu;
    PKS_TSDU_MDL        KsTsduMdl;
    BOOLEAN             bIsExpedited;
    BOOLEAN             bNewTsdu = FALSE;
    tconn = (ksock_tconn_t *) ConnectionContext;
    bIsExpedited = cfs_is_flag_set(ReceiveFlags, TDI_RECEIVE_EXPEDITED);
    ksocknal_get_tconn(tconn);
```

```
    spin_lock(&(tconn->kstc_lock));
    /* check whether we are conntected or not listener ¡-
*/
    if ( !((tconn->kstc_state == ksts_connected) &&
         (tconn->kstc_type == kstt_sender ||
          tconn->kstc_type == kstt_child))) {
        *BytesTaken = BytesIndicated;
        spin_unlock(&(tconn->kstc_lock));
        ksocknal_put_tconn(tconn);
        return (STATUS_SUCCESS);
    }
    /* retrieve the latest Tsdu buffer form TsduMgr list.
       Just set NULL if the list is empty. */
    if (tconn->kstc_type == kstt_sender) {
        KsChain = &(tconn->sender.kstc_recv);
    } else {
        LASSERT(tconn->kstc_type == kstt_child);
        KsChain = &(tconn->child.kstc_recv);
    }
    if (bIsExpedited) {
        KsTsduMgr = &(KsChain->Expedited);
    } else {
        KsTsduMgr = &(KsChain->Normal);
    }
    if (list_empty(&(KsTsduMgr->TsduList))) {
        LASSERT(KsTsduMgr->NumOfTsdu == 0);
        KsTsdu = NULL;
    } else {
        LASSERT(KsTsduMgr->NumOfTsdu > 0);
        KsTsdu = list_entry(KsTsduMgr->TsduList.next, KS_TSDU, Link);
        LASSERT(KsTsdu->Magic == KS_TSDU_MAGIC);
        if (sizeof(KS_TSDU_MDL) > KsTsdu->TotalLength - KsTsdu->LastOffset) {
            KsTsdu = NULL;
        }
    }
    /* if there's no Tsdu or the free size is not enough from this
       KS_TSDU_MDL structure. We need re-allocate a new Tsdu.  */
    if (NULL == KsTsdu) {
        KsTsdu = KsAllocateKsTsdu();
        if (NULL == KsTsdu) {
            goto errorout;
        } else {
            bNewTsdu = TRUE;
        }
    }
    /* Just queue the KS_TSDU_MDL to the Tsdu buffer */
```

```
        KsTsduMdl = (PKS_TSDU_MDL)((PUCHAR)KsTsdu + KsTsdu->LastOffset);
        KsTsduMdl->TsduType     =  TSDU_TYPE_MDL;
        KsTsduMdl->DataLength   =  ReceiveLength;
        KsTsduMdl->StartOffset  =  StartingOffset;
        KsTsduMdl->Mdl          =  Tsdu;
        KsTsduMdl->Descriptor   =  TsduDescriptor;
        KsTsdu->LastOffset += sizeof(KS_TSDU_MDL);
        KsTsduMgr->TotalBytes += ReceiveLength;
        KsPrint((2, "KsTcpChainedReceiveEventHandler: Total %xh bytes.\n",
                    KsTsduMgr->TotalBytes ));
        /* re-active the ksocknal connection and wake up the scheduler */
        if (tconn->kstc_conn) {
            ksocknal_sched_conn(tconn->kstc_conn, FALSE);
        }
        Status = STATUS_PENDING;
        /* attach it to the TsduMgr list if the Tsdu is newly created. */
        if (bNewTsdu) {
            list_add(&(KsTsdu->Link), &(KsTsduMgr->TsduList));
            KsTsduMgr->NumOfTsdu++;
        }
        /* wake up the threads waiing in sock_recvmsg */
        KeSetEvent(&(KsTsduMgr->Event), 0, FALSE);
        spin_unlock(&(tconn->kstc_lock));
        ksocknal_put_tconn(tconn);
        /* Return STATUS_PENDING to system because we are still
           owning the MDL resources. sock_recvmsg is expected
           to free the MDL resources. */
        return (Status);
    errorout:
        spin_unlock(&(tconn->kstc_lock));
        if (bNewTsdu && (KsTsdu != NULL)) {
            KsFreeKsTsdu(KsTsdu);
        }
        /* abort the tdi connection */
        ksocknal_abort_tconn(tconn);
        ksocknal_put_tconn(tconn);
        Status = STATUS_SUCCESS;
        return (Status);
    }
```

Use case:

```
    N/A
```

2.9.12 KsTcpChainedReceiveExpeditedEventHandler

Logic:

```
/*
 *  Expedited & Bulk receive event handler
 */
NTSTATUS
KsTcpChainedReceiveExpeditedEventHandler (
    IN PVOID                TdiEventContext,       // the event context
    IN CONNECTION_CONTEXT   ConnectionContext,
    IN ULONG                ReceiveFlags,
    IN ULONG                ReceiveLength,
    IN ULONG                StartingOffset,        // offset of start of client data in
    IN PMDL                 Tsdu,                  // TSDU data chain
    IN PVOID                TsduDescriptor         // for call to TdiReturnChainedRecei
    )
{
    return KsTcpChainedReceiveEventHandler(
                TdiEventContext,
                ConnectionContext,
                ReceiveFlags | TDI_RECEIVE_EXPEDITED,
                ReceiveLength,
                StartingOffset,
                Tsdu,
                TsduDescriptor );
}
```

2.10 ksocknal routines

2.10.1 ksocknal_cmd

```
int
ksocknal_cmd(struct portals_cfg *pcfg, void * private)
{
        ......
        case NAL_CMD_START_DAEMON: {
                /* start a daemon thread to listen on the port */
                rc = ksocknal_start_daemon( pcfg->pcfg_msic, /*port*/
                                            pcfg->pcfg_count /*backlog*/
                                            );
            break;
        }
        case NAL_CMD_STOP_DAEMON: {
            /* stop the daemon thread listening on the specified port */
                ksocknal_stop_daemon(pcfg->pcfg_misc);
                rc = 0;
```

```
            break;
        }
        case NAL_CMD_CONNECT_PEER: {
                /* connect to remote peer */
                rc = ksocknal_build_conn( 0 , 0,  /* local ip and port*/
                                          pcfg->pcfg_id,    /* IP */
                                          pcfg->pcfg_misc,  /* port */
                                          );
                break;
        }
        ......
}
```

2.10.2 ksocknal_built_conn

```
/*
 * ksocknal_build_conn
 *   build a socknal conntion
 *
 * Arguments:
 *   laddr: local ip address
 *   lport: local port number
 *   paddr: peer's ip address
 *   pport: peer's port number
 *
 * Return Value:
 *   ksocknal return code ...
 *
 * Notes:
 *   N/A
 */
int
ksocknal_build_conn(__u32 laddr, __u16 lport, __u32 paddr, __u16 pport)
{
    ksock_tconn_t * tconn;
    /* create the tdi connecion structure */
    tconn = ksocknal_create_tconn();
    if (!tconn) {
        return -ENOMEM;
    }
    /* initialize the tdi sender connection */
    ksocknal_init_sender(tconn);
    /* bind the local ip address with the tconn */
    rc = ksocknal_bind_tconn(tconn, NULL, laddr, lport);
    if (rc < 0) {
        ksocknal_free_tconn(tconn);
```

```
        return rc;
    }
    /* connect it to the remote peer */
    rc = ksocknal_build_tconn(tconn, paddr, pport);
    if (rc < 0) {
    } else {
        /* create the ksocknal conn structure for the tonn */
        rc = ksocknal_create_conn(NULL, tconn, SOCK_CONN_ANY)
    }
    /* drop the ownship of tconn ... */
    ksocknal_put_tconn(tconn);
    return rc;
}
```

2.10.3 ksocknal_create_conn

```
int
ksocknal_create_conn (ksock_route_t *route, ksock_tconn_t *tconn, int type)
{
     ......
     rc = ksocknal_setup_tconn(tconn);
     if (rc != 0)
             return rc;
     ......
     tconn->kstc_conn = conn;
     ......
     ksocknal_get_tconn(tconn);
     ......
}
```

2.10.4 ksocknal_recv_iov

```
int
ksocknal_recv_iov (ksock_conn_t *conn)
{
     ......
             ksock_mdl_t *   mdl;
        /* NB we can't trust socket ops to either consume our iovs
         * or leave them alone, so we only receive 1 frag at a time. */
        LASSERT (conn->ksnc_rx_niov > 0);
        /* lock the whole tx iovs into a single mdl chain */
        mdl = ksocknal_lock_iovs(iov, conn->ksnc_rx_niov, &size);
        if (!mdl) {
            rc = -ENOMEM;
            return (rc);
        }
```

```
        LASSERT (size <= conn->ksnc_rx_nob_wanted);
        /* try to request data for the whole mdl chain */
        rc = ksocknal_recv_mdl (conn, mdl, size, MSG_DONTWAIT);
        /* release the chained mdl */
        ksocknal_release_mdl(mdl);
        if (rc <= 0)
                return (rc);
        ......
}
```

2.10.5 ksocknal_recv_kiov

```
int
ksocknal_recv_kiov (ksock_conn_t *conn)
{
        ......
        ksock_mdl_t *   mdl;
        /* NB we can't trust socket ops to either consume our iovs
         * or leave them alone, so we only receive 1 frag at a time. */
        LASSERT (conn->ksnc_rx_nkiov > 0);
        /* lock the whole tx kiovs into a single mdl chain */
        mdl = ksocknal_lock_kiovs(kiov, conn->ksnc_rx_nkiov, &size);
        if (!mdl) {
            rc = -ENMEM;
            return (rc);
        }

        LASSERT (size <= conn->ksnc_rx_nob_wanted);
        /* try to request data for the whole mdl chain */
        rc = ksocknal_recv_mdl (conn, mdl, size, MSG_DONTWAIT);
        /* release the chained mdl */
        ksocknal_release_mdl(mdl);
        if (rc <= 0)
                return (rc);
        ......
}
```

2.10.6 ksocknal_send_iov

```
int
ksocknal_send_iov (ksock_conn_t *conn, ksock_tx_t *tx)
{
        ......
        {
                /* lock the whole tx iovs into a single mdl chain */
```

```
                mdl = ksocknal_lock_iovs(tx->tx_iov, tx->tx_niov, &len);
                if (mdl) {
                    /* send the total mdl chain */
                    rc = ksocknal_send_mdl(
                       conn->kstc_tconn, tx, mdl, len,
                       mmore ? (MSG_DONTWAIT | MSG_MORE) :MSG_DONTWAIT);
                } else {
                    rc = -ENOMEM;
                }
        }
        ......
}
```

2.10.7 ksocknal_send_kiov

```
int
ksocknal_send_kiov (ksock_conn_t *conn, ksock_tx_t *tx)
{
        ......
                {
                /* lock the whole tx kiovs into a single mdl chain */
                mdl = ksocknal_lock_kiovs(tx->tx_kiov, tx->tx_nkiov, &len);
                if (mdl) {
                    /* send the total mdl chain */
                    rc = ksocknal_send_mdl(
                                conn->kstc_tconn, tx, mdl, leng,
                                more ? (MSG_DONTWAIT | MSG_MORE) : MSG_DONTWAIT);
                } else {
                    rc = -ENOMEM;
                }
        }
        ......
}
```

2.10.8 ksocknal_process_transmit

```
int
ksocknal_process_transmit (ksock_conn_t *conn, ksock_tx_t *tx)
{
        unsigned long  flags;
        int            rc;

        if (tx->tx_resid == 0) {
                ksocknal_tx_launched (tx);
                return (0);
        }
```

```
        rc = ksocknal_transmit (conn, tx);
        CDEBUG (D_NET, ("send(%d) %d\n", tx->tx_resid, rc));
        if (rc == -EAGAIN)
                return (rc);
        if (tx->tx_resid == 0) {
                /* Sent everything OK */
                LASSERT (rc == 0);
                ksocknal_tx_launched (tx);
                return (0);
        }

        ......
}
```

2.10.9 ksocknal_get_tconn_addrs

```
/*
 * ksocknal_get_tconn_addrs
 *   Query the ip address / port infromation of the local
 *   or the peer of a connection.
 *
 * Arguments:
 *   conn: the socknal connection object
 *
 * Return Value:
 *   int: the socknal return code. ALWAYS returns 0.
 *
 * Notes:
 *   N/A
 */
int
ksocknal_get_conn_addrs (ksock_conn_t *conn)
{
        PTRANSPORT_ADDRESS tdi_addr = NULL;
        PTDI_ADDRESS_IP    ip_addr  = NULL;
        LASSERT( conn->ksnc_tconn->kstc_type == kstt_sender ||
                conn->ksnc_tconn->kstc_type == kstt_child );
        /*
         *   Get peer's address
         */
        if (conn->ksnc_tconn->kstc_type == kstt_sender) {
            tdi_addr = (conn->ksnc_tconn->sender.kstc_info.Remote);
        } else {
            tdi_addr = (conn->ksnc_tconn->child.kstc_info.Remote);
        }
        ip_addr = (PTDI_ADDRESS_IP)(&(tdi_addr->Address[0].Address));
```

```
        /* Didn't need the {get,put}connsock dance to deref ksnc_sock... */
           LASSERT (!conn->ksnc_closing);
           conn->ksnc_ipaddr = ntohl (ip_addr->in_addr);
           conn->ksnc_port   = ntohs (ip_addr->sin_port);
           /*
            *   Get local's address
            */
           tdi_addr = &(conn->ksnc_tconn->kstc_addr.Tdi);
           ip_addr = (PTDI_ADDRESS_IP)(&(tdi_addr->Address[0].Address));
           conn->ksnc_myipaddr = ntohl (ip_addr->in_addr);
           return 0;
    }
```

2.10.10 ksocknal_init_tdi_data

```
    /*
     * ksocknal_init_tdi_data
     *   initialize the global data in ksockal_data
     *
     * Arguments:
     *   N/A
     *
     * Return Value:
     *   int: ksocknal error code
     *
     * Notes:
     *   N/A
     */
    int
    ksocknal_init_tdi_data()
    {
        int rc = 0;
        /* initialize tconn related globals */

        spin_lock_init(&ksocknal_data.ksnd_tconn_lock);
        CFS_INIT_LIST_HEAD(&ksocknal_data.ksnd_tconns);
        cfs_init_event(&ksocknal_data.ksnd_tconn_exit, TRUE, FALSE);
        ksocknal_data.ksnd_tconn_slab = cfs_mem_cache_create(
            "tcon", sizeof(ksock_tconn_t) , 0, 0);
        if (!ksocknal_data.ksnd_tconn_slab) {
            rc = -ENOMEM;
            goto errorout;
        }
        /* initialize tsdu related globals */

        spin_lock_init(&ksocknal_data.ksnd_tsdu_lock);
```

```
    CFS_INIT_LIST_HEAD(&ksocknal_data.ksnd_freetsdus);
    ksocknal_data.ksnd_tsdu_size = 0x10000; /* 64k */
    ksocknal_data.ksnd_tsdu_slab = cfs_mem_cache_create(
        "tsdu", ksocknal_data.ksnd_tsdu_size, 0, 0);
    if (!ksocknal_data.ksnd_tsdu_slab) {
        rc = -ENOMEM;
        cfs_mem_cache_destroy(ksocknal_data.ksnd_tconn_slab);
        ksocknal_data.ksnd_tconn_slab = NULL;
        goto errorout;
    }
    /* initialize daemon related globals */
    spin_lock_init(&ksocknal_data.ksnd_daemon_lock);
    CFS_INIT_LIST_HEAD(&ksocknal_data.ksnd_daemons);
    cfs_init_event(&ksocknal_data.ksnd_daemon_exit, TRUE, FALSE);
errorout:
    return rc;
}
```

2.10.11 ksocknal_fini_tdi_data

```
/*
 * ksocknal_fini_tdi_data
 *   finalize the global data in ksockal_data
 *
 * Arguments:
 *   N/A
 *
 * Return Value:
 *   int: ksocknal error code
 *
 * Notes:
 *   N/A
 */
void
ksocknal_fini_tdi_data()
{
    PKS_TSDU            KsTsdu = NULL;
    struct list_head *  list   = NULL;
    /* we need wait until all the tconn are freed */
    spin_lock(&(ksocknal_data.ksnd_tconn_lock));
    if (list_empty(&(ksocknal_data.ksnd_tconns))) {
        cfs_wake_event(&ksocknal_data.ksnd_tconn_exit);
    }
    spin_unlock(&(ksocknal_data.ksnd_tconn_lock));
    /* now wait on the tconn exit event */
    cfs_wait_event(&ksocknal_data.ksnd_tconn_exit, 0);
```

```
    /* it's safe to delete the tconn slab ... */
    cfs_mem_cache_destroy(ksocknal_data.ksnd_tconn_slab);
    ksocknal_data.ksnd_tconn_slab = NULL;
    /* clean up all the tsud buffers in the free list */
    spin_lock(&(ksocknal_data.ksnd_tsdu_lock));
    list_for_each (list, &ksocknal_data.ksnd_freetsdus) {
        KsTsdu = list_entry (list, KS_TSDU, Link);
        cfs_mem_cache_free(
                ksocknal_data.ksnd_tsdu_slab,
                KsTsdu );
    }
    spin_unlock(&(ksocknal_data.ksnd_tsdu_lock));
    /* it's safe to delete the tsdu slab ... */
    cfs_mem_cache_destroy(ksocknal_data.ksnd_tsdu_slab);
    ksocknal_data.ksnd_tsdu_slab = NULL;
    /* good! it's smoothing to do the cleaning up...*/
}
```

2.10.12 ksocknal_api_startup

```
int
ksocknal_api_startup (nal_t *nal, ptl_pid_t requested_pid,
                      ptl_ni_limits_t *requested_limits,
                      ptl_ni_limits_t *actual_limits)
{
        ......
        ksocknal_data.ksnd_init = SOCKNAL_INIT_LIB; // flag lib_init() called
        /* initialize the tdinal specified strctures in ksocknal_data */
        rc = ksocknal_init_tdi_data();
        if (rc != 0) {
            CERROR (("Can't intialize tdinal specified data ... (rc = %d)\n", rc));
            ksocknal_api_shutdown (nal);
            return  rc;
        }
        ksocknal_data.ksnd_init = SOCKNAL_INIT_TDI;
        ......
}
```

2.10.13 ksocknal_api_shutdown

```
void
ksocknal_api_shutdown (nal_t *nal)
{
        ksock_sched_t *sched;
        int            i;
        if (nal->nal_refct != 0) {
```

```
                /* This module got the first ref */
                PORTAL_MODULE_UNUSE;
                return;
        }
        CDEBUG(D_MALLOC, ("before NAL cleanup: kmem %d\n",
               atomic_read (&portal_kmemory)));
        LASSERT(nal == &ksocknal_api);
        switch (ksocknal_data.ksnd_init) {
        default:
                LASSERT (0);
        case SOCKNAL_INIT_ALL:
                libcfs_nal_cmd_unregister(SOCKNAL);
                /* stop all the daemon threads */
                ksocknal_stop_all_daemons();
                ksocknal_data.ksnd_init = SOCKNAL_INIT_LIB;
                /* fall through */
                ......
        case SOCKNAL_INIT_LIB:
                ......
                /* Tell lib we've stopped calling into her. */
                lib_fini(&ksocknal_lib);
                ksocknal_data.ksnd_init = SOCKNAL_INIT_TDI;
                /* fall through */
        case SOCKNAL_INIT_TDI:
                /* de-initialize the tdinal global data */
                ksocknal_fini_tdi_data();
                ksocknal_data.ksnd_init = SOCKNAL_INIT_DATA;
                /* fall through */
                ......
}
```

2.10.14 ksocknal_tcon_write

```
int
ksocknal_tconn_write (struct socket *sock, void *buffer, int nob)
{
        int           rc;
        ksock_mdl_t * mdl;
        int           offset = 0;
        while (nob > offset) {
                /* lock the user buffer */
                rc = ksocknal_lock_buffer(
                        (char *)buffer + offset,
                        nob - offset,
                        IoReadAccess, &mdl );
                if (rc < 0) {
```

```
                        return (rc);
                }
                /* send out the whole mdl */
                rc = ksocknal_send_mdl(
                        tconn, NULL, mdl,
                        nob - offset,
                        0 );
                /* release the chained mdl */
                ksocknal_relese_mdl(mdl);
                if (rc > 0) {
                        offset += rc;
                }
        }

        return (0);
}
```

2.10.15 ksocknal_tconn_read

```
int
ksocknal_tconn_read (ksock_tconn_t * tconn, void *buffer, int nob)
{
        int           rc;
        ksock_mdl_t * mdl;
        int           offset = 0;
        while (nob > offset) {
                /* lock the user buffer */
                rc = ksocknal_lock_buffer(
                        (char *)buffer + offset,
                        nob - offset,
                        IoWriteAccess, &mdl );
                if (rc < 0) {
                        return (rc);
                }
                /* recv the requested buffer */
                rc = ksocknal_recv_mdl(
                        tconn, NULL, mdl,
                        nob - offset,
                        0 );
                /* release the chained mdl */
                ksocknal_relese_mdl(mdl);
                if (rc > 0) {
                        offset += rc;
                }
        }
        return (0);
```

```
}
```

3 Summary