T-Engine Forum Specification

TEF040-S215-01.00.00/en
Febrary 24, 2004

T-Engine Device Driver Interface Library Specification

Number: TEF040-S215-01.00.00/en
Title: T-Engine Device Driver Interface Library Specification
Status: [ ] Working Draft, [ ] Final Draft for Voting, [X] Standard
Date: January 28, 2004
Febrary 24, 2004 Voted

Contents

1. Introduction

This specification consists of a specification of the T-Engine device driver interface library, a resource available for developing device drivers based on the T-Kernel/SM device management specification.

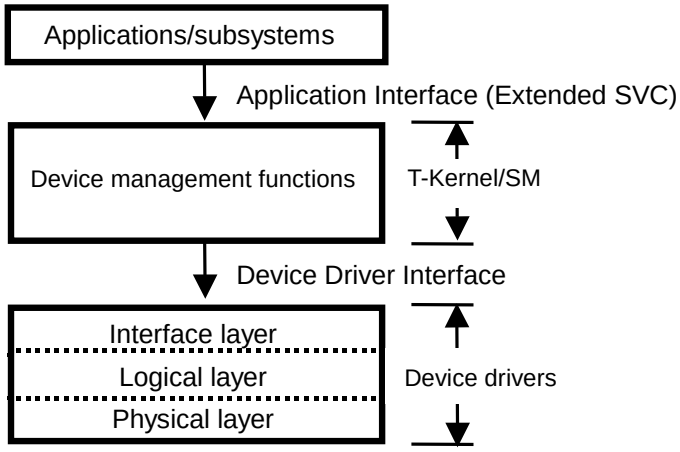
This library enables convenient development of device drivers designed to run on T-Kernel.

This library is not included in the T-Kernel specification.

2. Device Driver Interface Library Specification

2.1 Overview

The device drivers consist of an interface layer, logical layer, and physical layer. This structure makes it easy to maintain the drivers and to port them between different hardware platforms. The interface layer handles the interface with the T-Kernel device management function. The logical layer is common to each device and does not depend on the hardware controller. The physical layer provides actual control of the hardware controllers.
Of these, the interface layer is able to achieve a large degree of commonality across most device drivers. The role of the device driver interface layer is to reduce the burden on device driver developers and to prevent object code from becoming bloated, by bringing together the aspects that can be standardized.



(from the T-Kernel Specification)

2.2 Types of Device Driver Interface Layers

Currently two kinds of driver interfaces are available.

- Simple device driver interface layer
  For simple devices that can perform all processing immediately without queuing requests (e.g., RTC and other register-based devices).

- General device driver interface layer
  For ordinary devices that handle requests in the order of receipt, especially devices that must interrupt processing (e.g., RS-232C).

  A major difference between the two is that a general driver interface must create a task to execute an operation in response to IO, whereas a simple driver interface is able to handle all IO operations by function calls.

  Also, when a general drive interface is used, functions can be defined for interrupting a device

IO operation, whereas operations cannot be interrupted when a simple driver interface is used (since this would obviously involve going into an indefinite WAIT state, which is not possible with a simple driver interface).

These driver interface layers form libraries designed solely to facilitate the creation of device drivers; their use is therefore not strictly required. When the abovementioned driver models cannot be used due to the nature or functions of your device, you may establish your particular drivers independently.

2.3 Common Matters Applying to All Device Driver Interface Layers

The following points need to be noted when using functions defined for a driver interface.

- Except where otherwise noted, driver interface functions cannot be called from the task-independent portion or while dispatching or interrupts are disabled.

- The processing functions defined in a driver interface must perform their processing quickly and must not go into an indefinite WAIT state.

- A processing function defined in a driver interface runs as a quasi-task in the context of the requesting task. For this reason, care must be taken regarding the following.

  - When task priority or the like is changed, it must be restored to the original state before a function returns.
  - It is always necessary to be aware of the task ID of the task running a processing function.
  - Stack memory use must be taken into account.

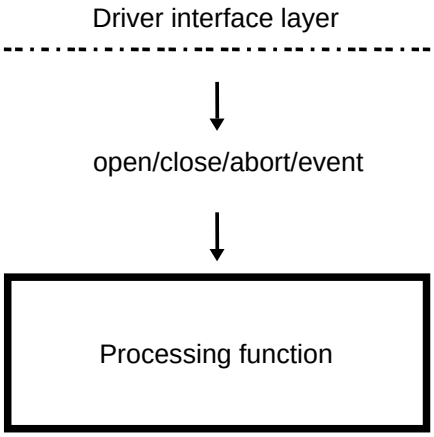
  Since, however, exclusion control is performed for calling of processing functions, these functions are never requested to run at the same time (the one exception is the abort processing function of the general driver interface, which can be called at any time).

- A driver interface operates under the T-Kernel device management functions, which means the structures and other T-Kernel device management definitions can be used.

When referring to the structures, values and other matters defined in the T-Kernel specification, this manual simply indicates (T-Kernel) without giving further details. Refer as necessary to the section on device management functions in the T-Kernel specification.

2.4 Simple Device Driver Interface Layer <driver/sdrvif.h>

The functions defined here are used in creating device drivers for very simple devices that do not cause a wait of indefinite length.
A device driver is implemented making use of functions that handle requests, as in the figure below.



2.4.1 Define device

```
ER SDefDevice(SDefDev *ddev, T_IDEV *idev, SDI *sdi)
```

ddev: Pointer to device driver registration information

idev: Pointer to location of device initialization information (T-Kernel)

sdi: Pointer to location of driver interface access handler

Return code = 0 : Normal completion
< 0 : Error

Defines a device based on the ddev registration information. The device initialization information is returned in idev, except when idev = NULL is designated. At the same time the driver interface access handle needed for simple driver interface operations is returned in sdi.

The SDI and SDefDev structures are as indicated below.

```
typedef struct SimpleDriverInterface * SDI;

typedef struct {
        VP      exinf;              /* extended information */
        UB      devnm[L_DEVNM+1];   /* physical device name (T-Kernel) */
        ATR     drvatr;             /* driver attributes (T-Kernel) */
        ATR     devatr;             /* device attributes (T-Kernel) */
```

```
        INT     nsub;           /* subunit count (T-Kernel) */
        INT     blksz;          /* block size (T-Kernel) */

        ER   (*open )(ID devid, UINT omode, SDI);
        ER   (*close)(ID devid, UINT option, SDI);
        INT (*read )(ID devid, INT start, INT size, VP buf, SDI);
        INT (*write)(ID devid, INT start, INT size, VP buf, SDI);
        INT (*event)(INT evttyp, VP evtinf, SDI);
} SDefDev;
```

Any desired information may be set in exinf, for reference by SDI exinf() described later.

See the T-Kernel specification section on device management functions for details of the values set in devnm, drvatr, devatr, nsub, and blksz.

The following value can be designated in drvatr. (T-Kernel)

```
#define TDA_OPENREQ 0x0001 /* open and close with each request */
```

The functions open, close, read, write, and event are processing functions in the device driver interface corresponding to the following T-Kernel function calls.

Device Driver	T-Kernel
Function defined in open	tk_opn_dev()
Function defined in close	tk_cls_dev()
Function defined in read	tk_rea_dev()
Function defined in write	tk_wri_dev()
Function defined in event	Executed at suspend and resume, or when an event is generated by the USB Manager or PC Card Manager, etc.

The return code for the functions defined in read/write indicates the size of the IO result or error. The memory area designated in buf passed to read/write must already have been checked by the driver interface (ChkSpace).

2.4.2 Update device

```
ER SRedefDevice(SDefDev *ddev, SDI sdi)
```

ddev: Pointer to device driver registration information

sdi: Driver interface access handle

Return code = 0 : Normal completion
< 0 : Error

Updates SDI device registration information in accord with the information in ddev. The device name (devnm) cannot be changed.
The physical device ID does not change with an update.

2.4.3 Delete device

```
ER SDelDevice(SDI sdi)
```

sdi: Driver interface access handle

Return code = 0 : Normal completion
< 0 : Error

Deletes the registration of the device driver having the driver interface access handle designated in sdi.

2.4.4 Get information

```
ID SDI_devid(SDI sdi)
VP SDI_exinf(SDI sdi)
const SDefDev* SDI_ddev(SDI sdi)
```

sdi: Driver interface access handle

Return code SDI_devid(): Physical device ID (T-Kernel)
SDI_exinf():The value set in exinf in the currently registered device driver information.
SDI_ddev() : Pointer to the current registered device driver information

Gets various information.

- SDI_devid() is used to get the physical device ID required when the USB Manager or PC Card Manager executes an event handler function.

- SDI_exinf() gets only the value of exinf out of the various device driver information

registered by SDefDevice(), SRedefDevice(), etc.

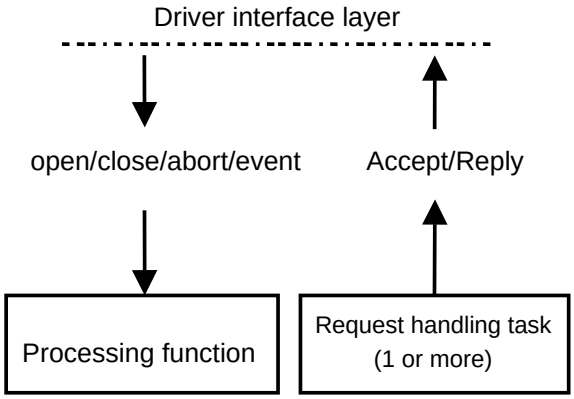
- SDI_exinf() gets a pointer to the device driver information registered by SDefDevice(), SRedefDevice(), etc.

These functions can be called from the task-independent portion or while dispatching or interrupts are disabled.

2.5 General Device Driver Interface Layer <driver/gdrvif.h>

These functions are used in creating device drivers for ordinary devices that must process requests in the order of their receipt.

A device driver queues tasks that request processing, handling the requests asynchronously.



2.5.1 Define device

```
ER GDefDevice(GDefDev *ddev, T_IDEV *idev, GDI *gdi)
```

ddev: Pointer to device driver registration information

idev: Pointer to location of device initialization information (T-Kernel)

gdi: Pointer to location of driver interface access handler

Return code = 0: Normal completion

< 0: Error

Defines a device based on the ddev registration information. The device initialization information is returned in idev, except when idev = NULL is designated. At the same time the driver interface access handle needed for simple driver interface operations is returned in gdi.

The GDI and GdefDev structures are as follows.

```
typedef struct GeneralDriverInterface * GDI;

typedef struct {
        VP      exinf;              /* extended information */
        UB      devnm[L_DEVNM+1];   /* physical device name (T-Kernel) */
        UH      maxreqq;            /* maximum queued requests */
        ATR     drvatr;             /* driver attributes (T-Kernel) */
        ATR     devatr;             /* device attributes (T-Kernel) */
        INT     nsub;               /* subunit count (T-Kernel) */
        INT     blksz;              /* block size (T-Kernel) */

        ER      (*open )(ID devid, UINT omode, GDI);
        ER      (*close)(ID devid, UINT option, GDI);
        ER      (*abort)(T_DEVREQ *devreq, GDI);
        INT     (*event)(INT evttyp, VP evtinf, GDI);
} GDefDev;
```

Any desired information may be set in exinf, for reference by GDI_exinf() described later.

maxreqq indicates the maximum number of requests that can be queued. This must be set to at least 1 or above.

See the T-Kernel specification section on device management functions for details of the values set in devnm, drvatr, devatr, nsub, and blksz.

The following values can be set in drvatr. (T-Kernel)

```
#define TDA_OPENREQ     0x0001  /* open and close with each request */
#define TDA_LOCKREQ     0x8000  /* address space lock required */
#define TDA_LIMITEDREQ  0x4000  /* restricts the number of queued requests,
                                   depending on request type */
```

* When TDA LOCKREQ is designated, the IO buffer (T_DEVREQ.buf) area is locked (made resident) by the driver interface.

* When the TDA_LIMITEDREQ is specified, the number of queued requests is restricted to about half of the maximum number of queued requests (maxreqq), depending on request type (TDC_READ, TDC_WRITE).
  In this case, a value of two or more must be specified for the maxreqq.
  When the TDA_LIMITEDREQ is not specified, requests are queued until the queue is full, regardless of request type.
  The TDA_LIMITEDREQ is used specifically when you wish to prevent a queue from becoming filled up with certain requests (thus blocking others) in a device that processes reading and writing concurrently in an asynchronous manner.

The functions open, close, abort, and event are processing functions in the device driver interface corresponding to the following T-Kernel function calls.

Device Driver	T-Kernel
Function defined in open	tk_opn_dev()
Function defined in close	tk_cls_dev()
Function defined in abort	Executed when it is necessary to stop a de-vice IO operation by tk_cls_dev() or other function.
Function defined in event	Executed at suspend and resume, or when an event is generated by the USB Manager or PC Card Manager, etc.

The abort function differs in the following ways from other processing functions.

- It can be called while another function is executing (although the abort function itself cannot be called more than once at a time).
- The general device driver interface functions that can be used are limited to GDI_devid(), GDI_exinf(), GDI_ddev(), and GDI_SendCmd().

The current IO request stopped by the abort function is indicated in devreq.

2.5.2 Update device

```
ER GRedefDevice(GDefDev *ddev, GDI gdi)
```

ddev: Pointer to device driver registration information

gdi : Driver interface access handle

Return code = 0: Normal completion

< 0: Error

Updates GDI device registration information in accord with the information in ddev. The device name (devnm) and maximum queued requests (maxreqq) cannot be changed.

* The physical device ID does not change with an update.
* Requests that are queued for acceptance and not yet accepted are all aborted.

2.5.3 Delete device

```
ER GDelDevice(GDI gdi)
```

gdi : Driver interface access handle

Return code = 0: Normal completion

< 0: Error

Deletes the registration of the device driver having the driver interface access handle designated in gdi.

2.5.4 Accept IO request

```
INT GDI_Accept(T_DEVREQ **devreq, INT acpptn, TMO tmout, GDI gdi);
```

devreq: Pointer to the location of a pointer to the device driver IO request packet
acpptn: Request type
tmout: Timeout duration (ms)
gdi: Driver interface access handle

Return code = 0 : Received request pattern (differs with request type)
< 0 : Error

Fetches one request from the queue of requests waiting for acceptance. If there are no queued requests, this function waits for one to arrive.

acpptn designates, in OR format, either the type of the request (TDC READ/TDC WRITE) or the value obtained in DEVREQ ACPPTN() by the user command pattern.

```
/* cmd = TDC_READ || TDC_WRITE || user command (16 to 23) */
#define DEVREQ_ACPPTN(cmd)    (1 << (cmd))

#define DRP_READ       DEVREQ_ACPPTN(TDC_READ)
#define DRP_WRITE      DEVREQ_ACPPTN(TDC_WRITE)
#define DRP_NORMREQ    (DRP_READ|DRP_WRITE) /* ordinary request */
#define DRP_USERCMD    0x00ff0000           /* user command */
```

Requests for device-specific data and requests for attribute data can be accepted separately. (Acceptance waiting extension function)

```
#define DRP_ADSEL      0x00000100  /* specify device-specific/attribute
                                      data separately */
#define DRP_DREAD      (DRP_ADSEL | DEVREQ_ACPPTN(TDC_READ) )
#define DRP_DWRITE     (DRP_ADSEL | DEVREQ_ACPPTN(TDC_WRITE) )
#define DRP_AREAD      (DRP_ADSEL | DEVREQ_ACPPTN(TDC_READ  + 8) )
#define DRP_AWRITE     (DRP_ADSEL | DEVREQ_ACPPTN(TDC_WRITE + 8) )

#define DRP_REQMASK    (DRP_ADSEL | DRP_NORMAREQ | (DRP_NORMREQ << 8))

DRP_DREAD       Read device-specific data
DRP_DWRITE      Write device-specific data
```

```
DRP_AREAD       Read attribute data
DRP_AWRITE      Write attribute data
```

These are specified in combination using OR.

* DRP_DREAD |DRP_AREAD is equivalent to DRP_READ.
* DRP_DWRITE|DRP_AWRITE is equivalent to DRP_WRITE.
* You cannot use these separate specifications for the device-specific/attribute data and DRP_READ, DRP_WRITE at the same time in combination.

An ordinary request (TDC_READ/TDC_WRITE) is accepted with higher priority than a user command, but in some cases an ordinary request and user command will be accepted at the same time. The values set in the return code and in *devreq depend on the received request, as follows.

- Ordinary request
  A pattern indicating the type of request accepted is put in the return code. The accepted request is returned in *devreq.

- User command
  A pattern indicating the accepted user command is put in the return code. In case multiple user commands are queued, all user commands designated in acpptn are accepted at once and returned in an OR pattern. NULL is returned in *devreq.

- When ordinary request and user command are accepted at the same time
  Both the accepted ordinary request and user command are put in the return code in an OR pattern. The accepted ordinary request is returned in *devreq.

- Timeout or error
  The error code is put in the return code. This is E TMOUT in the case of a timeout. Any value can be set in *devreq.

* The accepted request pattern is returned to the return code in the form specified at acpptn. In other words, when the acceptance waiting extension function is used, the request for unique data and the request for attribute data are shown separately within the accepted request pattern. In this case, DRP_ADSEL is also specified.

The request timeout interval is designated in tmout in milliseconds. TMO_POL and TMO_FEVR may also be designated.

The value of exinf of an accepted request (T_DEVREQ) must not be changed.

Checking of buf space (ChkSpace) is handled at the driver interface, but task space switching can be performed as necessary using tk set_tsp(devreq->tskspc) or the like.

A reply to a user command (GDI_Reply) is not necessary.

Normally one request is accepted and processed, and the result is returned before going on to the next request. It is also possible, however, to accept multiple requests and process them

concurrently.
When multiple requests are processed at the same time, this can be done by having multiple request processing tasks each issue GDI_Accept() and process concurrently, or by having one processing task issue GDI_Accept() multiple times and perform concurrent processing. The order in which processing results are returned does not have to be the same as the order in which requests were accepted.

2.5.5 Reply on IO request completion

```
void GDI_Reply(T_DEVREQ*, GDI gdi);
```

devreq:	Pointer to the device driver IO request packet for which IO processing was completed.
gdi:	Driver interface access handle
Return code :	none

Returns the result of processing a request fetched by GDI_Accept().
The task processing GDI_Accept() and that processing GDI_Reply() do not have to be the same task.

2.5.6 Send user command to IO request task

```
ER GDI_SendCmd(INT cmd, GDI gdi)
```

gdi:	Driver interface access handle
cmd:	user command (only values between 16 and 23 may be set)
Return code	= 0 : Normal completion < 0 : Error

Sends the user command designated in cmd.

After a user command is issued, it is accepted by GDI_Accept(). If there are multiple user commands waiting for acceptance by GDI_Accept(), each queued command will be a unique command; that is, even if the same command is issued more than once, it is queued only once, and GDI_Accept() is called only once for that command.

Once GDI_SendCmd() is called, it returns immediately without waiting for acceptance by GDI_Accept().

User commands are generally used for the purpose of clearing at any point queued requests waiting for acceptance by GDI_Accept().

2.5.7 Get information

```
ID GDI_devid(GDI gdi)
VP GDI_exinf(GDI gdi)
const GDefDev* GDI_ddev(GDI gdi)
```

gdi: Driver interface access handle

Return code GDI_devid() : Physical device ID (T-Kernel)
GDI_exinf() : The value set in exinf in the currently registered device driver information
GDI_ddev(): Pointer to the current registered device driver information

Gets various information.

- GDI_devid() is used to get the physical device ID required when the USB Manager or PC Card Manager executes an event handler function.
- GDI_exinf() gets only the value of exinf out of the various device driver information registered by GDefDevice(), GRedefDevice(), etc.
- GDI_exinf() gets a pointer to the device driver information registered by GDefDevice(), GRedefDevice(), etc.

These functions can be called from the task-independent portion or while dispatching or interrupts are disabled.