# T-Engine Forum
# Specification

November 21, 2012

## T-Engine Standard Device Driver Specifications

TEF040-S202-01.00.00/en T-Engine Device Driver Interface (2): Serial Communication
TEF040-S203-01.00.00/en T-Engine Device Driver Interface (3): USB
TEF040-S204-01.00.00/en T-Engine Device Driver Interface (4): NIC
TEF040-S205-01.00.00/en T-Engine Device Driver Interface (5): PCMCIA
TEF040-S206-01.00.00/en T-Engine Device Driver Interface (6): System Disk
TEF040-S207-01.00.00/en T-Engine Device Driver Interface (7): eTRON SIM
TEF040-S208-01.00.00/en T-Engine Device Driver Interface (8): Clock
TEF040-S209-01.00.01/en T-Engine Device Driver Interface (9): Keyboard/Pointing Device
TEF040-S211-01.00.00/en T-Engine Device Driver Interface (11): Console
TEF040-S214-01.00.00/en T-Engine Device Driver Interface (14): Screen (Display)

Number: TEF040-S202-01.00.00/en, TEF040-S203-01.00.00/en, TEF040-S204-01.00.00/en
TEF040-S205-01.00.00/en, TEF040-S206-01.00.00/en, TEF040-S207-01.00.00/en
TEF040-S208-01.00.00/en, TEF040-S209-01.00.01/en, TEF040-S211-01.00.00/en
TEF040-S214-01.00.00/en

Title: T-Engine Standard Device Driver Specifications
Status: [ ] Working Draft, [ ] Final Draft for Voting, [X] Standard
Date: August 18, 2003 First Edited
October 27, 2003 Updated to 01.A0.00
January 14, 2004 Voted
November 21, 2012 Updated to 01.00.01

# Contents

# 1. Introduction

This manual presents the specifications for T-Engine device drivers created based on the T-Kernel System Manager (T-Kernel/SM) device management specification.

# 2. RS-232C Driver

TEF040-S202-01.00.00/en

## 2.1 Applicable Devices

- This driver applies to RS-232C communications devices.

## 2.2 Device Names

- The device names are "rsa", "rsb", "rsc", and "rsd".
- The device name and corresponding RS-232C port differs with the hardware system.

## 2.3 Device-specific Functions

- Data input/output on the RS-232C port as well as various control functions
- Only asynchronous communication supported
- PC Card support

## 2.4 Attribute Data

The following attribute data is supported.

```
R    Read-only
W    Write-only
RW   Read/write enabled

/* RS data numbers */
typedef enum {
    /* Common attributes */
    DN_PCMCIAINFO = TDN_PCMCIAINFO,
    /* Device-specific attributes */
    DN_RSMODE           = -100,   /* communication mode       */
    DN_RSFLOW           = -101,   /* flow control             */
    DN_RSSTAT           = -102,   /* line status              */
    DN_RSBREAK          = -103,   /* break                    */
    DN_RSSNDTMO         = -104,   /* send timeout             */
    DN_RSRCVTMO         = -105,   /* receive timeout          */
    DN_RSADDIN          = -150,   /* additional features (not used) */
    /* Attributes for special IBM keyboard features (not used)  */
    DN_IBMKB_KBID       = -200,   /* keyboard ID (not used)   */
    /* Attributes for special touch panel features (not used)   */
    DN_TP_CALIBSTS      = -200,   /* calibration state (not used)*/
    DN_TP_CALIBPAR      = -201, /*calibration parameter(not used)*/
```

```
    /* Model-specific attributes */
    DN_RS16450              = -300    /* hardware setup   */
} RSDataNo;
```

DN_PCMCIAINFO: Get PC Card information (R)

data: PCMCIAInfo

```
typedef struct {
    UB      major;          /* specification version (major)      */
    UB      minor;          /* specification version (minor)      */
    UB      info[40];       /* product information        */
} PCMCIAInfo;
```

Reads the product information part of the card attribute data from the currently mounted PC Card.
info is an ASCII string terminated by '¥0'.
If no PC Card is mounted, an error (E_NOMDA) is returned.
If the media is not a PC Card, information cannot be read and an error (E_PAR) is returned.

DN_RSMODE: Get/set serial communication mode (RW)

data: RsMode

```
typedef struct {
    UW      parity: 2;      /* 0: none, 1: odd, 2: even, 3: --        */
    UW      datalen: 2;     /* 0: 5 bits, 1: 6 bits, 2: 7 bits, 3: 8 bits */
    UW      stopbits: 2;    /* 0: 1 bit, 1: 1.5 bits, 2: 2 bits, 3: --    */
    UW      rsv: 2;         /* reserved       */
    UW      baud: 24;       /* baud rate      */
} RsMode;
```

parity: 0: none, 1: odd, 2: even, 3: --
datalen: 0: 5 bits, 1: 6 bits, 1: 7 bits, 3: 8 bits
stopbits: 0: 1 bit, 1: 1.5 bits, 2: 2 bits, 3: --
baud: baud rate (bps)

Sets/gets the serial communication mode settings.
An error occurs if an unsupported setting is made.
Writing the above data initializes the communication environment as follows.

- Clears the receive buffer
- Clears the send buffer
- No send timeout (= 0)
- No receive timeout (= 0)
- No flow control

DN_RSFLOW: Get/set flow control (RW)

data: RsFlow

```
typedef struct {
    UW      rsv: 26;        /* reserved          */
    UW      rcvxoff: 1;     /* forced XOFF state change */
    UW      csflow: 1;      /* CTS control       */
    UW      rsflow: 1;      /* RTS control       */
    UW      xonany: 1;      /* XON for any character    */
    UW      sxflow: 1;      /* send XON/XOFF control    */
    UW      rxflow: 1;      /* receive XON/XOFF control */
} RsFlow;
```

rcvxoff: Indicates that sending is stopped due to receipt of XOFF.
The state can be overridden by a write operation.

csflow: Send flow control by CS signal.
1: No sending when CS signal is OFF.
0: Send regardless of CS signal.
* 1 is set by default.

rsflow: Receive flow control by RS signal.
When the receive buffer is close to becoming full, the RS signal is set to OFF, causing the peer to stop sending.
When there is available buffer space, the RS signal goes back ON.

xonany: When sending is stopped due to receipt of XOFF, receipt of any character (not just XON) clears the XOFF state. (Valid only when sxflow = 1.)

sxflow: Enable send flow control by XON/XOFF.
After XOFF is received, sending cannot continue until receipt of XON.

rxflow: Enable receive control by XON/XOFF.
When the receive buffer is nearly full, XOFF is sent; when there is available buffer space, XON is sent.

Sets flow control or gets flow control settings.

DN_RSSTAT: Line status (R)

data: RsStat

```
typedef struct {
#if BIGENDIAN
    UW      rsv1: 20;
    UW      BE: 1;      /* Recv Buffer Overflow Error*/
    UW      FE: 1;      /* Framing Error             */
    UW      OE: 1;      /* Overrun Error             */
    UW      PE: 1;      /* Parity Error              */
    UW      rsv2: 2;
    UW      XF: 1;      /* Recv XOFF                 */
    UW      BD: 1;      /* Break Detect              */
```

```
        UW      DR: 1;  /* Dataset Ready (DSR)              */
        UW      CD: 1;  /* Carrier Detect (DCD)             */
        UW      CS: 1;  /* Clear to Send (CTS)              */
        UW      CI: 1;  /* Calling Indicator(RI)            */
    #else
        UW      CI: 1;  /* Calling Indicator(RI)            */
        UW      CS: 1;  /* Clear to Send (CTS)              */
        UW      CD: 1;  /* Carrier Detect (DCD)             */
        UW      DR: 1;  /* Dataset Ready (DSR)              */
        UW      BD: 1;  /* Break Detect                     */
        UW      XF: 1;  /* Recv XOFF                        */
        UW      rsv2: 2;
        UW      PE: 1;  /* Parity Error                     */
        UW      OE: 1;  /* Overrun Error                    */
        UW      FE: 1;  /* Framing Error                    */
        UW      BE: 1;  /* Recv Buffer Overflow Error*/
        UW      rsv1: 20;
    #endif
    } RsStat;
```

Indicates the RS port signal state.

| | |
|---|---|
| FE, OE, PE: | Indicate error occurrence status; cleared on read. |
| BD, CD, CS, CI: | Indicate current (input) signal state. |
| XF: | Same as RsFlow.rcvxoff |

DN_RSBREAK: Send break (W)
data: UW (no operation if 0)

Sends a break signal for the designated number of milliseconds, causing a wait for the designated number of milliseconds until sending is complete.

DN_RSSNDTMO: Get/set send timeout (RW)
data: UW (no timeout if 0)

Sets the send timeout in milliseconds.
Timeout occurs if send ready is not achieved within the designated time.
This timeout applies not to the entire send time of a write operation, but to the interval from sending of the previous byte to the next byte.

DN_RSRCVTMO: Get/set receive timeout (RW)
data: UW (no timeout if 0)

Sets the receive timeout in milliseconds.
Timeout occurs if no data is received within the designated time.

This timeout applies not to the entire receive time of a read operation, but to the interval from receipt of the previous byte to the next bytes.

DN_RS16450: Get/set hardware settings (for 16450) (RW)
data: RsHwConf_16450

```
typedef struct {
    UW      iobase;         /* start address of 16450 IO space      */
    UW      iostep;         /* interval between IO addresses of individual
                               16450 registers     */
    INTVEC  intvec;         /* 16450 interrupt level        */
} RsHwConf_16450;
```

iobase: Start address of 16450 IO space
iostep: Interval between IO addresses of individual 16450 registers
intvec: 16450 interrupt vector number

Device use is stopped at iostep = 0. In this case the other field values are invalid. An error (E_NOMDA) occurs if an attempt is made to access other attribute data or a request is made for sending or receiving. Normally the driver itself sets the default automatically.
For a PC Card, the state remains out-of-use (iostep = 0) until the card is mounted.
When the PC Card is inserted, the driver automatically performs the hardware setup.
If an expansion board or the like is used and the settings are changed for that board, the setup can be modified by writing this attribute data. The invoking entity is responsible for setting data correctly. The behavior is not guaranteed in case settings are made incorrectly.

- This is hardware-dependent attribute data, so only certain hardware models are supported.
- Some devices are read-only and cannot be written to (implementation dependent).

## 2.5 Device-specific Data

Data number:

Fixed at 0

Data count:

Read/write byte count

Actual data read/write is performed at the RS-232C port.

When data count = 0:

R: Received bytes (data held in receive buffer) returned as the effective data size.
This is returned when the designated number of bytes have been read.

W: 0 is returned.
This is returned after the designated number of bytes have been written.

## 2.6 Event Notification

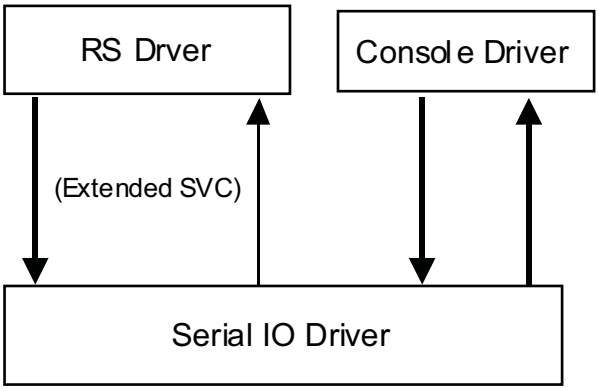
None

## 2.7 Role of the RS Driver

The RS driver provides interface functions with device management. Actual IO operations on the serial port are performed by the serial IO driver.

Thus, most actual RS driver operations are performed through corresponding serial IO driver function calls.

The RS driver handles processing such as PC Card setup.




## 2.8 Serial IO Driver

The serial IO driver is a low-level IO driver that provides the following functions by dedicated extended SVC.

ER serial_in(W port, B* buf, W len, W *alen, W tmout)

Reads len bytes of data from the port designated by port, returning the actually read byte count in *alen.

If len < = 0, the received byte count is returned in *alen without actually reading any data.

The tmout argument designates timeout in milliseconds as follows.

| | | |
|---|---|---|
| | > 0: | Wait until len bytes are read, error occurs, or timeout. |
| | = 0: | Read up to len bytes of received data; no waiting. |
| | < 0: | No timeout |
| Function value | = 0: | Normal |
| | < 0: | Error occurred. (The number of data bytes returned in alen are read.) |
| | | * Returns E_PAR or E_IO + (error information). |

ER serial_out(W port, B* buf, W len, W *alen, W tmout)

Writes len bytes of data to the port designated by port, returning the written byte count in *alen.

When len < = 0, no operation is performed.

The tmout argument designates timeout in milliseconds as follows.

| | | |
|---|---|---|
| | > 0: | Wait until len bytes are written, error occurs, or timeout. |
| | = 0: | Cannot be designated (error) |
| | < 0: | No timeout<br>Wait until the write operation is complete or an error occurs. |
| Function value | = 0: | Normal |
| | < 0: | Error occurred. (The number of data bytes returned in alen are written.)<br>* Returns E_PAR or E_IO + (error information). |

ER serial_ctl(W port, W kind, UW *arg)

Performs various operations on the port designated by port.

```
typedef enum {
        RS_ABORT        = 0,
        RS_SUSPEND      = -200,
        RS_RESUME       = -201,
        RS_RCVBUFSZ     = -202,
        RS_LINECTL      = -203,

        RS_EXTFUNC      =-9999    /* Special functions outside specification */
} SerialControlNo;
```

| <kind> | | <arg> |
|---|---|---|
| RS_ABORT | -- | Abort (WAIT cleared) |
| RS_SUSPEND | -- | Go to SUSPEND state |
| RS_RESUME | -- | Return from SUSPEND state |

• When entering SUSPEND state, there must not have been a request other than RS_RESUME (including serial_in/out). The behavior is undefined if the request is not RS_RESUME.

| | | |
|---|---|---|
| DN_RSMODE | RsMode | Set serial communication mode |
| - DN_RSMODE | RsMode | Get serial communication mode |
| DN_RSFLOW | RsFlow | Set flow control |

| | | |
|---|---|---|
| - DN_RSFLOW | RsFlow | Get flow control |
| - DN_RSSTAT | RsStat | Get line status |
| DN_RSBREAK | UW (ms) | Send break signal (cause wait) |
| RS_RCVBUFSZ | UW (bytes) | Set receive buffer size |
| - RS_RCVBUFSZ | UW (bytes) | Get receive buffer size |

• The minimum receive buffer size is 256 bytes and the default is 2 Kbytes.

| | | |
|---|---|---|
| RS_LINECTL | UW | Set control signal ON/OFF |
| RSCTL_DTR | 0x00000001 | DTR signal |
| RSCTL_RTS | 0x00000002 | RTS signal |
| RSCTL_SET | 0x00000000 | Set all signals |
| RSCTL_ON | 0xc0000000 | Designated signal ON |
| RSCTL_OFF | 0x80000000 | Designated signal OFF |

(RSCTL_SET/RSCTL_ON/RSCTL_OFF) | [RSCTL_DTR] | [RSCTL_RTS]

(Example)

RSCTL_SET | RSCTL_DTR DTR = ON, RTS = OFF
RSCTL_ON | RSCTL_DTR DTR = ON, RTS unchanged
RSCTL_OFF | RSCTL_DTR DTR = OFF, RTS unchanged

| | | |
|---|---|---|
| DN_RS16450 | RsHwConf_16450 | Set hardware configuration |
| - DN_RS16450 | RsHwConf_16450 | Get hardware configuration |

• In the case of a PC Card, after the RS driver maps IO ports and interrupt levels, the IO addresses and interrupt level information are set in the serial IO driver.

Function value =0: Normal
< 0: Error occurred
* Returns E_PAR or E_IO + (error information).

The serial IO driver is automatically started at system boot, at which time the hardware configuration of each port is set automatically (default settings) and hardware is initialized.

A PC Card port starts in out-of-use state.

Port numbers (port) are assigned as sequential integers from 0 to the number of ports less 1.

The number of ports is fixed and is implementation dependent.

The mapping of port numbers to device names is decided by the RS driver.

The RS driver performs a suitable serial_ctl() call to check for the existence of ports then, registers them as devices.

Error information is as follows.

```
typedef struct {
#if BIGENDIAN
    UW      ErrorClass: 16;     /* Error class = EC_IO          */
    UW      rsv1: 2;
    UW      Aborted:1;          /* aborted              */
    UW      Timout:1;           /* timed out            */
    /* Same as RsStat from this point       */
    UW      BE: 1;              /* Recv Buffer Overflow Error*/
    UW      FE: 1;              /* Framing Error        */
    UW      OE: 1;              /* Overrun Error        */
    UW      PE: 1;              /* Parity Error                 */
    UW      rsv2: 2;
    UW      XF: 1;              /* Recv XOFF                    */
    UW      BD: 1;              /* Break Detect         */
    UW      DR: 1;              /* Dataset Ready (DSR) */
    UW      CD: 1;              /* Carrier Detect (DCD) */
    UW      CS: 1;              /* Clear to Send (CTS) */
    UW      CI: 1;              /* Calling Indicator(RI)        */
#else
    UW      CI: 1;              /* Calling Indicator(RI)        */
    UW      CS: 1;              /* Clear to Send (CTS) */
    UW      CD: 1;              /* Carrier Detect (DCD) */
    UW      DR: 1;              /* Dataset Ready (DSR) */
    UW      BD: 1;              /* Break Detect         */
    UW      XF: 1;              /* Recv XOFF                    */
    UW      rsv2: 2;
    UW      PE: 1;              /* Parity Error         */
    UW      OE: 1;              /* Overrun Error        */
    UW      FE: 1;              /* Framing Error        */
    UW      BE: 1;              /* Recv Buffer Overflow Error*/
    /* Same as RsStat up to this point      */
    UW      Timout:1;           /* timed out            */
    UW      Aborted:1;          /* aborted              */
    UW      rsv1: 2;
    UW      ErrorClass: 16;     /* Error class= EC_IO           */
#endif
} RsError;
```

## 2.9 Error Codes

All line errors are E_IO, with error details set in RsError returned by the serial IO driver.

For other errors, see the section on device management functions in the T-Kernel specification.

## 2.10 T-Engine/SH7727 Related Information (Reference)

### 2.10.1 Applicable devices

For T-Engine/SH7727, the device names and corresponding RS-232C ports are as follows.

"rsa" On-board 16550 debug port (ch.B)
"rsb" PC Card
"rsc" (not used)
"rsd" (not used)

### 2.10.2 H8 power supply controller IO driver

A function for serial IO with the H8 power supply controller on the T-Engine/SH7727 is provided as an additional serial IO driver function using a dedicated extended SVC.

INT H8Read(W reg, W len)

Reads a value from the register designated by reg number. len indicates the register size, with 1 meaning an 8-bit (1-byte) register and 2 a 16-bit (2-byte) register width. (Values other than 1 and 2 must not be designated in len.)

This processing causes a wait until data exchange with the register is complete.

Function value> =0:Value read from register
< 0:Error occurred

Examples:

```
#define KEYSR     0x62
#define KBITPR    0x64

sts = H8Read(KEYSR, 1);
dat = H8Read(KBITPR, 2);
```

ER H8Write(W reg, W len, W dat)

Writes a value (dat) to the register designated by reg number.

len indicates the register size, with 1 meaning an 8-bit (1-byte) register and 2 a 16-bit (2-byte) register width. (Values other than 1 and 2 must not be designated in len.)

This processing causes a wait until data exchange with the register is complete.

Function value = 0: Write complete
< 0: Error occurred

Examples:

```
#define LEDR    0xa0
#define XAPDR   0x2c

err = WriteH8(XAPDR, 2, 0);
WriteH8(LEDR, 1, 0x5a);
```

ER H8Reset(void)

Reinitializes the 16550 used for communication with the H8 power supply controller.
Note that initialization of the 16550 does not entail initialization of the H8 power supply controller.

This process causes a wait until initialization of the 16550 used for communication with the H8 power supply controller is complete.

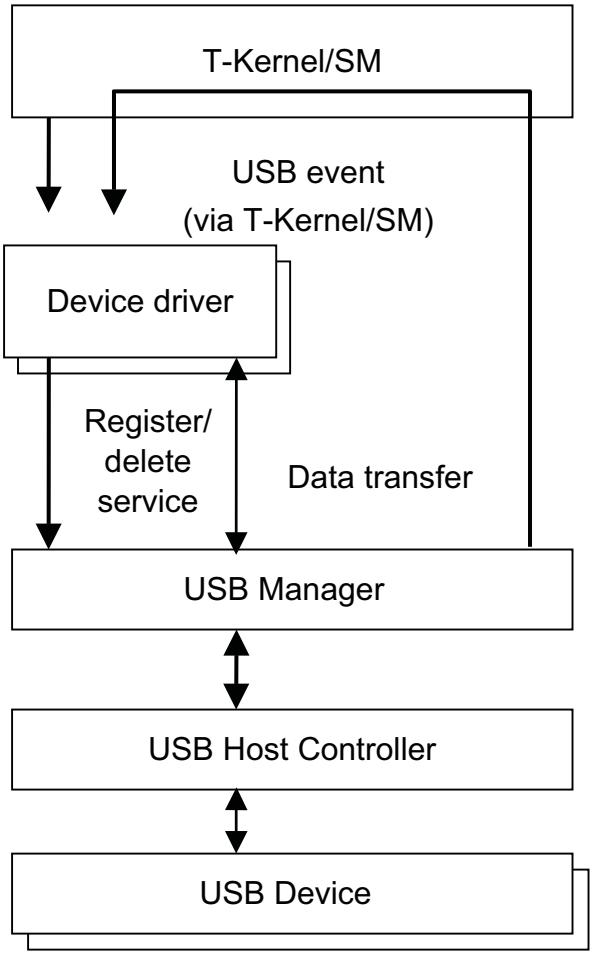
Function value =0:16550 initialization complete
<0: Error occurred

# 3. USB Manager

TEF040-S203-01.00.00/en

## 3.1 Role

The USB Manager is a driver corresponding to the USB Host Controller. It provides the device driver for a USB device with a standard means for communicating with the USB device, independent of the controller or other hardware. Because it is a manager, no device name is given.



## 3.2 USB Manager Functions

The USB Manager has the following functions.

- Notifies of device connection and disconnection

  Detects device connection or disconnection, notifying the associated driver using the T-Kernel/SM event notification function.

- Executes common device processing

  When a device is connected, the USB Manager performs the following tasks to enable access by the associated driver: reset processing, address setting, getting descriptors, and setting configuration as necessary.

  When a device is disconnected, the USB Manager performs the processing required to stop access to the device.

- Associates drivers with devices

  When a device is connected, the USB Manager queries the registered drivers to find a river or interface for the connected device. The USB Manager then associates the driver with the device.

- Communicates with devices

  Functions are provided for controlled / bulk / interrupted transfer (Isochronous transfer is not supported).

- Provides various services to drivers

  Gets descriptors.

  Gets device connection information.

  Provides other services.

## 3.3 Functions Required by Device Drivers

A device driver for a USB device must have the following functions in addition to the usual driver functions.

- Functions for device connection and disconnection

  A USB device may be connected or disconnected at any time. A driver must be able to handle these events properly under any circumstances.

- Functions for device matching

  The driver must be able to read the device and interface descriptors, etc. in order to determine whether it is the driver matching a connected device or interface.

- Device initialization functions

  Device initialization itself is performed by the USB Manager, but the individual drivers must handle setup enabling use of USB device functions required by the driver.

- Communication with devices

  The driver must communicate with a USB device in accordance with the descriptor contents based on the device and interface class.

## 3.4 Limitations

The USB Manager is subject to the following limitations.

- When a device has more than one configuration, only the first configuration (configuration index 0) can be used.

- A total of 31 devices, 64 interfaces, and 64 endpoints can be used. The USB Host Controller is also counted as one device.

- The maximum data size that can be exchanged along with a request in the case of usbRequestDevice() is 4088 (4096 - 8) bytes.

## 3.5 Data Definitions (usb.h)

```
/* Event type */
#define USB_ATTACH      1       /* device connected        */
#define USB_DETACH      2       /* device disconnected     */

/* Response code */
#define USB_NONE        0       /* not a matching device   */
#define USB_OWN         1       /* matching device         */

/* USB device request (USB standards) */
typedef struct {
        UB              bmRequestType;  /* sets the object of the request */
        UB              bRequest;       /* request code     */
        UH              wValue;         /* value to be set  */
        UH              wIndex;         /* designates string index, etc. */
        UH              wLength; /* transfer size       */
} usbDeviceRequest;

/* bRequest: Default request codes (USB standards) */
#define USB_GET_STATUS                  0
#define USB_CLEAR_FEATURE               1
#define USB_SET_FEATURE                 3
#define USB_SET_ADDRESS                 5
#define USB_GET_DESCRIPTOR              6
#define USB_SET_DESCRIPTOR              7
#define USB_GET_CONFIGURATION           8
#define USB_SET_CONFIGURATION           9
#define USB_GET_INTERFACE               10
#define USB_SET_INTERFACE               11
#define USB_SYNCH_FRAME                 12

/* bmRequestType */
#define bmR_DEVICE              0x00
#define bmR_INTERFACE           0x01
#define bmR_ENDPOINT            0x02
#define bmR_OTHER               0x03
```

```
#define bmR_STANDARD        0x00
#define bmR_CLASS           0x20
#define bmR_VENDOR          0x40
#define bmR_OUT             0x00
#define bmR_IN              0x80

/* bDescriptorType: Descriptor types (USB standards) */
#define USB_DEVICE          1
#define USB_CONFIGURATION   2
#define USB_STRING          3
#define USB_INTERFACE       4
#define USB_ENDPOIN         5

/* USB Device Descriptor (USB standards) */
typedef struct {
        UB      bLength;            /* descriptor length        */
        UB      bDescriptorType;    /* Device Descriptor (1)    */
        UH      bcdUSB;             /* USB standards version    */
        UB      bDeviceClass;       /* Device Class             */
        UB      bDeviceSubClass;    /* Device Subclass          */
        UB      bDeviceProtocol;    /* Device Protocol          */
        UB      bMaxPacketSize0;    /* pipe#0 PacketSize        */
        UH      idVendor;           /* vendor ID (USB-IF)       */
        UH      idProduct;          /* product ID               */
        UH      bcdDevice;          /* product version          */
        UB      iManufacturer;      /* string index (Mfg.)      */
        UB      iProduct;           /* string index (Prod.)     */
        UB      iSerialNumber;      /* string index (Ser#)      */
        UB      bNumConfigurations; /* number of configurations */
} usbDeviceDescriptor;

/* USB Configuration Descriptor (USB standards) */
typedef struct {
        UB      bLength;            /* descriptor length                  */
        UB      bDescriptorType;    /* Configuration Descriptor (2)       */
        UH      wTotalLength;       /* Configuration + other descriptor size */
        UB      bNumInterfaces;     /* number of interfaces               */
        UB      bConfigurationValue;/* ID of this configuration           */
        UB      iConfiguration;     /* string index (configuration)       */
```

```
        UB      bmAttributes;           /*attributes concerning power supply,etc.*/
        UB      MaxPower;               /* power consumption (×2 mA) */
} usbConfigurationDescriptor;

/* USB Interface Descriptor (USB standards) */
typedef struct {
        UB      bLength;                /* descriptor length        */
        UB      bDescriptorType;        /* Interface Descriptor (4) */
        UB      bInterfaceNumber;       /* ID of this interface     */
        UB      bAlternateSetting;      /* alternate setting ID     */
        UB      bNumEndpoints;          /* number of endpoints       */
        UB      bInterfaceClass;        /* Interface Class          */
        UB      bInterfaceSubClass;     /* Interface Subclass       */
        UB      bInterfaceProtocol;     /* Interface Protocol       */
        UB      iInterface;             /* string index (interface) */
} usbInterfaceDescriptor;

/* USB Endpoint Descriptor (USB standards) */
typedef struct {
        UB      bLength;                /* descriptor length        */
        UB      bDescriptorType;        /* Endpoint Descriptor (5)  */
        UB      bEndpointAddress;       /* endpoint address         */
        UB      bmAttributes;           /* transfer format (Ctrl/Iso..) */
        UH      wMaxPacketSize;         /* packet size              */
        UB      bInterval;              /* Iso/Int transfer interval (ms) */
} usbEndpointDescriptor;

/* Endpoint Descriptor bmAttributes definitions (USB standards) */
#define         USB_CONTROL     0
#define         USB_ISOCHRONOUS 1
#define         USB_BULK        2
#define         USB_INTERRUPT   3

/* USB String Descriptor (USB standards) */
typedef struct {
        UB      bLength;                /* descriptor length        */
        UB      bDescriptorType;        /* String Descriptor (3) */
        UH      bString[1];             /* string (Unicode)  */
} usbStringDescriptor;
```

```
/* USB event definition structure (USB Manager) */
typedef struct {
        UB      bClass;         /* Device/Interface Class      */
        UB      bSubClass;      /* Device/Interface Subclass   */
        UB      bProtocol;      /* Device/Interface Protocol   */
        UB      mask;           /* bClass/bSubclass/bProtocol/devid selection */
} usbEventPattern;

/* Response message in nowait mode (message buffer)   */
typedef struct {
        W       pid;    /* pipe ID */
        W       datacnt;/* data count */
        W       error;  /* error code */
} usbMsg;

/* mask values */
#define EVENT_CLASS             0x01
#define EVENT_SUBCLASS          0x02
#define EVENT_PROTOCOL          0x04
#define EVENT_ANY               0x08

/* Error codes (USB Manager) */
#define USB_OK                  (E_OK)
#define USB_ERR_BUSY            (E_BUSY     |  0)
#define USB_ERR_PAR             (E_PAR      |  0)
#define USB_ERR_DEVICE          (E_PAR      |  1)
#define USB_ERR_INTERFACE       (E_PAR      |  2)
#define USB_ERR_ENDPOINT        (E_PAR      |  3)
#define USB_ERR_POWER           (E_LIMIT    |  0)
#define USB_ERR_REQUEST         (E_OACV     |  0)
#define USB_ERR_SYSTEM          (E_SYS      |  0)
#define USB_ERR_NOMEM           (E_NOMEM    |  0)
#define USB_ERR_STALL           (E_IO       |  2)
#define USB_ERR_ABORT           (E_IO       |  3)
#define USB_ERR_IO_NAK          (E_IO       |  6)
#define USB_ERR_IO_SHORT        (E_IO       |  7)
#define USB_ERR_IO_BUFERR       (E_IO       |  9)
#define USB_ERR_IO_BABBLE       (E_IO       | 10)
```

```
#define   USB_ERR_IO_CRC        (E_IO           | 11)
#define   USB_ERR_IO_BITSTUFF   (E_IO           | 12)
#define   USB_ERR_IO_NORESP     (E_IO           | 13)

/* For designation in usbIoPipe() */
#define   USB_WAIT              0x00
#define   USB_SHORTNG           0x00
#define   USB_NOWAIT            0x01
#define   USB_SHORTOK           0x02

/* Structure used with usbGetHubInfo()   */
/* hub status structure */
typedef union {
    struct {
        UH      level: 3;       /* hub levels          */
        UH      self_power: 1;  /* 1 if self-powered hub       */
        UH      reserved: 12;
    } bmStatus;
    UH status;
} usbHubStatus;

/* device status (same as hub device port status)   */
#define   PS_PORT_CONNECTION        0x0001
#define   PS_PORT_ENABLE            0x0002
#define   PS_PORT_SUSPEND           0x0004
#define   PS_PORT_OVER_CURRENT      0x0008
#define   PS_PORT_RESET             0x0010

#define   PS_PORT_POWER             0x0100
#define   PS_PORT_LOW_SPEED         0x0200

/* Structure used for USB event notification (USB Manager)   */
typedef struct {
    ID          address;        /*address of connected device/interface */
    W           evtype;         /*event type          */
    BOOL        interface;      /* device=FALSE, interface=TRUE      */
    struct {
        UB      bNumber;        /* (interface)bInterfaceNumber         */
        UB      bClass;         /* bDeviceClass/bInterfaceClass        */
        UB      bSubClass;      /* bDeviceSubClass/bInterfaceSubClass */
```

```
        UB      bProtocol;      /* bDeviceProtocol/bInterfaceProtocol */
    } info;
} usbReq;
```

## 3.6 USB Events

The USB Manager uses T-Kernel/SM function tk_evt_dev() to notify registered device drivers of USB events.

When an event is notified, it triggers the device driver event handler function designated by tk_def_dev(UB *devnm, T_DDEV *ddev, T_IDEV *idev) in ddev.eventfn. An event handler must be able to accept USB events in any circumstances, process them quickly and pass a return code (response code).

USB event calling takes place using the usbReq structure.

When the event handler function ddev.eventfn(INT evttyp, VP evtinf, VP exinf) is called, TDV_USBEVT is put in evttyp, a pointer to the usbReq structure is put in evinf, and exinf holds the value designated in ddev.exinf when the device driver was registered by tk_def_dev(). in exinf. Note that the contents of the area indicated by evtinf (pointer to the usbReq structure) must not be discarded.

USB events can be notified on a per-device basis or per-interface basis.

- Device connection request

| | | |
|---|---|---|
| usbReq: | address | connected device address |
| | evtype | USB_ATTACH |
| | interface | FALSE |
| | info.bNumber | reserved (0) |
| | info.bClass | connected device bDeviceClass |
| | info.bSubClass | connected device bDeviceSubClass |
| | info.bProtocol | connected device bDeviceProtocol |

- Device disconnection request

| | | |
|---|---|---|
| usbReq: | address | disconnected device address |
| | evttype | USB_DETACH |
| | interface | FALSE |
| | info.bNumber | reserved (0) |
| | info.bClass | reserved (0) |
| | info.bSubClass | reserved (0) |
| | info.bProtocol | reserved (0) |

- Interface connection request

| | | |
|---|---|---|
| usbReq: | address | connected device address |
| | evttype | USB_ATTACH |

| | | |
|---|---|---|
| | interface | TRUE |
| | info.bNumber | connected interface bInterfaceNumber |
| | info.bClass | connected interface bInterfaceClass |
| | info.bSubClass | connected interface bInterfaceSubClass |
| | info.bProtocol | connected interface bInterfaceProtocol |

- Interface disconnection request

| | | |
|---|---|---|
| usbReq: | address | disconnected interface address |
| | evttype | USB_DETACH |
| | interface | TRUE |
| | info.bNumber | disconnected interface bInterfaceNumber |
| | info.bClass | reserved (0) |
| | info.bSubClass | reserved (0) |
| | info.bProtocol | reserved (0) |

- Response to all requests

  The return code (USB_OWN or USB_NONE) of the event handler function executed for an event request is the response code to the USB Manager.
  If any other value is returned, the action is undefined.

### 3.6.1 USB_ATTACH event (device interface connected)

This event is notified sequentially to the device drivers meeting the following conditions among the USB devices registered at the time a USB device is connected. When the response code from a driver is USB_OWN, that driver is associated with the device and event notification ends.

- The driver is not yet associated with a device or interface.
- The device or interface matches the class declared when the driver was registered.

If no driver is associated with the device, the USB Manager configures the device as configuration index 0. A similar procedure is then used to find a driver corresponding to the interface whose alternate setting (bAlternateSetting) is 0.

The sequence in which device interface connected events are notified is the opposite of that in which the drivers were registered. That is, the most recently registered driver is the first to receive an event.

An USB_ATTACH event notification is also made at the time of device driver registration by usbRegistDevice() or usbRegistInterface() if there is a device still without an associated driver.

A driver receiving an USB_ATTACH event notification checks for a match with the connected device or interface based on the device interface class obtained in the event call or using the descriptors obtained by usbDescriptorDevice(), etc.

If the check shows the device is not a matching one, USB_NONE is returned as the response code, and the driver does not access that device.

If the check results in a match, the response code USB_OWN is returned. The time up to

return of the response code may be used to prepare for communication with the device, such as by performing device configuration (required only if the driver is associated with the device) and analyzing descriptors.

### 3.6.2 USB_DETACH event (device interface disconnected)

When a USB device is disconnected, the USB_DETACH event is notified to the driver associated with that device and interface.

When a driver receives the USB_DETACH event, it performs whatever processing is defined for device or interface disconnection and returns a response. The response code for this event may be either USB_OWN or USB_NONE.

When a device is disconnected, the association between the device and driver is canceled and the driver is no longer able to perform operations on the device.

### 3.6.3 Suspend/resume processing

In SUSPEND state, the USB Manager voids the connection to the device. Accordingly, a device interface disconnected event is notified to the device driver using the USB device, and the device is effectively nonexistent during SUSPEND state.

Then, when going to RESUME state, the USB Manager connects the device, notifying a device interface connected event to the device driver for that device.

## 3.7 USB Manager System Calls

The USB Manager provides the following services to device drivers as extended system calls.

USB_ERR_IO_* in the error code description is an error that occurs during communication with a USB device. This error is explained in the next section.

All USB Manager system calls can be called independently of the invoking task. This means, for example, that the task used to open a device with usbOpenDevice() need not be the same task as that used to close the device with usbCloseDevice().

### 3.7.1 usbRequestDevice–Issue device request

[Format]

ER usbRequestDevice(W did, VP request, VP data, W len, W *rlen)

[Parameters]

| | |
|---|---|
| did | Device address |
| request | Pointer to the device request to be sent to the device |
| data | Pointer to the start of the memory buffer for holding data to be exchanged |
| len | Size of data to be exchanged |
| rlen | Pointer to the area to hold the size of data that was exchanged |

[Return Code]

| | |
|---|---|
| = 0 (USB_OK) | Device request issued successfully |
| < 0 | Error (error code) |

[Description]

Issues various device requests to the device. There are no limits on the device requests that can be issued. If, therefore, SET_ADDRESS or another standard device request that will change the device's basic settings (one with the settings of bits 5, 6, and 7 of bmRequestType are all 0) is issued, the subsequent USB Manager behavior is not guaranteed at all.

This function can be used whether the device is open or closed. Requests for the same device can be issued by multiple tasks, but the sequence of the requests sent to a device will be their order of arrival. If this is a device with multiple interfaces, requests to the same device may be sent by more than one driver. In such cases, careful attention must be paid to the request contents and sequence.

Interruption from a short packet (receipt of data shorter than the requested data) is not treated as error. It is therefore advisable that the transferred data length be confirmed.

The state during issuing of a device request is WAIT state.

NULL can be designated for data, in which case len must be set to 0.
If len is not zero when data is NULL, the behavior is not guaranteed.

Note that the value designated for len is set as wLength in the USB Manager, so it is not necessary to designate wLength with a device request.

[Error Code]

| | |
|---|---|
| USB_ERR_DEVICE | Illegal did (no such device) |
| USB_ERR_STALL | Stall occurred |
| USB_ERR_ABORT | Communication canceled |
| USB_ERR_IO_* | IO error occurred |
| USB_ERR_PAR | Request is NULL |

### 3.7.2 usbDescriptorDevice—Get device descriptor

### 3.7.3 usbDescriptorInterface—Get interface descriptor

### 3.7.4 usbDescriptorEndpoint—Get endpoint descriptor

[Format]

```
ER   usbDescriptorDevice(W did, VP data, W len, W *rlen)
ER   usbDescriptorInterface(W iid, VP data, W len, W *rlen)
ER   usbDescriptorEndpoint(W pid, VP data, W len, W *rlen)
```

[Parameters]

| | |
|---|---|
| did | Device address (usbDescriptorDevice()) |
| iid | Interface ID (usbDescriptorInterface()) |
| pid | Pipe ID(usbDescriptorPipe()) |

data Pointer to start of the memory buffer for holding data to be acquired
len Size of data to be acquired
rlen Pointer to the area to hold the descriptor size

[Return Code]
= 0 (USB_OK)Descriptor acquired successfully
< 0 Error (error code)

[Description]

(usbDescriptorDevice())
In the device designated by did, this function gets the device descriptor, configuration descriptors (including the interface descriptor, endpoint descriptor, and various class descriptors), and the string descriptor indicated by the iProduct of the device descriptor.

(usbDescriptorInterface(), usbDescriptorEndpoint())
This function gets the descriptor of the interface designated by iid or of the endpoints (pipe) designated by pid. If it is followed by a class descriptor, this can be obtained with it.

These functions can be used whether the device is open or closed. If an interface number or endpoint address is needed with usbRequestDevice(), the descriptor information obtained by these functions can be used.

These functions do not communicate with the device but simply copy information obtained by the USB Manager when the device was connected. A function should be called first with data=NULL and len=0, and then called again after allocating enough memory using the descriptor size obtained in rlen.

The descriptors specified in the USB standards are defined in include/device/usb.h.
See the USB 1.1 standard for details.

[Error Code]

| | |
|---|---|
| USB_ERR_DEVICE | Illegal did (device designated by did does not exist) |
| USB_ERR_INTERFACE | illegal iid (interface designated by iid does not exist) |
| USB_ERR_ENDPOINT | illegal pid (endpoint designated by pid does not exist) |

### 3.7.5 usbConfigDevice–Set/get device configuration

[Format]
INT usbConfigDevice(W did, W cfg)

[Parameters]
did Device address
cfg 0-255: Choice of configuration (standard device request SET_CONFIGURATION is issued)
-1: Get current configuration (standard device request GET_CONFIGURATION is issued)

[Return Code]

| | |
|---|---|
| = 0 (USB_OK) | Configuration set successfully (when cfg is 0 to 255) |
| 0-255 | Current configuration (when cfg is -1) |
| < 0 | Error (error code) |

[Description]

This function is used to configure the device by determining the interface to use. It can also be used to get the current configuration from the device.

When setting the configuration, all interfaces belonging to the device must be closed. When cfg=0 is designated, the device is put in an unconfigured state.

A device that has just been opened by usbOpenDevice() only has an address assigned to it. Thus, a configuration must be selected for it using this function, and the interface must be determined. Even if an interface is decided, no events occur for drivers registered using usbRegistInterface().

[Error Code]

| | |
|---|---|
| USB_ERR_DEVICE | Illegal did (no such device) |
| USB_ERR_STALL | Stall occurred |
| USB_ERR_ABORT | Communication canceled |
| USB_ERR_IO_* | IO error occurred |
| USB_ERR_BUSY | An interface has not been closed |
| USB_ERR_INTERFACE | Designated configuration not found |
| USB_ERR_POWER | Configuration cannot be set (insufficient hub current) |
| USB_ERR_PAR | cfg outside the range -1 to 255 |

### 3.7.6 usbConfigInterface–Set/get interface alternate setting

[Format]

INT usbConfigInterface(W iid, W alt)

[Parameters]

| | | |
|---|---|---|
| iid | Interface ID | |
| alt | 0 to 255: | Choice of I/F alternate setting (SET_INTERFACE is issued) |
| | -1: | Get current alternate setting(GET_INTERFACE is issued) |

[Return Code]

| | |
|---|---|
| = 0 (USB_OK) | Alternate setting made (when alt is 0 to 255) |
| 0-255 | Current alternate setting (when alt is -1) |
| < 0 | Error (error code) |

[Description]

This function is used to select the alternate setting for a device interface. It can also be used to get the current interface setting.

When the alternate setting is set, all pipes must be closed.

For devices that have just been opened, the designated alternate setting will be 0.
This function is used mainly with devices such as printer devices that have multiple alternate settings.

[Error Code]

| | |
|---|---|
| USB_ERR_DEVICE | Device operation disabled (device is closed) |
| USB_ERR_INTERFACE | illegal iid (interface does not exist) |
| USB_ERR_STALL | Stall occurred |
| USB_ERR_ABORT | Communication canceled |
| USB_ERR_IO_* | IO error occurred |
| USB_ERR_BUSY | A pipe has not been closed |
| USB_ERR_ENDPOINT | Designated alternate setting not found |
| USB_ERR_PAR | alt outside the range -1 to 255 |

### 3.7.7 usbStallPipe-Set/clear endpoint stall

[Format]

INT usbStallPipe(W pid, W stl)

[Parameters]

| | | |
|---|---|---|
| pid | Pipe ID | (obtained by usbOpenPipe()) |
| stl | 1 | SET_FEATURE(ENDPOINT_STALL) is issued |
| | 0 | CLEAR_FEATURE(ENDPOINT_STALL) is issued |
| | -1 | GET_STATUS(ENDPOINT) is issued |
| | 2 | SET_FEATURE(ENDPOINT_STALL) is issued, followed by CLEAR_FEATURE(ENDPOINT_STALL). |

[Return Code]

| | |
|---|---|
| = 0 (USB_OK) | Endpoint set successfully (when stl is 0, 1, or 2) |
| >=0 | Endpoint status (when stl is -1) |
| < 0 | Error (error code) |

[Description]

This function is used to set or get the STALL state of the endpoint.

When USB data transfer is performed, the data sequence is determined by the value 0 and 1 of toggle bits. If the device and host have different toggle bit values, data transfer does not take place correctly. When CLEAR_FEATURE (ENDPOINT_STALL) is issued using usbStallPipe(pid, 0) or usbStallPipe(pid, 2), the toggle bit in the device is set to 0 by the device request and the toggle bit for the pid kept in the USB Manager is also reset to 0.

The reason for providing usbStallPipe(pid, 2) is that for some devices, simply issuing CLEAR_FEATURE (ENDPOINT_STALL) does not set the toggle bit to 0. SET_FEATURE (ENDPOINT_STALL) must be issued first, followed by CLEAR_FEATURE (ENDPOINT_STALL).

[Error Code]

| | |
|---|---|
| USB_ERR_DEVICE | Device operation disabled |
| USB_ERR_INTERFACE | Interface operation disabled |
| USB_ERR_ENDPOINT | Illegal did (no such pipe) |
| USB_ERR_STALL | Stall occurred |
| USB_ERR_ABORT | Communication canceled |
| USB_ERR_IO_* | IO error occurred |
| USB_ERR_PAR | stl not -1, 0, 1, or 2 |

### 3.7.8 usbSyncPipe–Synchronize endpoints

[Format]

INT usbSyncPipe(W pid)

[Parameters]

| | |
|---|---|
| pid | Pipe ID (obtained by usbOpenPipe()) |

[Return Code]

| | |
|---|---|
| >=0 | Frame number returned by device |
| < 0 | Error (error code) |

[Description]

This function issues the USB standard device request SYNCH_FRAME to an endpoint.

[Error Code]

| | |
|---|---|
| USB_ERR_DEVICE | Device operation disabled |
| USB_ERR_INTERFACE | Interface operation disabled |
| USB_ERR_ENDPOINT | Illegal did (no such pipe) |
| USB_ERR_STALL | Stall occurred |
| USB_ERR_ABORT | Communication canceled |
| USB_ERR_IO_* | IO error occurred |

### 3.7.9 usbOpenDevice–Open device

[Format]

INT usbOpenDevice(W did)

[Parameters]

| | |
|---|---|
| did | Device address (can be retrieved from a device connected event) |

[Return Code]

| | |
|---|---|
| > 0 | Opened successfully (value designated in did is returned) |
| < 0 | Error (error code) |

[Description]

This function declares the start of a device operation. Multiple opening of a device is not

allowed.

[Error Code]

| | |
|---|---|
| USB_ERR_DEVICE | Illegal did (no such device) |
| USB_ERR_BUSY | Device already open |

### 3.7.10 usbCloseDevice–Close device

[Format]

ER usbCloseDevice(W did)

[Parameters]

| | |
|---|---|
| did | Device address |

[Return Code]

| | |
|---|---|
| = 0 (USB_OK) | Closed the device |
| < 0 | Error (error code) |

[Description]

This function declares the end of a device operation. When a device is closed, the interfaces and pipes belonging to that device are also closed.

A device can be closed by a task other than the one that opened it. Thus, it is important to set the did correctly.

[Error Code]

| | |
|---|---|
| USB_ERR_DEVICE | Illegal did (no such device) |

[Additional Note]

Because usbCancelDevice() is issued internally, the function waits until all pipes belonging to the device are closed.

### 3.7.11 usbOpenInterface–Open interface

[Format]

INT usbOpenInterface(W did, W ifno)

[Parameters]

| | |
|---|---|
| did | Device address (can be retrieved from an event) |
| ifno | Interface number (bInterfaceNumber in interface descriptor) |

[Return Code]

| | |
|---|---|
| >=0 (USB_OK) | Interface ID (iid) |
| < 0 | Error (error code) |

[Description]

This function declares the start of an interface operation. Multiple opening of an interface is not allowed.

[Error Code]

| | |
|---|---|
| USB_ERR_DEVICE | Illegal did (no such device, or operation disabled) |
| USB_ERR_INTERFACE | Illegal ifno (no such interface) |
| USB_ERR_BUSY | Interface already open |
| USB_ERR_NOMEM | No more interfaces can be opened |
| USB_ERR_SYSTEM | USB Manager internal error |

### 3.7.12 usbCloseInterface–Close interface

[Format]

ER usbCloseInterface(W iid)

[Parameters]

iid Interface ID

[Return Code]

| | |
|---|---|
| = 0 (USB_OK) | Closed the interface |
| < 0 | Error (error code) |

[Description]

This function declares the end of an interface operation. When an interface is closed, all the endpoints belonging to that interface are closed. An interface can be closed by a task other than the one that opened it. Thus, it is important to set the iid correctly.

[Error Code]

| | |
|---|---|
| USB_ERR_INTERFACE | Illegal iid (no such interface) |

[Additional Note]

Because usbCancelInterface() is issued internally, the function waits until all pipes belonging to the interface are closed.

### 3.7.13 usbOpenPipe–Open endpoint (create pipe)

[Format]

INT usbOpenPipe(W iid, W epadr, W mode, W mbfid)

[Parameters]

| | | |
|---|---|---|
| iid | Interface ID (obtained by usbOpenInterface()) | |
| epadr | Endpoint address of operation (bEndpointAddress in endpoint descriptor) | |
| mode | Mode (USB_WAIT \|\| USB_NOWAIT) \| (USB_SHORTNG \|\| USB_SHORTOK) | |
| | USB_NOWAIT | Do not wait for read/write to end (nowait mode) |
| | USB_WAIT | Wait for read/write to end (wait mode) |
| | USB_SHORTNG | If a short packet (transfer ends with shorter data size than requested data) is detected during read/write, terminate with USB_ERR_SHORT error |

| | | |
|---|---|---|
| | USB_SHORTOK | If short packet is detected during read/write, terminate with USB_OK |
| mbfid | Message buffer ID for receiving status in nowait mode (designating a negative value means no status is to be received) | |

[Return Code]

>=0 Opened the pipe (pipe ID)
< 0 Error (error code)

[Description]

This function creates a pipe (communication channel between endpoints) and declares the start of an operation on the designated endpoint. Multiple opening of an endpoint is not allowed.

nowait mode can be designated only for a pipe that uses interrupt transfer. It cannot be designated for a pipe using bulk transfer or other transfer modes.

When USB_WAIT is designated, the mbfid value is ignored. When USB_NOWAIT is designated, after transfer ends, the status message defined in the usbMsg structure is sent to the message buffer designated in mbfid.

[Error Code]

| | |
|---|---|
| USB_ERR_DEVICE | Device operation disabled |
| USB_ERR_INTERFACE | Illegal iid (no such interface) |
| USB_ERR_ENDPOINT | Illegal epadr (no such endpoint) |
| USB_ERR_BUSY | Pipe already open |
| USB_ERR_NOMEM | No more pipes can be opened |

### 3.7.14 usbClosePipe–Close endpoint (delete pipe)

[Format]

ER usbClosePipe(W pid)

[Parameters]

| | |
|---|---|
| pid | Pipe ID |

[Return Code]

= 0 (USB_OK) Closed the pipe
< 0 Error (error code)

[Description]

This function declares the end of an endpoint operation. All communication using the pipe is canceled.

Because usbCancelPipe() is issued internally, the function waits until cancellation is complete.

A pipe can be closed by a task other than the one that opened it. Thus, it is important to

set the pid correctly.

[Error Code]

USB_ERR_ENDPOINT Illegal pid (no such pipe)

### 3.7.15 usbIoPipe–Exchange data with endpoint

[Format]

ER usbIoPipe(W pid, VP buf, W len, W *rlen)

[Parameters]

pid Pipe ID
buf Pointer to start of data to be output
len Size of data to be output
rlen Pointer to area holding size of actually transferred data

[Return Code]

= 0 (USB_OK) Successful transfer
< 0 Error (error code)

[Description]

This function is used to transfer data on the pipe designated by pipe ID in the direction designated when usbOpenPipe() was called.

Except when USB_NOWAIT was designated with usbOpenPipe(), this function does not return control until the action is complete.

When USB_NOWAIT was designated, 0 is stored in rlen. Attention must also be paid to the following points.

The transfer complete notice and status are stored in the message buffer designated with usbOpenPipe(). (If the message buffer is full, notification cannot be made, so it is important to make sure there is space available.)

Space for holding the transfer data must have been allocated before transfer ends. The behavior is not guaranteed if the buffer space for this data is not freed before transfer ends.

If dat is set to NULL when data is received from an endpoint, it is possible to read and discard only the bytes of data designated in len. In this case, however, len must be an exact multiple of the endpoint wMaxPacketSize. Otherwise, subsequent data transfer will not be possible.

When this function is used to send data, dat must not be set to NULL.

[Error Code]

USB_ERR_DEVICE Device operation disabled
USB_ERR_INTERFACE Interface operation disabled
USB_ERR_ENDPOINT Illegal pid (no such pipe)
USB_ERR_STALL Stall occurred

| | |
|---|---|
| USB_ERR_ABORT | Communication canceled |
| USB_ERR_IO_* | IO error occurred |
| USB_ERR_BUSY | Transfer request cannot be accepted (insufficient bandwidth) |

### 3.7.16 usbCancelDevice–Cancel communication (for designated device)

### 3.7.17 usbCancelInterface–Cancel communication (for designated interface)

### 3.7.18 usbCancelPipe–Cancel communication (for designated endpoint)

[Format]

ER usbCancelDevice(W did)
ER usbCancelInterface(W iid)
ER usbCancelPipe(W pid)

[Parameters]

did Device address (designated with usbCancelDevice())
iid Interface ID (designated with usbCancelInterface())
pid Pipe ID (designated with usbCancelPipe())

[Return Code]

= 0 (USB_OK) Communication was canceled successfully
< 0 Error (error code)

[Description]

usbCancelPipe() cancels communication on the pipe designated by pid.

usbCancelInterface() cancels communication on all pipes included in the interface designated by iid.

usbCancelDevice() cancels communication on all pipes included in the device designated by did.

In wait mode, the error code USB_ERR_ABORT is returned to the task executing communication by usbIoPipe(). In nowait mode, a status message is put in the message buffer designated with usbOpenPipe(), and the error code is USB_ERR_ABORT.

Regardless of whether wait or nowait is designated for a pipe, this function waits until the pipe communication is aborted.

[Error Code]

| | |
|---|---|
| USB_ERR_DEVICE | Illegal did (no such device) |
| USB_ERR_INTERFACE | Illegal iid (no such interface) |
| USB_ERR_ENDPOINT | Illegal pid (no such pipe) |

### 3.7.19 usbAlivePipe–Check pipe availability

[Format]

ER usbAlivePipe(W pid)

[Parameters]

pid Pipe ID

[Return Code]

= 0 (USB_OK) Pipe available
< 0 Error (error code)

[Description]

This function checks whether the pipe designated by pid is available. It is used, for example, after communication is cut off to find out whether continued access is possible.

[Error Code]

| | |
|---|---|
| USB_ERR_DEVICE | No such device |
| USB_ERR_INTERFACE | No such interface |
| USB_ERR_ENDPOINT | No such pipe |

### 3.7.20 usbRegistDevice–Register destination for device connected/disconnected event notification

[Format]

ER usbRegistDevice(ID devid, usbEventPattern *pattern)

[Parameters]

| | |
|---|---|
| devid | Device ID of device driver |
| pattern | Pointer to event notification conditions (When pattern == NULL, registration is canceled and no more event notification is made to the designated device driver.) |

[Return Code]

= 0 (USB_OK) Registration succeeded
< 0 Error (error code)

[Description]

This function designates the destination for notifying device events when a USB device is connected or disconnected. The conditions for receiving a device event are designated in pattern. Multiple sets of conditions can be registered for the same devid (physical device ID), but upon deletion, all conditions associated with the device driver are deleted.

The usbEventPattern structure has the following format. This structure is used also with usbRegistInterface() as described later on.

```
typedef struct {
        UB   bClass;
```

```
    UB  bSubClass;
    UB  bProtocol;
    UB  mask;
} usbEventPattern;
```

When usbRegistDevice() is used, the bClass, bSubClass, and bProtocol values correspond to bDeviceClass, bDeviceSubClass, and bDeviceProtocol in the device descriptor.
When usbRegistInterface() is used, the bClass, bSubClass, and bProtocol values correspond to bInterfaceClass, bInterfaceSubClass, and bInterfaceProtocol in the interface descriptor.

There are four kinds of mask, as follows. An event is issued to a device by designating either EVENT_ANY or a combination of EVENT_CLASS, EVENT_SUBCLASS, and EVENT_PROTOCOL (mask cannot be 0).

| | |
|---|---|
| EVENT_ANY | Any device interface type |
| EVENT_CLASS | Compare by bClass |
| EVENT_SUBCLASS | Compare by bSubClass |
| EVENT_PROTOCOL | Compare by bProtocol |

[Error Code]

| | |
|---|---|
| USB_ERR_DEVICE | No device connected event defined for the designated device driver (when pattern == NULL was designated) |
| USB_ERR_NOMEM | No more device events can be registered |
| USB_ERR_PAR | pattern.mask is 0 |

[Additional Note]

From the standpoint of device configuration difficulty, it is easier to use the usbRegistInterface() function (described next) to create drivers at the interface level.

### 3.7.21 usbRegistInterface–Register destination for interface connected/disconnected event notification

[Format]

```
ER  usbRegistInterface(ID devid, usbEventPattern *pattern)
```

[Parameters]

| | |
|---|---|
| devid | Device ID of device driver |
| pattern | Pointer to event notification conditions (When pattern == NULL, registration is canceled and no more event notification is made to the designated device driver.) |

[Return Code]

| | |
|---|---|
| = 0 (USB_OK) | Registration succeeded |
| < 0 | Error (error code) |

[Description]

When a USB device is connected but no corresponding driver exists, configuration is performed by the USB Manager. The configuration used in this case is the one in the first configuration descriptor that was read (index 0).

This function designates the destination for interface event notification, made when a usable interface is decided after configuration by the USB Manager is completed. The conditions for receiving an interface event are set in the pattern parameter.

Note that even if the interface class matches, an event is not notified if bAlternateSetting is not 0.

See the description of usbRegistDevice() for details on the usbEventPattern structure.

[Error Code]

| | |
|---|---|
| USB_ERR_INTERFACE | No interface event defined for the designated device driver (when pattern == NULL was designated) |
| USB_ERR_NOMEM | No more interface events can be registered |
| USB_ERR_PAR | pattern.mask is 0 |

[Additional Note]

Event notification when an interface is connected is made for an interface descriptor with bAlternateSetting of 0 and having bClass, bSubClass, and bProtocol values.

### 3.7.22 usbResetDevice–Reset device (software-based disconnection and reconnection)

[Format]

```
ER   usbResetDevice(W did)
```

[Parameters]

did  Device address

[Return Code]

| | |
|---|---|
| = 0 (USB_OK) | Device was reset |
| < 0 | Error |

[Description]

This function resets the device designated by did.

This action has the same effect as disconnecting and then reattaching a device, and event notification first for device disconnection and then for device reconnection is made to the driver associated with the reset device.

After resetting, there is no guarantee that the device will have the same address as before resetting. This function does not wait for resetting to complete before returning control.

[Error Code]

| | |
|---|---|
| USB_ERR_DEVICE | Illegal did (no such device) |

### 3.7.23 usbGetHubInfo–Get device connection information

[Format]

```
INT     usbGetHubInfo(W *report, W size)
```

[Parameters]

| | |
|---|---|
| report | Pointer to the memory space for putting the obtained information |
| size | Size of the area for holding the obtained information |

[Return Code]

| | |
|---|---|
| >=0 | Size in bytes of the actually obtained information |
| < 0 | Error |

[Description]

This function acquires connection information about a USB hub device. Either an error code or the size (in bytes) of the acquired information is set in the return code. If report is NULL, only the size of the information is obtained and not the information itself.

The report format is as follows.

```
bit 31                                  16  15                          0
+0 [[  hub device address                  ][          hub device status   ]]
bit 31                                      8  7                        0
+4 [[          not used (0)                    ][hub device ports         ]]

+8 -    (repeated for each port)
bit 31                              16  15                              0
   [[ address of connected devices ][        device                      ]]
```

The hub device status is indicated in the following structure.

```
typedef union  {
      struct     {
            UH   level: 3;          /* hub level          */
            UH   self_power: 1;     /* 1 if self-powered hub      */
            UH   reserved: 12;
      }          bmStatus;
      UH         status;
}     usbHubStatus;
```

If there is no device connected to the hub device, the device address is -1.

The device status is given as follows.

PS_PORT_CONNECTION 0x0001
Set if a device is connected

PS_PORT_ENABLE 0x0002
Set if a device is open

PS_PORT_SUSPEND 0x0004
Set if a device is in SUSPEND state (normally this value will not be set)

PS_PORT_OVER_CURRENT 0x0008
Set in case of an overcurrent in a device (If two bus-powered hubs are connected in series, this is set for the hub farthest from the host and that hub cannot be used.)

PS_PORT_RESET 0x0010
Set if a device was reset (normally this value will not be set)

PS_PORT_POWER 0x0100
Set if power is supplied to the device (This value is set ordinarily, but it may not be set if a problem occurs in the hub or device.)

PS_PORT_LOW_SPEED 0x0200
Set if a low speed device (keyboard, mouse, etc.) is connected.

[Error Code]
USB_ERR_NOMEM Space for holding the report is not allocated

## 3.8 Additional Notes on USB Manager System Calls

This section describes the errors indicated in the previous section (on USB Manager system calls) as USB_ERR_IO * (errors occurring during communication with a USB device).

USB_ERR_IO_NAK
An error code returned when the device returns a NAK response for more than a predetermined duration (10 seconds). This error does not occur for a pipe using interrupt transfer.

USB_ERR_IO_SHORT
An error code returned when USB_SHORTOK was not designated with usbOpenPipe() and in data exchange using usbIoPipe(), a data transfer ended with a shorter data length than the designated size.

USB_ERR_IO_BABBLE
An error code returned when babble occurred

USB_ERR_IO_CRC
An error code returned when CRC error occurred.

USB_ERR_IO_BITSTUFF
An error code returned when bit stuff error occurred.
When USB_ERR_IO_BABBLE, USB_ERR_IO_CRC, or USB_ERR_IO_BITSTUFF occur frequently, some problem in the USB device may be the cause.

USB_ERR_IO_BUFERR

An error code returned in case of a transfer problem between the USB Host Controller and main memory. Ordinarily this error does not occur, but it is possible depending on the host controller type.

USB_ERR_IO_NORESP

An error code returned when communication to a device is attempted during the time lag between device disconnection and notification of device disconnection by the USB Manager.
When this error occurs, further operations on the device should be avoided.

# 4. LAN Driver

TEF040-S204-01.00.00/en

## 4.1 Applicable Devices

- This driver applies to network interface devices in a LAN (Local Area Network).

## 4.2 Device Name

- The device name is "Neta".

## 4.3 Device-specific Functions

- Sending and receiving LAN packets
- Getting and setting information required for transmission control
- The LAN driver interface is mainly involved with asynchronous packet reception.
  It uses the following methods to avoid wasteful data copying.

  Sending: Individual packets are written to the device driver using regular methods.

  Receiving: Multiple receive buffer (pointers) are passed to the device driver beforehand, and when packets are received, notification is issued with the message buffer.
  Buffer management is provided by higher layer software (via TCP/IP).

- Unicast, broadcast, and multicast are supported for the packets to be received.

## 4.4 Attribute Data

DN_NETEVENT (-100): Event notification message buffer ID RW

data: ID

gets/sets event notification message buffer ID.

DN_NETRESET (-103): Reset RW

data: W

The network adaptor is reset and operation restored by reading or writing arbitrary data.

DN_NETADDR (-105): Network physical address R

data: NetAddr

typedef struct {

```
        UB      c[6];
    } NetAddr;
```

Gets the Ethernet physical address set in the network adaptor.

DN_NETDEVINFO (-110): Network device information R

data: NetDevInfo

```
#define L_NETPNAME  (40)        Product name length
typedef struct {
        UB  name[L_NETPNAME];   Product name (ASCII)
        UW  iobase;             IO start address
        UW  iosize;             IO size
        UW  intno;              Interrupt number
        UW  kind;               Index by hardware type
        UW  ifconn;             Connector
        W   stat;               Operating status (> =0: Normal)
} NetDevInfo;
```

Gets device information on the network adaptor. (Details omitted)

DN_NETSTINFO (-111): Network statistics information R

data:NetStInfo

```
typedef struct {
        UW  rxpkt;          Number of packets received
        UW  rxerr;          Receive error rate
        UW  misspkt;        Number of packets received and discarded
        UW  invpkt;         Number of invalid packets
        UW  txpkt;          Number of sent (requested) packets
        UW  txerr;          Transmission error rate
        UW  txbusy;         Number of instances transmission was busy
        UW  collision;      Number of collisions
        UW  nint;           Number of interrupts
        UW  rxint;          Number of receive interrupts
        UW  txint;          Number of send interrupts
        UW  overrun;        Number of hardware overruns
        UW  hwerr;          Number of hardware errors
        UW  other[3];       Other information
} NetStInfo;
```

Gets statistics information for the network adaptor.

DN_NETCSTINFO (-112): Clears network statistics information R

data:NetStInfo

Gets network adaptor statistics information and then clears all states to 0.

DN_NETRXBUF (-113): Receive buffer W
data:VP

Sets the receive buffer. The receive buffer must be larger than the maximum size for received packets as set in DN_NETRXBUFSZ.
Setting it at NULL discards all previously set receive buffers.

Packet reception requires setting the appropriate number of receive buffers beforehand. When a packet is received, data is set in one of the specified receive buffers and the message buffer specified by DN_NETEVENT is notified of the receive event. Reception a packet decreases the specified receive buffers by one, so it is necessary to set up a new receive buffer.

DN_NETRXBUFSZ (-114): Size of receive buffer RW
data:NetRxBufSz

```
typedef struct {
        W   minsz;          Minimum receive packet size
        W   maxsz;          Maximum receive packet size
} NetRxBufSz;
```

Sets/gets the maximum and minimum sizes of packets to be received. Received packets of sizes outside the specified range are discarded.
The defaults are minsz=60 and maxsz=1520.

Invalid values cause errors. If maxsz exceeds the maximum value defined by the driver, however, there is no error and it is reset to the maximum value.

DN_SET_MCAST_LIST (-115) //: Multicast setting W

data: NetAddr

size: Number of NetAddr

Enables reception of multicast address packets as indicated in NetAddr.
All multicast receptions are disabled if size is 0.

DN_SET_ALL_MCAST (-116) //: All multicast settings W

data: None

Enables support for all multicast receptions.

DN_NETWLANCONFIG (-130): Wireless LAN settings RW

data: WLANConfig

```
#define WLAN_SSID_LEN       32      Maximum SSID length
#define WLAN_WEP_LEN        16      Maximum WEP key length
typedef struct {
```

```
        W       porttype;                   Network type (rw)
        W       channel;                    Channel used (rw)
        W       ssidlen;                    SSID length (in bytes) (rw)
        UB      ssid[WLAN_SSID_LEN];        SSID (rw)
        W       wepkeylen;                  WEP key length (in bytes) (rw)
        UB      wepkey[WLAN_WEP_LEN];       WEP key (wo)
        W       systemscale;                Sensitivity (rw)
        W       fragmentthreshold;          Fragment threshold (rw)
        W       rtsthreshold;               RTS threshold (rw)
        W       txratecontrol;              Transmission rate (rw)
        UW      function;                   Extended functions (ro)
        UW      channellist;                Available channels (ro)
    } WLANConfig;
```

Gets/sets required information for wireless LAN use. (Details omitted)

DN_NETWLANSTINFO (-131): Gets line information for wireless LAN R

data:WLANStatus

```
typedef struct {
    UB      ssid[WLAN_SSID_LEN+2];  Destination SSID
    UB      bssid[6];               Destination BSSID
    W       channel;                Current channel
    W       txrate;                 Transmission rate (kbps)
    W       quality;                Line quality
    W       signal;                 Signal level
    W       noise;                  Noise level
    UW      misc[16];               Extended statistics information
} WLANStatus;
```

Gets wireless LAN line information (destination access point information and signal strength) as well as statistics information (with different details depending on the driver). (Details omitted)

DN_NETWLANCSTINFO (-132): Clears wireless LAN line information R

data: WLANStatus

Gets line information for the wireless LAN card, and then clears required items in the extended statistics information to 0.

## 4.5 Device-specific Data

start 0: Sends packet . This is the last packet of fragmented packets. W
1: This is the middle packet of a single fragmented packet.
size: The written number of bytes (packet size)

start=0: A writing session is sent as a single packet.

start=1: Wait until start=0 before transmission.

An error occurs if the maximum packet size for transmission is exceeded, preventing transmission.

## 4.6 Event Notification

The following message for event notification is sent to the message buffer specified in DN_NETEVENT.

```
typedef struct {
        UH      len;    Number of bytes of received data
        VP      buf;    Receive buffer address
} NetEvent;
```

Receive message:

Event notification when a packet is received.

The value for buf is any of the addresses for the receive buffer specified in DN_NETRXBUF, but it is not necessarily in the specified order.

len is the number of bytes of the received packet stored in buf, a value between minsz and maxsz as set in DN_NETRXBUFSZ.

Transmission possible message:

Under the following conditions of event notification, len=0 and buf=NULL.

-After a packet is sent, when transmission of another packet is possible.

-After E_BUSY occurs during transmission, when packet transmission is possible.

## 4.7 Instructions for Use

Common usage by higher layer software (e.g. TCP/IP) are as follows.

1. Open the device for exclusive writing.
2. Read the physical address. (DN_NETADDR)
3. Set the receive buffer size. (DN_NETRXBUFSZ)
4. Keep the receive buffer setting at an appropriate number. (DN_NETRXBUF)
5. Write the event notification message buffer ID. (DN_NETEVENT)
6. Enter standby for event notification message buffer and transmission request.
   For receive messages
      Handle received packets.
      Perform additional settings for the receive buffer. (DN_NETRXBUF)
   For send requests
      For messages that can be sent If there are packets to be sent, send them. (start=0)
7. Discard the receive buffer. (Write NULL to the DN_NETRXBUF)

8. Empty the event notification message buffer.
9. Close the device.

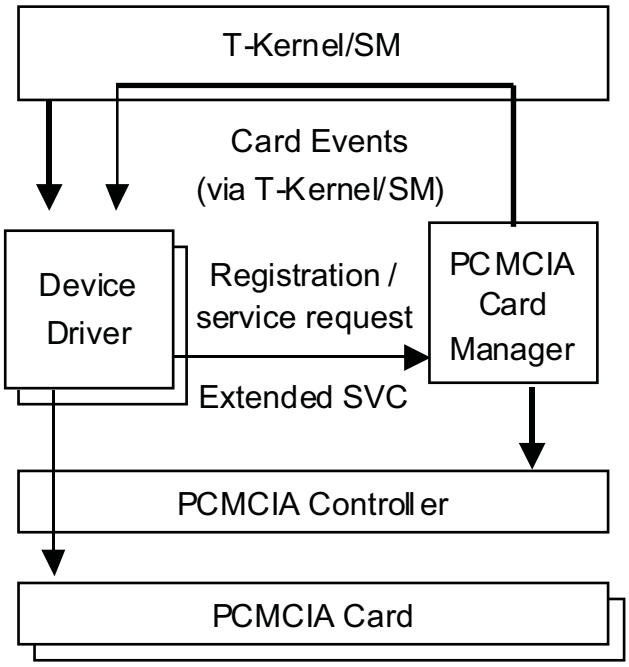
In higher layer software (e.g. TCP/IP), opening for exclusive write enables the device to be opened for read by other utilities, enabling retrieval of network device information and network statistics information.

# 5. PCMCIA Card Manager

TEF040-S205-01.00.00/en

## 5.1 Overview

The PCMCIA Card Manager is a driver corresponding to a PCMCIA controller. It provides an integrated device driver interface for a PCMCIA card ("PC card"), independent of other controllers or machines. Because it is a manager, no device name is given.




## 5.2 Card Manager Functions

The PC Card Manager has the following functions.

- Event detection and notification

  The PC Card Manager detects events such as card insertion and removal as well as changes in battery status (for memory cards only). It notifies associated drivers using the T-Kernel/SM event notification function.

- Executes common card processing

  On card insertion, the card manager handles power supply, reset, attribute data acquisition, and other required processing to enable the corresponding driver to access the card.

  On card ejection, the card manager stops the power supply, unmaps, and performs other processing to disable access to the card.

- Associates cards and drivers

On card insertion, the card manager queries the registered drivers to determine a match for the card in order to associate a driver with the card. Then only the associated driver is allowed to access the card.

On card ejection, the card manager disassociates the driver and card.

- Provides various services to drivers

  Getting and setting attribute data

  Mapping of IO space and memory space

  Memory read/write

  Registering interrupt handlers for interrupts

  Controlling power supply

  Other services

## 5.3 Functions Required by Device Drivers

A device driver for a PC card must have the following functions in addition to the usual driver functions.

- Detecting whether a card is one it matches

  When a card is inserted, the driver must read the CIS tuple information on the card to determine whether it matches the driver.

- Card initialization

  If a card is one matching the driver, the driver must be able to perform card configuration, memory space/IO space mapping, interrupt handler registration, etc., enabling the card to be used for its intended purposes. This driver functionality is commonly called "PC Card Enabler."

- Functions on card insertion and extraction

  Functions must be provided to handle operations such as ejecting or reinserting a card at any time. Especially important here is the handling of access to a card at the time it is removed. Device events must also be generated as necessary.

- Other functions

  In case of a memory card, handling of problems with battery power supply is necessary.

## 5.4 Limitations

The card manager is subject to the following limitations.

- Essentially only one-on-one mapping of card and driver is supported. If a card has multiple functions, it cannot be associated with more than one driver. In other words, the response code CR_SHARE is not supported.

- Support is provided for 16-bit PC Cards powered by 5V or 3.3V PC cards. In some cases only one of these voltages may be supported, however, in consideration of the PC card controller specification and overall power consumption. Also, VPP power supply control is not supported.

- Only CIS tuple information in attribute memory is supported. There is no support for CIS tuple information in common memory using LONG_LINK.

## 5.5 Data Definitions (pcmcia.h)

```
/* Card event types */
#define CE_INSERT       1       /* card insertion              */
#define CE_EJECT        2       /* card ejection               */
#define CE_BATTERY      3       /* card battery status change */

/* Card event response codes */
#define CR_NONE         0       /* not a matching card         */
#define CR_OWN          1       /* matching card: owned        */
#define CR_SHARE        2       /* matching card: shared       */

/* Card types: FUNCID tuple codes */
#define CK_MULTI        0       /* multifunction card          */
#define CK_MEMORY       1       /* memory card                 */
#define CK_SERIAL       2       /* serial port                 */
#define CK_PARALLEL     3       /* parallel port               */
#define CK_FIXED        4       /* fixed disk                  */
#define CK_VIDEO        5       /* video                       */
#define CK_NETWORK      6       /* network                     */
#define CK_AIMS         7       /* AIMS                        */
#define CK_SCSI         8       /* SCSI                        */
#define CK_ANY          255     /* any (unknown)               */
#define CK_NONE         (-1)    /* none (deleted)              */
```

```
/* Card status */
typedef struct {
        UW      kind: 8;        /* card type: CK_xxx                    */
        UW      battery: 2;     /* battery status (memory card only)    */
                                /* 0: OK, 1: alarm, 2, 3: problem (dead)*/
        UW      wprotect: 1;    /* write-protected (memory card only)   */
        UW      power: 1;       /* power supply status                  */
        UW      client: 1;      /* in use                               */
        UW      rsv: 19;        /* reserved                             */
} CardStat;

/* Mapping and memory attributes */
#define CA_IONOCHK      0x10000 /* no IO space resource management */
#define CA_IOMAP        0x8000  /* IO space map                    */
#define CA_ATTRMEM      0x4000  /* attribute memory                */
#define CA_WPROTECT     0x2000  /* write protection                */
#define CA_16BIT        0x0800  /* 16-bit access                   */
#define CA_IOCS16       0x0400  /* IOCS16 source                   */

#define CA_WAIT0        0x0100  /* additional WAIT 0 designation   */
#define CA_WAIT1        0x0101  /* additional WAIT 1 designation   */
#define CA_WAIT2        0x0102  /* additional WAIT 2 designation   */
#define CA_WAIT3        (CA_WAIT1 | CA_WAIT2) /* additional WAIT 3 designation   */
#define CA_ZWAIT        0x0104  /* zero WAIT designation   */
#define CA_SPEED        0x00FF  /* access speed designation in 50 ns increments */

/* Power control   */
#define CP_POWEROFF     0x00    /* power off        */
#define CP_POWERON      0x01    /* power on         */
#define CP_SUSPEND      0x10    /* suspend (power off)      */
#define CP_RESUME       0x11    /* resume (power on)        */
#define CP_SUSRIRES     0x12    /* suspend (RI resume)      */
#define CP_POFFREQ(tm) (((tm)<<16)+0x99) /* power off after designated time (sec) */

/* Special card ID*/
#define TEST_CARDID     0x12345678      /* special test card ID     */
```

```
/* Maximum size of tuple data */
#define MAX_TUPLESZ     (255 + 2)

/* Card events */
typedef struct {
        ID              cardid;         /* card ID                  */
        W               evtype;         /* event type (CE_xxx)      */
        CardStat        cardstat;       /* card status              */
} CardReq;
```

## 5.6 Card Events

The card manager uses T-Kernel/SM function tk_evt_dev() to notify registered device drivers of card events.

Events trigger the device driver event handler function designated by tk_def_dev(UB *devnm, T_DDEV *ddev, T_IDEV *idev) in ddev.eventfn. Event handler functions must be able to accept card events in any circumstances, process them as quickly as possible, and pass a return code (response code).

Card event calling and response take place as follows.

Call (request):

| | |
|---|---|
| CardReq:cardid | card ID |
| evtype | Event type (CE_xxxx) |
| cardstat | Card status = CardStat |

*The card status is always 0 (no significance).

*When ddev.eventfn(INT evttyp, VP evtinf, VP exinf) is called due to acard event, the parameter settings are evttyp = TDV_CARDEVT, evtinf =pointer to CardReq, and exinf = (information designated by tk def dev() in ddev.exinf).

*The contents of the area indicated by evtinf (pointer to CardReq) must not be discarded.

Response:

The event processing function return code must be the card event response code (CR_NONE, CR_OWN, or CR_SHARE). No other value can be returned.

### 5.6.1 CE_INSERT event (card insertion)

When a card is inserted, the event is notified in sequence to each device driver meeting the following conditions. If a driver returns CR_OWN as the response code, that driver is associated with the card and event notification ends.

The driver is not yet associated with a card.

The card type designated during driver registration matches that of the card.

The initial notification sequence for CE_INSERT events is the order in which drivers were registered. The order subsequently changes, however, to give priority to the driver associated with the card previously ejected.

(The reason for this reordering is to handle re-insertion if a card was ejected illegally.)

If a card is already inserted when a driver is registered, CE_INSERT event is notified even if no driver is associated with the card.

A driver receiving CE_INSERT event notification checks whether it is the driver for the inserted card by examining the card status (CardStat) in the event and by executing pcGetTuple() to get the CIS tuple information. Here, the card ID in the event is used as a parameter.

The least significant 2 bits of the card ID represent the physical PCMCIA slot (0 to 3), so that if a driver applies only to a specific slot, this can be determined from the card ID..

If the check shows that the driver is not associated with any card, CR_NONE is returned in the response code, and after this point the driver must not access the card or card manager.

If the check matches the driver to a card, the necessary card configuration, memory and IO mapping, and interrupt handler registration, etc., are performed, then CR_OWN is returned in the response code. The driver is subsequently allowed exclusive access to the card, and this card ID is used as a parameter in system calls to the card manager.

### 5.6.2 CE_EJECT event (card ejection)

When a card is ejected, CE_EJECT event is notified to the associated driver.

A driver receiving CE_EJECT event notification performs processing defined for handling ejection of the associated card and returns a response. The response code in the case of this event has no meaning. The association between the ejected card and driver is canceled, and the card ID in use up to that time is no longer valid.

After a card is ejected, the memory space and IO space maps also become invalid, but the map information is retained, so normally pcUnMap() is issued to release the map information. If, however, it is preferable not to change the map information on the expectation that the same card will be reinserted, it is possible not to execute pcUnMap() and to use pcReMap() when the card is reinserted , making the map valid again.

### 5.6.3 CE_BATTERY event (card battery alarm / problem)

Detection of a memory card battery alarm or battery problem triggers notification of the CE_BATTERY event to the associated driver.

A driver receiving CE_BATTERY event notification confirms the battery status from the card status in the event (CardStat), performs the corresponding device event notification, etc., and returns a response. The response code in the case of this event has no meaning.

If there is already a battery alarm or problem when a card is inserted, notification of the CE_BATTERY event will not always occur. Thus, the battery status is also checked as part of the CE_INSERT event handling based on the card status (CardStat). If the status is not normal, the appropriate processing must be performed.

## 5.7 Suspend / Resume Processing

A device driver using a PC card must perform the following suspend and resume processing.

Suspend processing:

Executes pcPowerCtl (cardid, CP_SUSPEND) to turn off the card.

*If resume by modem card RI is supported, CP_SUSRIRES is designated.

If necessary, pcUnMap() is executed for unmapping.

No further access to the card is allowed from this point.

Resume processing:

Executes pcPowerCtl(cardid, CP_RESUME) to turn on the card power and performs the following processing depending on the function value.

| | |
|---|---|
| == 0: | Power already on (cannot occur) |
| == 1: | Power was turned on, so initialization is performed including the following.<br>IO and memory mapping<br>Interrupt handler registration<br>Card configuration |
| == E_NOMDA: | The card was removed during SUSPEND state and replaced with another card, so the same processing as for ejection is performed (CE_EJECT event).<br>However, no CE_EJECT notification is made.<br>It must be noted that this case differs from the CE_EJECT event in that the card ID is already invalid.<br>The map ID is still valid, so unmapping is performed as necessary. |
| == E_IO: | Power was turned on, but could not go to READY state.<br>Ordinarily this cannot occur; the power remains off. |

The processing when a suspended card is removed and reinserted.

Card removal:

In the driver resume processing, E_NOMDA is returned by pcPowerCtl().

Card replacement (same card):

Same as when replacement does not occur.

In the driver resume processing, (1) is returned by pcPowerCtl().

Card replacement (different card):

In the driver resume processing, E_NOMDA is returned by pcPowerCtl().

When resume is complete, CE_INSERT event notification occurs for the replacement card.

New card insertion:

When resume is complete, CE_INSERT event notification occurs for the

inserted card.

## 5.8 Card Manager System Calls

The card manager provides the following services to device drivers as extended system calls.

### 5.8.1 Register/delete client

[Function]

ER pcRegClient(ID devid, W kind)

[Parameters]

| | |
|---|---|
| devid | Device ID of device driver |
| kind | PC card type |

[Function values]

| | |
|---|---|
| E_OK | Normal completion |
| E_PAR | Parameter error (devid, kind) |
| E_LIMIT | Too many entries |
| E_NOEXS | Nonexistent (when kind == CK_NONE) |

[Description]

This function registers or deletes the device driver designated by devid (physical device ID) as a client.

When kind != CK_NONE, this function registers a driver for the PC card type designated by kind. When kind == CK_NONE, this function deletes the registration.
If the driver is for more than one kind of PC card, registration is made setting kind == CK_ANY.

If a device driver with one physical device ID must be associated with more than one kind of PC card, pcRegClient() must be called once for each kind. On the other hand, if a device driver is designed to be used with more than one PC card of the same kind, first it is necessary to get physical device Ids identifying each PC card, and then to register them by calling pcRegClient() for each ID.
Deleting registrations is done on the basis of each physical device ID. Thus, if there are several kinds of PC card associated with one physical device ID, they can all be deleted in one operation. Attempting to delete more than once returns an E_NOEXS error.

If a deleted driver was associated with an inserted PC card, that association is canceled.

### 5.8.2 Getting CIS tuple data

[Function]

INT pcGetTuple(ID cardid, W tuple, W order, UB* buf)

[Parameters]

| | |
|---|---|
| cardid | Card ID |
| tuple | Tuple code to acquire (0: get all) |

| | |
|---|---|
| order | Order of tuple code appearance (0: first tuple) |
| buf | Buffer for putting acquired tuple data<br>Must be at least MAX_TUPLESZ bytes.<br>buf[0]: Tuple code<br>buf[1]: Tuple link (n: 0 to 255)<br>buf[2 to n + 1]: Tuple data |

[Function values]

| | |
|---|---|
| > 0 | Size of acquired data in bytes (= n + 2 < = MAX_TUPLESZ) |
| = 0 | No such data |
| E_ID | Invalid card ID |
| E_MACV | Invalid address (buf) |

[Description]

This function gets the tuple data for the tuple code designated by the tuple argument at the position designated by order, from the CIS tuple data for the card associated with cardid, and puts the resulting data in the location pointed to by buf.
By incrementing the order parameter successively from 0, data for the same tuple code can be fetched in order. Alternatively, by setting tuple == 0, any tuple data at the position designated by order can be acquired.

### 5.8.3 Map memory space or IO space

[Function]

INT pcMap(ID cardid, W offset, W len, UW attr, VP *addr)

[Parameters]

| | | |
|---|---|---|
| cardid | Card ID | |
| offset | Start address of memory or IO space on card to be mapped | |
| len | Size in bytes of memory or IO space on card to be mapped | |
| attr | Map attribute | |
| | IO space mapping: | CA_IOMAP \| the following attributes<br>CA_16BIT, CA_IOCS16,<br>CA_WAIT0 to 3, CA_ZWAIT, CA_SPEED<br>CA_IONOCHK |
| | Memory space mapping: | the following attributes<br>CA_16BIT, CA_WPROTECT,<br>CA_WAIT0 to 3, CA_ZWAIT, CA_SPEED |
| | (* Designation of other invalid attributes (CA_xxx) is simply ignored.) | |
| addr | Start address of memory or IO space mapped in CPU (return code) | |

[Function values]

| | |
|---|---|
| > 0 | Map ID |
| E_ID | Invalid card ID |
| E_PAR | Parameter error (offset, len)<br>The designated IO space is in use. |
| E_LIMIT | Too many mappings |
| E_NOMEM | Not enough memory space or IO space for mapping |

E_MACV Illegal address (addr)

[Description]

This function maps the memory space or IO space designated by offset and len on the card associated with cardid to a corresponding memory space or IO space in the CPU based on the attribute designated in attr, returning the start address of the mapped area in *addr. A map ID identifying the map is returned as a function value.
Memory space mapping is performed only for common memory in the card; attribute memory cannot be mapped. The memory space is mapped in hardware-dependent units, usually of 4 KB.
The start address in the mapped CPU is allocated automatically in available memory space so as not to overlap with space already in use and is returned in *addr.
Because the maximum size of memory that can be mapped at one time is hardware-dependent, mapping should be done in small units, preferably 64 KB or smaller.

In IO space mapping, the IO start address in the CPU corresponding to the IO start address in the mapped card is returned in *addr. The same value as that designated in offset is returned if the IO space in the CPU and card match. If they do not match, a value different from offset is returned.

In designating map attributes, it is possible to designate wait cycles as follows, but a hardware-independent absolute designation is preferable.

Absolute designation (CA_SPEED)
Access speed designated in 50 ns units from 1 to 255
Wait cycle designation

| | |
|---|---|
| CA_ZWAIT | Zero wait |
| CA_WAIT0 | No additional wait (default) |
| CA_WAIT1 | Additional wait 1 |
| CA_WAIT2 | Additional wait 2 |
| CA_WAIT3 | Additional wait 3 |

When a card is ejected or the power is turned off, the card mapping becomes invalid. But because the mapping data is retained, the mapping can be restored with pcReMap().

In IO space mapping, normally the IO space is checked and registered using the hardware resource manager. This procedure can be skipped by designating the CA_IONOCHK attribute.

### 5.8.4 Remap memory space or IO space

[Function]

ER pcReMap(ID cardid, ID mapid)

[Parameters]

| | |
|---|---|
| cardid | Card ID |
| mapid | Map ID |

[Function values]

| | |
|---|---|
| E_OK | Normal |
| E_ID | Illegal card ID or map ID |

[Description]

This function restores the validity of the mapping designated by mapid for the card associated with cardid.

It is used when a card has been ejected and then reinserted, in order to restore the previous mapping.

Remapping is functionally the same as when a mapping is unmapped and then mapped again, with one difference: when mapping is made anew, memory mapping is not guaranteed to be made to the same CPU addresses.

### 5.8.5 Unmap memory space or IO space

[Function]

ER pcUnMap(ID mapid)

[Parameters]

| | |
|---|---|
| mapid | Map ID |

[Function values]

| | |
|---|---|
| E_OK | Normal |
| E_ID | Illegal map ID |

[Description]

Clears the mapping designated by mapid.

### 5.8.6 Read memory

[Function]

ER pcReadMem(ID cardid, W offset, W len, UW attr, VP buf)

[Parameters]

| | |
|---|---|
| cardid | Card ID |
| offset | Byte offset of read data |
| len | Byte length to be read |
| attr | Memory attribute<br>Attribute memory: CA_ATTRMEM \| The following attributes<br>CA_16BIT, CA_WPROTECT<br>Common memory: The following attributes<br>CA_16BIT, CA_WPROTECT,<br>CA_WAIT0 to 3, CA_ZWAIT, CA_SPEED<br>(* Designation of other invalid attributes (CA_xxx) is simply ignored.) |
| buf | Buffer for holding read data |

[Function values]

| | |
|---|---|
| E_OK | Normal |
| E_ID | Invalid card ID |
| E_PAR | Parameter error (offset, len) |
| E_MACV | Illegal address (buf) |

[Description]

This function reads len bytes of data, from common memory or attribute memory, starting at offset, in the card associated with cardid to buf.

Because attribute memory in a card is valid only at even-numbered addresses, only the bytes at even-numbered addresses are read as a contiguous byte array. Thus, offset must be designated as 1/2 the value of the actual offset on the card.

### 5.8.7 Write to memory

[Function]

ER pcWriteMem(ID cardid, W offset, W len, UW attr, VP buf)

[Parameters]

| | |
|---|---|
| cardid | Card ID |
| offset | Byte offset of write data |
| len | Byte length to be written |
| attr | Memory attribute<br>Attribute memory: CA_ATTRMEM \| The following attributes<br>CA_16BIT, CA_WPROTECT<br>Common memory: The following attributes<br>CA_16BIT, CA_WPROTECT,<br>CA_WAIT0 to 3, CA_ZWAIT, CA_SPEED<br>(* Designation of other invalid attributes (CA_xxx) is simply ignored.) |
| buf | Buffer containing the data to be written |

[Function values]

| | |
|---|---|
| E_OK | Normal |
| E_ID | Illegal card ID |
| E_PAR | Parameter error (offset, len) |
| E_MACV | Illegal address (buf) |

[Description]

This function writes len bytes of data from the memory area pointed to by buf to common memory or attribute memory, starting at offset, in the card associated with cardid.

Because attribute memory in a card is valid only at even-numbered addresses, only the bytes at even-numbered addresses are written as a contiguous byte array. Thus, offset must be designated as 1/2 the value of the actual offset on the card.

### 5.8.8 Get card status

[Function]

INT pcGetStat(ID cardid)

[Parameters]

| | |
|---|---|
| cardid | Card ID |

[Function values]

| | |
|---|---|
| > 0 | Card status (CardStat structure) |
| E_ID | Illegal card ID |

[Description]

This function gets the status of the card associated with cardid and returns the status information in a function value.

The returned value is the CardStat structure value cast on W.

### 5.8.9 Define/cancel interrupt handler

[Function]

INT pcDefInt(ID cardid, T_DINT *dint, INTVEC vec, UW par)

[Parameters]

| | |
|---|---|
| cardid | Card ID |
| dint | Interrupt handler definition information |
| vec | Interrupt vector (0: auto-assign) |
| par | Parameter passed to the interrupt handler |

[Function values]

| | |
|---|---|
| > = 0 | Normal (the value is the corresponding interrupt vector) |
| E_ID | Illegal card ID |
| E_PAR | Parameter error (dint->intatr, vec illegal) |
| E_BUSY | No available interrupt vector.vec already in use. |
| E_MACV | Illegal address (dint) |

[Description]

This function defines a handler for interrupts from the card associated with cardid. For the interrupt handler attribute, TA_HLNG must be designated.

The interrupt vector corresponding to the card is designated in vec. If vec == 0 is designated, an available interrupt vector is automatically assigned. Ordinarily vec == 0 is the setting used.
An error will occur if the designated vec cannot be assigned.

The interrupt vector assigned to the card is returned as a function value.

Interrupt resetting and related actions are handled automatically and need not be performed in the interrupt handler.
The interrupt handler has the following format.

```
VOID    inthdr(UW par)
```

If the card is used as an IO card, it is necessary to execute pcDefInt() setting dint ==

NULL, even if no interrupts from the card are used.

When dint.inthdr == NULL is set, the card generates interrupts, but because no interrupt handler is defined, the client device driver must itself define an interrupt handler corresponding to the interrupt vector returned as a function value.
In this case, the device driver must enable interrupts of the corresponding vector and reset interrupts that have been generated. Here, par is ignored.

Deleting an interrupt handler is executed setting dint == NULL.

When a card is ejected or the card power is turned off, the interrupt handler is automatically canceled and interrupts are disabled. Thus, when the card is reinserted or turned on, it is therefore necessary to execute pcDefInt() again for interrupt handler setting.

When a card is ejected or when an interrupt handler is deleted by calling pcDefInt (dint == NULL), the interrupt vector assigned to the card is also released. There is no guarantee that the same interrupt vector will be assigned the next time pcDefInt() is called.
Turning the power off, however, does not release the interrupt vector. In this case, the next time pcDefInt() is executed, assignment of the same interrupt vector as before is guaranteed.

These operations can be summarized as follows.

| | |
|---|---|
| dint == NULL | No interrupts are generated from the card.<br>Defining an interrupt handler with the setting dint.inthdr != NULL cancels the interrupt handler.<br>Interrupts of the corresponding vector are disabled. |
| dint.inthdr == NULL | Interrupts can be generated from the card.<br>No interrupt handler is defined for the corresponding vector.<br>Interrupts of the corresponding vector are disabled. |
| dint.inthdr != NULL | Interrupts can be generated from the card.<br>An interrupt handler is defined for the corresponding vector.<br>Interrupts of the corresponding vector are allowed. |

### 5.8.10 Control card power

[Function]

ER pcPowerCtl(ID cardid, UW power)

[Parameters]

| | | |
|---|---|---|
| cardid | Card ID | |
| power | Power control | |
| | CP_POWEROFF | Turn card off |
| | CP_POWERON | Turn card on |
| | CP_SUSPEND | SUSPEND state (power off) |
| | CP_RESUME | RESUME state (power on) |
| | CP_SUSRIRES | SUSPEND state (RI resume enabled) |

CP_POFFREQ(tm) Turn card off after tm seconds

[Function values]

| | |
|---|---|
| 1 | Turned card power on/off |
| 0 | Card power was already on/off (no change) |
| E_ID | Illegal card ID |
| E_PAR | Parameter error (power) |
| E_NOMDA | Card was removed (when CP_POWERON/CP_RESUME is designated) |
| E_IO | Card error (when CP_POWERON/CP_RESUME is designated) |

[Description]

Controls power supply to the card associated with cardid.

CP_POWEROFF: Turns card power off
The card can no longer be accessed.
The card map becomes invalid. (Mapping information is retained, so the map ID remains valid.) The interrupt handler for the card is canceled.
(The interrupt vector is not released.)

CP_SUSPEND: Turns card power off, putting it in SUSPEND state.
The processing is the same as for CP_POWEROFF.

CP_POWERON: Turns card power on.
After card reset processing, the state goes into a WAIT for READY state before returning. This WAIT period requires special attention.

If the card could not go to READY state, E_IO is returned and the card remains off.

If the card was ejected, E_NOMDA is returned and the client driver must perform the processing for card ejection. The card ID is already invalid at this point, so access to the card is not possible. But because the map ID is valid, it must be unmapped if no longer needed.
Note that in this case, the card event CE_EJECT is not notified.

Even if the card goes to READY state, the card is initialized, which requires the driver to perform configuration, mapping, interrupt handler definition, and related processing.

CP_RESUME: Turns card power on, and puts it in RESUME state.
The processing is the same as for CP_POWERON.

CP_SUSRISUS: For a modem card, puts the card in SUSPEND state while enabling resume by RI signal.
The processing is the same as for CP_POWEROFF, except that the card power remains on to enable resume.

CP_POFFREQ(tm): Turns the card power off after tm seconds.
When this request is accepted, the power is not yet turned off but function value 1 (actually turned card power on or off) is returned.
When CP_POWERON is executed, if the power is not yet off, 0 is returned. If the power is already off, 1 is returned.
tm is set in seconds, with a tolerance of -0 to +2 seconds or less.
If tm is set to 0,the behavior is the same as for CP_POWEROFF.

When a card is inserted, the power to the card is turned on and remains on until the driver turns it off by calling pcPowerCtl().

# 6. System Disk Driver

TEF040-S206-01.00.00/en

## 6.1 Applicable Devices

- This driver applies to most system disks in general, such as these.
  - PC card, ATA card, SRAM card, ATAPI CD-ROM card, or ATAPI hard disk inserted in a PCMCIA slot
  - RAM disk
  - ROM disk
  - USB storage device

- Only the following devices support subunits.
  - ATA card, ATAPI CD-ROM/hard disk, or USB storage device
  - Maximum subunits: 4

## 6.2 Device Names

The following device names are used.

| | |
|---|---|
| pca | Disk connected to a PC card slot |
| rda | ROM disk |
| uda | USB storage device |

## 6.3 Device-specific Functions

- PC card support
- USB storage device support
- Subunit support
- Support for event notification of removable media insertion and ejection
- Physical formatting support

  * Logical formatting is executed by the application software (using a format command and the like)

## 6.4 Attribute Data

The following attribute data is supported.

R Read-only

W Write-only

RW Read/write enabled

```
/* Disk attribute data numbers */
typedef enum {
        DN_DISKEVENT   = TDN_EVENT,     /* message buffer for event notification use */
        DN_DISKINFO    = TDN_DISKINFO, /* disk information */
        DN_DISKFORMAT= -100,            /* disk formatting */
        DN_DISKINIT    = -101,          /* disk initialization */
        DN_DISKCMD     = -102,          /* disk command */
        DN_DISKMEMADR= -103,            /* start address of memory disk space */
        DN_DISKPARTINFO= -104,          /* disk partition information */
        DN_DISKCHSINFO= -105,           /* disk CHS information */
        DN_DISKIDINFO  = -106           /* disk identification information */
} DiskDataNo;
```

DN_DISKEVENT: Set/get event notification message buffer ID (RW)

data: ID

Sets or gets the event notification message buffer ID.

DN_DISKINFO: Get/set disk information (R)

data: DiskInfo

```
typedef struct {
        DiskFormat format;          /* format type                */
        BOOL    protect: 1;         /* protection status          */
        BOOL    removable: 1;       /* removable or not */
        UW      rsv: 30;            /* reserved (0)               */
        W       blocksize;          /* block size in bytes        */
        W       blockcont;          /* total number of blocks     */
    } DiskInfo;
```

format: Format type

protect: Hardware write protection status

removable: Removable or not

blocksize: Total block size (in bytes)
Ordinarily 512 bytes

blockcont: Total number of blocks
For a subunit, the total blocks in the subunit

Gets disk information.

DN_DISKFORMAT: Format disk (W)

data: DiskFormat

```
typedef enum {
        DiskFmt_MEMINIT= -2,    /* initialize memory disk      */
        DiskFmt_MEM    = -1,    /* memory disk      */
        DiskFmt_STD    = 0,     /* only this setting if standard or HD     */
        DiskFmt_2DD    = 1,     /* 2 DD 720 KB      */
        DiskFmt_2HD    = 2,     /* 2 HD 1.44 MB     */
        DiskFmt_VHD    = 3,     /* floptical 20 MB  */
        DiskFmt_CDROM  = 4,     /* CD-ROM 640 MB          */
        DiskFmt_2HD12  = 0x12   /* 2HD 1.2 MB       */
} DiskFormat;
```

Starts physical formatting by writing the correct format type.

For a RAM disk, all blocks are filled with a fixed value, completely erasing all data on the disk.

Some devices and subunits do not allow this command.

After a device is physically formatted, the subunits will no longer exist. Processing requests for subunits will return an error until the disk has been initialized and subunits have been redefined.

DiskFmt_MEMINIT is a special designation to change the RAM disk size. Designating the following data directly after it initializes the RAM disk at the designated size. The block size (bytes), from 512 to 8192, must be a multiple of 512.

```
        W       blocksize;      /* block size in bytes     */
        W       blockcont;      /* total number of blocks  */
```

DN_DISKINIT: Initialize disk (W)

data: DiskInit

```
typedef enum {
        DISKINIT = 1
} DiskInit;
```

Resets and reregisters the designated device by writing DISKINIT. If the disk is partitioned, the partition information is read and each partition is registered as a subunit. This command is normally used to change disk partition information.

When the command is issued for a subunit, it has no effect.

DN_DISKCMD: Execute disk command (W)

data: DiskCmd

```
typedef struct {
        B       clen;       /* ATAPI command length    */
        UB      cdb[12];    /* ATAPI command           */
        W       dlen;       /* data length     */
        UB      *data;      /* data address    */
} DiskCmd;
```

Executes the written ATAPI command. The length of the ATAPI command is fixed at 12 bytes. Setting clen to 12 indicates a read command and setting it to 12 + 0x80 indicates a write command.

DN_DISKMEMADR: Get start address of memory disk space (R)

data:VP

Gets the start address (logical address) of memory used as a disk.

The contiguous physical memory space starting at this address and extending for the disk size obtained by DiskInfo (blocksize * blockcont bytes) is the memory available as disk space.

Arbitrary access can be made only when the device has been opened by tk_opn_dev(). After the device is closed by tk_cls_dev(), it can no longer be accessed. If a disk is closed and then re-opened, the start address must be retrieved again and access made using that address.

Device that do not allow direct memory access return the error code E_NOSPT.

DN_DISKPARTINFO: Get disk partition information (R)

data: DiskPartInfo

```
typedef enum {                      /* disk system ID                       */
        DSID_NONE       = 0x00,
        DSID_DOS1       = 0x01,
        DSID_BTRON_X    = 0x03,  /* XENIX, but interpreted as BTRON    */
        DSID_DOS2       = 0x04,
        DSID_DOSE       = 0x05,
        DSID_DOS3       = 0x06,
        DSID_HPFS       = 0x07,
        DSID_FS         = 0x08,
        DSID_AIX        = 0x09,
        DSID_OS2        = 0x0A,
        DSID_WIN95      = 0x0B,
        DSID_WIN95L     = 0x0C,
        DSID_DOS3L      = 0x0E,
        DSID_DOS3E      = 0x0F,
        DSID_BTRON      = 0x13,
        DSID_VENIX      = 0x40,
        DSID_CPM1       = 0x52,
```

```
        DSID_UNIX       = 0x63,
        DSID_NOVELL1    = 0x64,
        DSID_NOVELL2    = 0x65,
        DSID_PCIX       = 0x75,
        DSID_MINIX1     = 0x80,
        DSID_MINIX2     = 0x81,
        DSID_LINUX1     = 0x82,
        DSID_LINUX2     = 0x83,
        DSID_AMOEBA     = 0x93,
        DSID_BSDI       = 0x9F,
        DSID_386BSD     = 0xA5,
        DSID_CPM2       = 0xDB,
        DSID_DOSSEC     = 0xF2
} DiskSystemId;

typedef struct {
DiskSystemId    systemid;       /* system ID                */
        W       startblock;     /* starting block number    */
        W       endblock;       /* ending block number      */
} DiskPartInfo;
```

systemid: System ID of partition
startblock: Starting absolute block number of partition
endblock: Ending absolute block number of partition
= startblock + DiskInfo.blockcont - 1

Gets partition information of a subunit.
Calling this function for a physical device returns an error.

DN_DISKCHSINFO: Get disk CHS information (R)
data: DiskCHSInfo

```
typedef struct {
        W       cylinder;       /* total number of cylinders */
        W       head;           /* heads per cylinder       */
        W       sector;         /* sectors per head         */
} DiskCHSInfo;
```

Gets information for the disk cylinder (C), head(H), and sector (S).

In the case of a floppy disk, a cylinder refers to a track. With a RAM disk, C = 1, H = 1, and S = total blocks.

Normally C * H * S = DiskInfo.blockcount. Depending on the limitations for C, H, and S values, however, in some cases C * H * S < DiskInfo.blockcount.

DN_DISKIDINFO: Get disk identification information (R)
data: UB[]

Gets disk identification information. The details depend on the type of disk.

With an ATA disk, the data obtained from the disk by the IDENTIFY command is the 48 H (96 bytes) of data arranged in following sequence (in H units).

| Position | Original position | Description |
|---|---|---|
| 0: | [ 0] | General configuration bit |
| 1: | [ 1] | Number of logical cylinders |
| 2: | [49] | Capabilities |
| 3: | [ 3] | Number of logical heads |
| 4: | [80] | Major version number |
| 5: | [53] | 7-0: Validity<br>12: DMA support, 15: MSN support |
| 6: | [ 6] | Number of logical sectors per logical track |
| 7: | [54] | Number of current logical cylinders |
| 8: | [55] | Number of current logical heads |
| 9: | [56] | Number of current logical sectors per track |
| 10-19: | [10-19] | Serial number (20 ASCII characters) |
| 20-21: | [60-61] | Total number of user-addressable sectors |
| 22: | [63, 88] | 7-0: Multiword DMA Mode Supported<br>15-8: Ultra DMA Mode Supported |
| 23-26: | [23-26] | Firmware revision (8 ASCII characters) |
| 27-46: | [27-46] | Model number (40 ASCII characters) |
| 47: | [47] | Maximum number of sectors on R/W MULTIPLE cmds |

## 6.5 Device-specific Data

The following device-specific data is supported.

| | |
|---|---|
| Data number ( 0- ): | Disk block number |
| Data count: | Read/write block count |

In the case of a physical device (unit), the block number matches the physical block number. With a logical device (subunit), however, the block number is the relative block number (0- ) in the partition.

## 6.6 Event Notification

```
typedef struct {
        T_DEVEVT_ID     h;      /* standard header (with device ID)     */
        UW              info;   /* additional information       */
} DiskEvt;
```

| | | |
|---|---|---|
| h.evttyp: | TDE_MOUNT | Disk or card inserted |
| | TDE_EJECT | Disk or card ejected |
| | TDE_ILLEJECT | Disk or card illegally ejected |
| | TDE_ILLMOUNT | Disk or card illegally mounted |
| | TDE_REMOUNT | Disk or card removed |

Notifies of disk mount/demount events.

TDE_ILLEJECT notifies that a disk was illegally ejected while still open.

TDE_ILLMOUNT notifies that after illegal ejection, the reinserted disk is different from the one that was illegally ejected.

TDE_REMOUNT notifies that after illegal ejection, the reinserted disk is the same one that was illegally ejected and that a normal state has been restored.

info is a bit array indicating the open states of the physical unit and subunits when the event occurred.

When (info & (1 << N)) != 0 , the physical unit or subunit N is open.

Here, N = 0 means a physical unit (e.g., pca)
N = 1 - means a subunit (e.g., pca0 - )

* For TDE_MOUNT and TDE_EJECT, because no unit/subunit is open, info is always 0.

Event notification is made only to a physical unit, not to a subunit.

The speed of response to event notification is driver-dependent.

## 6.7 Error Codes

See the section on device management functions in the T-Kernel specification.

The IO error detail codes are as follows.

| | |
|---|---|
| E_IO \| 0x0000 | Aborted |
| E_IO \| 0x0001 | Interrupt timeout |
| E_IO \| 0x0002 | Media error |
| E_IO \| 0x0003 | Hardware error |
| E_IO \| 0x0010 | Command busy error |
| E_IO \| 0x0011 | Data busy error |
| E_IO \| 0x0012 | Not ready error |
| E_IO \| 0x8xxx | ATA/ATAPI disk operation error |

ATA: 1000 0000 SSSS SSSS

S: error status = DF UNC MC IDNF MCR ABRT TKO AMNF

ATAPI: 1QQQ KKKK CCCC CCCC

K: Sense Key (SK) != 0

C: Additional Sense Code (ASC)

Q: Additional Sense Code Qualifier (ASCQ)

## 6.8 Partition Information

Following the ATA specification, the initial disk block (the master boot record) contains the following partition information.

```
typedef struct {
        UB      BootInd;        /* boot indicator      */
        UB      StartHead;      /* starting head number        */
        UB      StartSec;       /* starting sector number      */
        UB      StartCyl;       /* starting cylinder number    */
        UB      SysInd;         /* system indicator  */
        UB      EndHead;        /* ending head number          */
        UB      EndSec;         /* ending section number       */
        UB      EndCyl;         /* ending cylinder number      */
        UH      StartBlock[2];  /* relative starting sector number         */
        UH      BlockCnt[2];    /* sector count        */
} PartInfo;

typedef struct {
        VB              boot_prog[0x1be];       /* boot program      */
        PartInfo        part[MAX_PARTITION];    /* partition information */
        UH              signature;              /* signature         */
} DiskBlock0;
```

- Data of 2 bytes or more is in little endian format.

- The system indicator becomes the DiskSystemId in disk partition information.

- Cylinder/head/sector numbers are ignored in the case of a partition, using only the StartBlock[2] and BlockCnt[2] information.

- Part[].StartBlock cannot be a word boundary, so some special care is required depending on the machine.

## 6.9 T-Engine/SH7727 Related Information (Reference)

### 6.9.1 Supported devices

The following devices are currently supported.

PC Card:

Device name: pca

ATA cards, CF cards with a PC card adapter, and related cards are supported.

ROM disk:

Device name: rda

A disk in system ROM

USB storage device:

Device name: uda

Floppy disks, CD-ROMs, card readers and writers, and the like conforming to USB mass storage device class specifications are supported.

### 6.9.2 DEVCONF file-related entries

HdSpec HD specification

xxNI xxxx xxxx xxxx xxxx

I : Automatically check for insertion of ejectable disk media

N: Automatic ejection disabled for ejectable disk media

HdChkPeriod interval (ms)

Ejectable disk (including CD-ROM) check interval

(Default 3000)

### 6.9.3 Master boot record access function

A special function is provided for accessing the master boot record of an ATA disk (the physical unit only).

Attribute record number: -999999 (R)

```
data:   UW          magic
        DiskBlock0  mboot;
```

magic = CH4toW('M','B','R','R'): read master boot record
CH4toW('M','B','R','W'): write to master boot record
(write in read processing)

- The behavior is not guaranteed if a partition in use is changed by writing to the master boot record.

### 6.9.4 Partitions (subunits)

An ATA disk always has four subunits defined in order to respond to dynamic changes in partition information. Attempts to open an empty subunit result in an E_NOMDA error.

On a CD-ROM, the following fixed allocation is made of partitions (subunits). Partitions 2 and 3 do not exist in some cases.

Partition 1: Entire CD-ROM (same as physical unit)

Partition 2: Boot record (2 HD FD boot image)

Partition 3: BTRON volume partition

- The El Torito standard format is used for the boot record.
- The TRON volume partition uses an original BTRON format.

### 6.9.5 CHS information

Basically, information matching the BIOS settings is returned as CHS information. The total disk capacity as calculated from the CHS information therefore is generally less than the actual disk capacity.

| | |
|---|---|
| Cylinders (C) | Max. 1023 |
| Heads (H) | Max. 255 |
| Sectors (S) | Max. 63 |

The maximum number of cylinders is 1024, but normally the last cylinder is not used.
The maximum number of heads is 256, with 255 being the usual number.

CHS information is determined as follows. It is made to match the BIOS settings as much possible, but in some cases there will be differences.

1. From the physical CHS information, the CHS information is calculated as follows so as to keep within the above limitations. (pC; pH; pS: physical CHS information)

```
T = pC * pH * pS; C = pC; H = pH; S = pS;
while (C > 1024) {C >>= 1; H <<= 1;}
```

if (S > 63) S = 63;
if (H > 255) H = 255;
C = T / H / S;
if (C > 1023) C = 1023;

2. When a partition is set, CHS information is calculated from the partition information.
S = last sector in partition;
H = last head in partition +1;
C = T / H / S - 1;

# 7. eTRON SIM Driver

TEF040-S207-01.00.00/en

## 7.1 Applicable Devices

- Applicable devices are equipped with an ISO 7816 contact interface for communication with eTRON SIM chips.

## 7.2 Device Name

- The device name is "etsim".

## 7.3 Device-specific Functions

- Sending and receiving packets with eTRON SIM chips.
- Resetting eTRON SIM chips. Getting the ATR.
- Setting the communication mode of devices with an ISO 7816 interface.

## 7.4 Attribute Data

The following attribute data is supported.

```
typedef enum {
        DN_ETSIMATR             = -100,
        DN_ETSIMRESET           = -101,
        DN_ETSIMMODE            = -102,
} ETSIMDataNo;
```

DN_ETSIMATR: Get ATR data(R)
buf: UB[size]

Gets ATR data produced when the eTRON SIM is reset.

DN_ETSIMRESET: Reset (W)
buf: not used

Resets eTRON SIM chips.

DN_ETSIMMODE: Set communication mode (W)
buf: SimMode

```
typedef struct {
        UW      baud;           /* 9600, 19200, 38400, 76800, 155270, 310539 */
        UW      mode[3];        /* For expansion. Currently to be set at '0'. */
```

```
} SimMode;
```

Sets the communication mode of devices with an ISO 7816 interface.
When the device is open or reset, the default is 9600 bps and T=0.
If unsupported modes are designated, an error is returned.
This function is only for setting up interface devices. Executing it does not start packet exchanges with eTRON SIM chips.

## 7.5 Device-specific Data

start: Fixed at 0

buf: UB[size]

During write Sends a command byte array of size bytes from the area of buf.

During read Stores the response byte array received in reply to the command in buf.

## 7.6 Event Notification

None

## 7.7 Error Codes

See the section on device management functions in the T-Kernel specification.

# 8. Clock Driver

TEF040-S208-01.00.00/en

## 8.1 Applicable Devices

- A real-time clock (RTC) or other device for time management.

## 8.2 Device Name

- The device name is "CLOCK".

## 8.3 Device-specific Functions

- Getting and setting the time of the real-time clock
- Supporting hardware-dependent functions
  Scheduled automatic power-on
  Non-volatile register access
  And other functions

## 8.4 Attribute Data

The following attribute data is supported.

R Read-only
W Write-only
RW Read/write enabled

```
/* CLOCK data numbers */
typedef enum {
    /* Common attributes */
    DN_CKEVENT                  = TDN_EVENT,
    /* Device-specific attributes */
    DN_CKDATETIME               = -100,
    DN_CKAUTOPWON               = -101,
    /* Hardware-dependent functions */
    DN_CKREGISTER               = -200
} ClockDataNo;
```

DN_CKDATETIME is mandatory.

Functions -101 to -199 have been standardized to be essentially hardware-independent. Whether or not they are supported, however, depends on the hardware.

Functions -200 and the following functions are largely hardware-dependent and have little

relation to clock functions. These are not standardized.

If a requested data number is not supported, error code E_NOSPT is returned.

DN_CKEVENT: Set/get event notification message buffer ID (RW)
data: ID

DN_CKDATETIME: Set/get current time (RW)
data: DATE_TIM

```
typedef struct {
        W       d_year;         /* offset from year 1900 (85- )         */
        W       d_month;        /* month (1 to 12, 0)                   */
        W       d_day;          /* day (1 to 31)                        */
        W       d_hour;         /* hour (0 to 23)                       */
        W       d_min;          /* minute (0 to 59)                     */
        W       d_sec;          /* second (0 to 59)             */
        W       d_week;         /* week (1 to 54) *Not used             */
        W       d_wday;         /* day of week (0 to 6, 0=Sunday)       */
        W       d_days;         /* day of year (1 to 366) *Not used     */
} DATE_TIM;
```

Sets or gets the current time (local time) for a real-time clock.
d_wday is not checked for an incorrect day-of-week setting. The accuracy of the retrieved day of the week is therefore not guaranteed. On hardware that does not support the day-of-week setting, the value is undefined.
d_week and d_days are not used. These values are undefined.

DN_CKAUTOPWON: Set/get automatic power-on time (RW)
data: DATE_TIM

Sets or gets the automatic power-on time (local time) for a real-time clock.
When d_year = 0 is set (in which case other values are ignored), automatic power-on is canceled. If a time in the past is set, this also effectively cancels the feature. d_week, d_wday, and d_days are not used, in principle. When the time is set, however, d_wday must be set correctly.

Non-standard functions: implementation-dependent

Depending on the hardware, support for settings such as these is conceivable.

Ex.: Turn the power on at 10:00 every Monday

```
d_year = d_month = d_day = -1;          /* ignored */
d_hour = 10; d_min = d_sec = 0;         /* 10:00:00 */
d_wday = 1;                             /* Monday */
```

DN_CKREGISTER: Read/write in non-volatile registers (RW)

data: CK_REGS

```
typedef struct {
        W               nreg;           /* number of registers to access */
        struct ck_reg {
                W       regno;          /* applicable register number */
                UW      data;           /* applicable data */
        } c[1];
} CK_REGS;
```

Reads or writes in the non-volatile register provided for a real-time clock.
In a write operation, data is written to the register designated by regno.
In a read operation, the contents of the register designated by regno are read and set in data.
The write and read operation is performed for nreg number of registers. (The use of nreg and regno as input parameters for a read operation as well distinguishes this from other general approaches.)
data is valid only for the bit width of the register. In the case of an 8-bit register, for example, the low 8 bits of data are written to the register. In a read operation, 0 is set in the high bits of data.

## 8.5 Device-specific Data

None

## 8.6 Event Notification

```
typedef struct t_devevt_id {
        TDEvtTyp        evttyp;     /* event type */
        ID              devid;      /* device ID */
        /* Information for the event type is then appended here. */
} T_DEVEVT_ID;

typedef T_DEVEVT_ID  ClockEvt;
```

DE_CKPWON is set in kind.

DE_CKPWON: Automatic power-on notification

Event notification occurs when it is the automatic power-on time set in DN_CKAUTOPWON.

- If the power is already on (RESUME state) at the designated time:
  Only event notification occurs.
- If the designated time comes during SUSPEND state:
  After resuming, event notification occurs.
- If power is off (fully stopped) when the designated time comes:
  Reboots; no event notification occurs.

* Depending on the hardware, not all these functions may be enabled.

## 8.7 Error Codes

See the section on device management functions in the T-Kernel specification. There are no special error codes specific to the clock driver.

## 8.8 T-Engine/SH7727 Related Information (Reference)

Only DN_CKDATETIME is supported as device-specific data.
Event notification is not supported.

# 9. Keyboard and Pointing Device Driver

TEF040-S209-01.00.01/en

## 9.1 Applicable Devices

- The standard T-Engine keypad and touch panel as well as other input devices.
- On a T-Engine supporting a USB host, a keyboard and mouse conforming to the HID (Human Interface Device) class.
- This is one driver, not separate keyboard (KB) and pointing device (PD) drivers.

## 9.2 Device Name

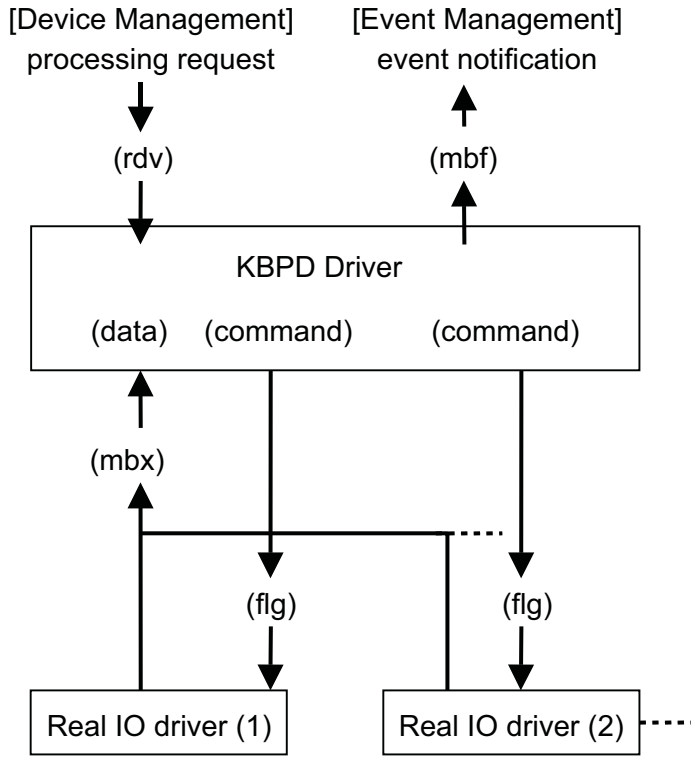
- The device name is "kbpd".

## 9.3 Device-specific Functions

- Key event notification (key on/off)
- Key valid interval / invalid interval / handling of simultaneously pressing interval
- Temporary shift / temporary shift specification / simple lock
- PD simulation
- Meta key state management
- Key code conversion
- PD event notification (buttons, movement)
- PD valid interval / invalid interval / timeout handling
- PD attributes and range changes
- The KBPD driver is not involved in on-screen display of the pointer.

## 9.4 Driver Design

To support a variety of KB and PD devices, the KBPD driver structure is separate from real IO drivers. Thus, this KBPD driver is not directly dependent on the KB/PD device and does not perform IO access.



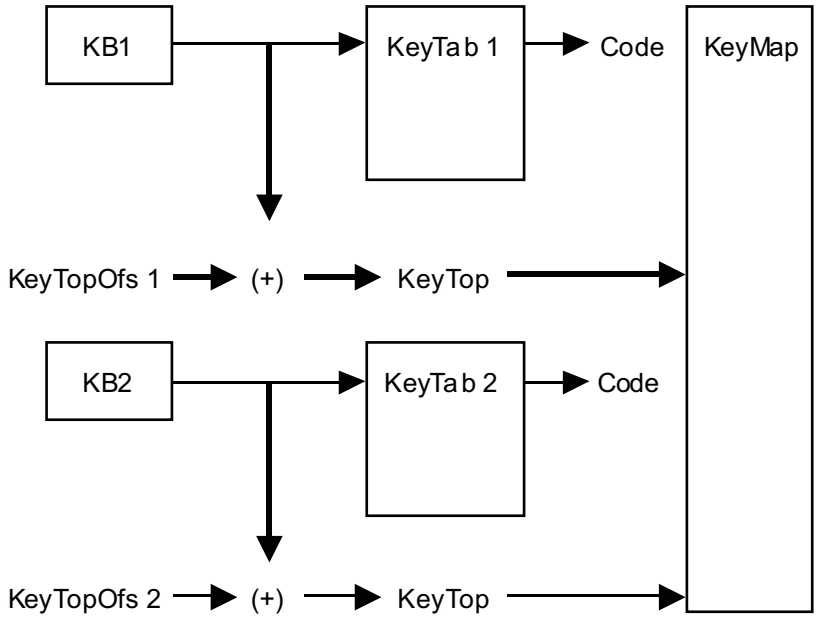

| | |
|---|---|
| (rdv) | Rendezvous port for accepting processing requests |
| (mbf) | Message buffer for event notification |
| (mbx) | Mailbox for data |
| (flg) | Event flag used with commands |

A real IO driver sends device KB/PD operations to a data mailbox in a fixed format. Operations are sent by multiple real IO drivers to the same mailbox.

The KBPD driver can send a command (one word) using an event flag provided by a real IO driver. If there are multiple IO drivers, commands are sent to all of them. These commands are used for functions such as controlling the LED on a keyboard.

## 9.5 Multiple Keyboard Support

The specification allows for connecting two or more different types of keyboards and using them at the same time.




A separate key table (KeyTab) is kept for each type of keyboard. The keyboard and key table are associated by a keyboard ID. When two or more keyboards of the same type are connected, they are associated with the same key table.

A key table is created to map to the keytop codes generated by the keyboard.

A key map (KeyMap) contains bits that are on or off for the corresponding keytop codes, which are the keytop codes generated by the keyboard plus an offset value (KeyTopOfs). These keytop codes with offset values are notified to higher levels (event management).

KeyTopOfs can be used to prevent overlapping of the keyboard codes from different types of keyboards. If such overlapping would not cause problems, however, KeyTopOfs can be set with partial or full overlapping.

## 9.6 Attribute Data

The following attribute data is supported.

```
R       Read-only
W       Write-only
RW      Read/write enabled

/* KBPD attribute data numbers */
typedef enum {
        /* Common attributes */
        DN_KPEVENT              = TDN_EVENT,
        /* Device-specific attributes */
        DN_KPINPUT              = -100,
        DN_KPSTAT               = -101,
        DN_KEYMAP               = -102,
        DN_KEYTAB               = -103,
        DN_KEYMODE              = -104,
        DN_PDMODE               = -105,
        DN_PDRANGE              = -106,
        DN_PDSIM                = -107,
        DN_PDSIMINH             = -108,
        DN_KEYID                = -109,
        DN_KPMETABUT            = -110,
        /* Keyboard definition 1 (-200 to -327) */
        DN_KEYDEF_S             = -200,
        DN_KEYDEF_E             = -327,
        /* Keyboard definition 2 (-400 to -527) */
        DN_KEYDEF2_S            = -400,
        DN_KEYDEF2_E            = -527
} KPDataNo;

DN_KPEVENT: Set/get event notification message buffer ID (RW)
        data:   ID

DN_KPINPUT: Get input mailbox ID (R)
        data:   ID

        Gets the mailbox ID where a real IO driver sends keyboard input.
        * Created by the KBPD driver at initialization.

DN_KPSTAT: Set/get KB/PD state (RW)
        data:   KPStat

        typedef struct {
```

```
        H       xpos;   /* X coordinate position        */
        H       ypos;           /* Y coordinate position        */
        MetaBut stat;           /* meta button state  */
} KPStat;
typedef enum {
        HiraMode        = 0,    /* Japanese hiragana            */
        AlphaMode       = 1,    /* alphabet (lower case)        */
        KataMode        = 2,    /* Japanese katakana            */
        CapsMode        = 3,    /* alphabet (upper case)        */
} InputMode;

typedef enum {
        PdSim_Off       = 0,    /* PD simulation off */
        PdSim_Std       = 1,    /* standard PD simulation       */
        PdSim_MainBut   = 2,    /* main button PD simulation */
        PdSim_TenKey    = 3     /* numeric keypad PD simulation */
} PdSimMode;

typedef struct {
#if BIGENDIAN
        UW      rsv1: 8;        /* reserved (0)                 */
        UW      pdsim: 2;       /* PD simulation (PdSimMode) */
        UW      nodsp: 1;       /* no pointer display           */
        UW      rsv2: 3;        /* reserved (0)                 */
        UW      kbsel: 1;       /* keyboard selection           */
        UW      han: 1;         /* half-width mode  */

        UW      tcmd: 1;        /* command temporary shift  */
        UW      text: 1;        /* extended temporary shift   */
        UW      trsh: 1;        /* right shift temporary shift   */
        UW      tlsh: 1;        /* left shift temporary shift     */

        UW      lcmd: 1;        /* command simple lock      */
        UW      lext: 1;        /* extended simple lock       */
        UW      lrsh: 1;        /* right shift simple lock       */
        UW      llsh: 1;        /* left shift simple lock         */

        UW      cmd: 1;         /* command shift            */
        UW      ext: 1;         /* extended shift             */
        UW      rsh: 1;         /* right shift                */
        UW      lsh: 1;         /* left shift                 */

        UW      mode: 2;        /*key input mode (InputMode) */
```

```
            UW      sub: 1;         /* sub button                   */
            UW      main: 1;        /* main button                  */
#else
            UW      main: 1;        /* main button                  */
            UW      sub: 1;         /* sub button                   */

            UW      mode: 2;        /*key input mode (InputMode) */

            UW      lsh: 1;         /* left shift                   */
            UW      rsh: 1;         /* right shift                  */
            UW      ext: 1;         /* extended shift               */
            UW      cmd: 1;         /* command shift                */

            UW      llsh: 1;        /* left shift simple lock       */
            UW      lrsh: 1;        /* right shift simple lock      */
            UW      lext: 1;        /* extended simple lock         */
            UW      lcmd: 1;        /* command simple lock          */

            UW      tlsh: 1;        /* left shift temporary shift   */
            UW      trsh: 1;        /* right shift temporary shift  */
            UW      text: 1;        /* extended temporary shift     */
            UW      tcmd: 1;        /* command temporary shift      */

            UW      han: 1;         /* half-width mode              */
            UW      kbsel: 1;       /* keyboard selection           */
            UW      rsv2: 3;        /* reserved (0)                 */
            UW      nodsp: 1;       /* no pointer display           */
            UW      pdsim: 2;       /* PD simulation (PdSimMode)*/
            UW      rsv1: 8;        /* reserved (0)                 */
#endif
} MetaBut;
```

stat: The current meta key state and button state. A write operation is ignored. The values in cmd, ext, rsh, and lsh also reflect the temporary shift/simple lock tate. That is, if tcmd or lcmd is 1, cmd will always be 1.

nodsp: 1 if a pointer is not displayed (during touch panel input, etc.)

han: 1 during half-width input mode

kbsel: Keyboard (key table) selection state
0: keyboard definition 1 (for kana input)
1: keyboard definition 2 (for alphabet input)

xpos:
ypos: current PD position
If a value outside the PD range is written, it is corrected to fall inside the range.

A change in the position written here triggers event notification.

DN_KPMETABUT:Set meta key/button state (W)

Data: MetaBut[2]

Changes the current meta key and button state as follows.

new state = current state & MetaBut[0] | MetaBut[1]

When the state changes as a result, event notification occurs. The behavior is undefined if a logically inconsistent state setting is made.

DN_KEYMAP: Get key map (R)

data: KeyMap

```
#define KEYMAX    256
typedef UB        KeyMap[KEYMAX/8];
```

The current key state.
When a key is pressed, the corresponding keytop code bit sequence (0 to 255) is set to 1; when a key is not pressed, the bit is cleared to 0.
If more than one keyboard is connected and keys with the same keytop code (with KeyTopOfs added) are pressed simultaneously, the KeyMap state is undefined.

DN_KEYTAB: Get/set Key table (RW)

data: KeyTab

```
typedef struct {
        W       keymax;         /* actual maximum key count             */
        W       kctmax;         /* actual number of conversion tables   */
        UH      kctsel[KCTSEL]; /* conversion table number    */
        UH      kct[KCTMAX];    /* conversion table (variable length)   */
} KeyTab;

#define KCTSEL          64
#define KCTMAX          4000
```

The theoretical maximum of KCTMAX is 256 * 64. Because this many tables is rarely needed, the default is set at 4000.

keymax: Actual maximum number of keys (1 to KEYMAX)
kctmax: Actual number of conversion tables (1 to KCTSEL)
kctsel: Conversion table number for meta key states Includes conversion table numbers 0 to (kctmax -1) corresponding to arrays indexed by the value of MetaKey.
MetaKey: The 6-bit value of CERLKA
That is, the value of (MetaBut >> 2) & (KCTSEL-1)
kct: The conversion table itself, actually keymax * kctmax elements
The converted key codes are obtained as follows:

kct[keymax * kctsel[MetaKey] + keytop]
keytop: keytop code (0 to KEYMAX)

In read/write operations, the size of the key table is the data length.
DN_KEYTAB is retained for backward compatibility. From now on DN_KEYDEF should be used. DN_KEYTAB is used to set or get the key table for keyboard definition 1 corresponding to the keyboard ID DN_KEYID.

DN_KEYMODE: Get/set key mode (RW)
data: KeyMode

```
typedef struct {
        MSEC    ontime;   /* valid on time                    */
        MSEC    offtime;  /* valid off time                   */
        MSEC    invtime;  /* invalid time                     */
        MSEC    contime;  /* interval for pressing concurrently */
        MSEC    sclktime; /* short click interval                */
        MSEC    dclktime; /* double-click interval            */
        BOOL    tslock;   /* temporary shift specification */
} KeyMode;

#define KB_MAXTIME      10000
```

ontime: valid time until key is considered on
offtime: valid time until key is considered off
invtime: invalid time after key is off
contime: permitted time for pressing concurrently with meta key
sclktime: click interval until temporary shift is valid
dclktime: double-click interval until simple lock is valid

When values are written, they are adjusted within the range of 0 to KB_MAXTIME. Negative values are not changed.

tslock: temporary shift specification
TRUE: temporary shift specification
FALSE: normal

Because sclktime and dclktime are valid even with temporary shift specifications, ordinarily sclktime is set to the maximum value and dclktime to 0 in upper software.

DN_PDMODE: PD mode (RW)
data: PdMode

```
typedef struct {
        MSEC    ontime;   /* valid on time    */
        MSEC    offtime;  /* valid off time   */
        MSEC    invtime;  /* invalid time     */
```

```
        MSEC    timeout;  /* timeout interval   */
        PdAttr  attr;     /* PD attribute       */
} PdMode;

typedef struct {
#if BIGENDIAN
        UW      rsv1: 17;         /* reserved (0)       */
        UW      wheel: 1;         /* wheel              */
        UW      qpress: 1;        /* quick press        */
        UW      reverse: 1;       /* left-right reversal */
        UW      accel: 3;         /* acceleration       */
        UW      absolute: 1;      /* absolute/relative  */
        UW      rate: 4;          /* scan rate          */
        UW      sense: 4;         /* sensitivity        */
#else
        UW      sense: 4;         /* sensitivity        */
        UW      rate: 4;          /* scan rate          */
        UW      absolute: 1;      /* absolute/relative  */
        UW      accel: 3;         /* acceleration       */
        UW      reverse: 1;       /* left-right reversal */
        UW      qpress: 1;        /* quick press        */
        UW      wheel: 1;         /* wheel              */
        UW      rsv1: 17;         /* reserved (0)       */
#endif
} PdAttr;

#define PD_MAXTIME      10000
```

ontime: valid time until PD button is considered on
offtime: valid time until PD button is considered off
invtime: invalid time after PD button is off
timeout: PD button timeout interval

When values are written, they are adjusted within the range of 0 to PD_MAXTIME. Negative values are not changed.

attr.wheel: wheel (0: invalid, 1: valid)
attr.qpress: quick press (0: invalid, 1: valid)

attr.reverse: left-right reversal (0: right-hand mode, 1: left-hand mode)
In left-hand mode:
PD main and sub buttons are reversed.
XY coordinate signs are inverted.

attr.accel: pointer movement acceleration (valid only in relative coordinates mode)
0: no acceleration
1 to 7: acceleration, small to large

attr.absolute: 0: relative coordinates mode 1: absolute coordinates mode
If the input device operates by relative coordinates, setting absolute coordinates mode has no effect.
attr.rate: PD scan rate 0 to 15 (minimum: 0)
This is dependent on the input device, so the data is passed to the real IO driver.
attr.sense: PD sensitivity 0 to 15 (minimum: 0)
For a relative coordinates device, this value sets the ratio of PD position to input movement.
With an absolute coordinates device, it sets the ratio of the PD position to input movement around the center coordinates as a fixed point.
This is dependent on the input device, so the data is passed to the real IO driver.

DN_PDRANGE: Get/set PD range (RW)
data: PdRange

```
typedef struct {
        H       xmax;           /* Maximum x coordinate value      */
        H       ymax;           /* Maximum y coordinate value      */
} PdRange;
```

The PD position cannot lie beyond this range.
If writing this value results in the current PD position lying outside the range, the position is corrected to fall within the range and event notification occurs.

DN_PDSIM: Get/set PD simulation (RW)
data: W
Designates the rate of movement in the range of 0 to 15.
0: PD simulation disabled
1 to 15: Rate of movement (minimum: 1)

DN_PDSIMINH: Temporarily disable/re-enable PD simulation (RW)
data: BOOL

When TRUE, PD simulation is temporarily disabled.
When FALSE, the temporary disable is canceled.

DN_KEYID: Get/set keyboard ID (RW)
data: UW

Sets/gets the keyboard ID of the default keyboard.
This value identifies the default keyboard that is the object of DN_KEYTAB.
In the initial state, the keyboard connected first is the default keyboard.

DN_KEYDEF (kid): Get/set keyboard definition 1 (RW)

DN_KEYDEF2 (kid): Get/set keyboard definition 2 (RW)

```
#define DN_KEYDEF(kid)      (DN_KEYDEF_S - (kid))
#define DN_KEYDEF2(kid)   (DN_KEYDEF2_S - (kid))
data:       KeyDef

typedef struct {
        W       keytopofs;      /* offset   */
        KeyTab  keytab;         /* key table (variable length)  */
} KeyDef;
```

Sets the key table and keytop code offset value for the keyboard designated by keyboard ID (kid). Alternatively, gets the current setting.
Keyboard definition 1 is for kana input, and keyboard definition 2 is for alphabet input.
Setting keytab.keymax = 0 clears the keyboard definition for that keyboard ID.

```
Keyboard ID (0x00 to 0x7f)
#define KID_unknown     0x00    /* undefined keyboard */
#define KID_TRON_JP     0x01    /* TRON Japanese keyboard */
#define KID_IBM_EG      0x40    /* IBM 101-key English keyboard */
#define KID_IBM_JP      0x41    /* IBM 106-key Japanese keyboard */
```

## 9.7 Device-specific Data

None

## 9.8 Event Notification

Event notification occurs as follows for key events (KeyEvt) and PD events (PdEvt).

```
typedef struct {
        T_DEVEVT        h;      /* standard header */
        UH              keytop; /* keytop code     */
        UH              code;   /* character code  */
        MetaBut         stat;   /* meta key state  */
} KeyEvt;

h.evttyp:  TDE_KEYDOWN  Key down
           TDE_KEYUP    Key up
           TDE_KEYMETA  Meta key state change
```

TDE_KEYDOWN and TDE_KEYUP event notification occurs for all keys except meta keys and unused keys (keycode = 0) when each key is pressed or released.

TDE_KEYMETA notification occurs for any of the following meta key state changes.

```
tcmd - tlsh     temporary shift state
lcmd - llsh     simple lock state
cmd - lsh       shift state
```

| | |
|---|---|
| mode | key input mode |
| han | half-width mode |
| kbsel | keyboard selection |

keytop: A code indicating key position (keytop code).

The value of the keytop code sent by the real IO driver, with KeyTopOfs added. Invalid (0) in the case of TDE_KEYMETA.

code: The character code derived from the key conversion table.

Invalid (0) in the case of TDE_KEYMETA.

```
typedef struct {
        T_DEVEVT        h;      /* standard header */
        KPStat          stat;   /* PD position/button change */
} PdEvt;
```

| | | |
|---|---|---|
| h.evttyp: | TDE_PDBUT | PD button change and position change |
| | TDE_PDMOVE | PD position change |
| | TDE_PDSTATE | PD state change |

Event notification occurs when a PD button is pressed or released, or when the PD position changes.

| Button state | Position | PdEvt.h.evttyp |
|---|---|---|
| no change | no change | - |
| no change | changed | TDE_PDMOVE |
| changed | no change | TDE_PDBUT |
| changed | changed | TDE_PDBUT<br>(no notification of TDE_PDMOVE) |

TDE_PDSTATE event notification occurs when the following state changes.

| | |
|---|---|
| pdsim | PD simulation mode |

```
typedef struct {
        T_DEVEVT        h;      /* standard header */
        H               wheel;  /* wheel rotation amount       */
        H               rsv[3]; /* reserved (0)        */
} PdEvt2;
```

h.evttyp: TDE_PDEXT PD extended event

Event notification occurs when the PD wheel turns.

| | | |
|---|---|---|
| wheel: | > 0 | wheel rotation toward user |
| | < 0 | wheel rotation away from user |

- If event notification cannot be made because the message buffer is full, processing must be executed to prevent the PD button and keys from remaining on.

## 9.9 Data from Real IO Driver

The real IO driver sends any of the following messages to the input mailbox.

```
/* Sent by real IO driver */
typedef enum {
        INP_PD = 0,     /* PD data                      */
        INP_KEY = 1,    /* key data                     */
        INP_FLG = 2,    /* event flag registration      */
        INP_PD2 = 3,    /* PD data 2                    */
        SpecialReserve = -1/* Negative values reserved for special uses */
} InputCmd;

/* Device error      */
typedef enum {
        DEV_OK          = 0,    /* normal               */
        DEV_OVRRUN      = 1,    /* receive overrun      */
        DEV_FAIL        =2,     /* hardware failure     */
        DEV_SYSERR      = 3,    /* real IO driver problem */
        DEV_RESET       = 15    /* reset                */
} DevError;

/* INP_PD: send PD input */
typedef struct {
        UW              read: 1;  /* read complete flag   */
        InputCmd        cmd: 7;   /* =INP_PD              */
        UW              rsv1: 4;  /* reserved (0)         */
        DevError        err: 4;   /* device error         */

        UW              nodsp: 1; /* no pointer display   */
        UW              rsv2: 1 1; /* relative mode disabled */
        UW              onebut: 1;/* 1-button mode        */
        UW              abs: 1;   /*absolute/relative coordinates*/
        UW              norel:t: 1; /* PD timeout valid   */
        UW              tmout: 1; /* PD timeout valid     */
        UW              butrev: 1; /* button left-right reversal valid */
        UW              xyrev: 1;  /* XY coordinate inversion valid */

#if BIGENDIAN
        UW              rsv3: 3;  /* reserved (0)         */
```

```
        UW      qpress: 1; /* quick press modifier          */
        UW      inv: 1;    /* out of range (illegal coordinate)    */
        UW      vst: 1;    /* out-of-range corrected        */
        UW      sub: 1;    /* sub button state              */
        UW      main: 1;   /* main button state             */

    #else
        UW      main: 1;   /* main button state             */
        UW      sub: 1;    /* sub button state              */
        UW      vst: 1;    /* out-of-range corrected        */
        UW      inv: 1;    /* out of range (illegal coordinate) */
        UW      qpress: 1; /* quick press modifier          */
        UW      rsv3: 3;   /* reserved (0)                  */
    #endif
    } PdInStat;
```

nodsp: Pointer display is disabled by upper software, so operations are reflected directly in MetaBut. (1 is set on touch panel input.)

norel: (Valid when abs = 1. Even if the PD attribute is relative mode, operation is in absolute mode.)
Currently norel is not used. When abs = 1, operation is always in absolute mode.

tmout: When set to 1, PD timeout is valid.

butrev: When set to 1, the main and sub buttons are switched if the left-right reversal attribute is 1.

xyrev: When set to 1, x and y coordinate values are sign-inverted if the left-right reversal attribute is 1.
This is valid only when abs = 0.

qpress: When set to 1, main button press actions are treated as quick press actions.
qpress modifies the main button and is treated like a shift key. Normally this indicates the state of the second side button on a pen or the middle button on a mouse.

onebut: When set to 1, qpress operates in one-button mode.
With 1 button operation, qpress = 1 is treated as though a main button is also pressed. This is normally used for quick press operations only on the middle button of a mouse.

inv: Outside the valid area. The coordinate value is invalid.

vst: This is set once after moving from outside the valid area into the valid area.

```
typedef struct {
    T_MSG   head;
    PdInStat stat;
    H       xpos;    /* X coordinate position (relative/absolute)    */
    H       ypos;    /* Y coordinate position (relative/absolute)    */
} PdInput;
```

xpos:
ypos: In absolute coordinates mode, these values are in the fixed range: (0, 0)-(PDIN_XMAX-1, PDIN_YMAX-1). In relative coordinates mode, they indicate the amount of change (plus/minus) in the coordinate value.

```
#define PDIN_XMAX       4096
#define PDIN_YMAX       3072
```

The coordinates range is hardware-dependent.
Sent when the PD position or button state changes.
After the KBPD driver is loaded, the initial setting is read = 1.

INP_PD2: Send PD input 2 (Wheel Mouse)

```
typedef struct {
        UW              read: 1;   /* read complete flag       */
        InputCmd        cmd: 7;    /* =INP_PD2                 */
        UW              rsv1: 4;   /* reserved (0)             */
        DevError        err: 4;    /* device error             */
        UW              rsv2: 16;  /* reserved (0)             */
} PdIn2Stat;

typedef struct {
        T_MSG           head;
        PdIn2Stat       stat;
        H               wheel;     /* wheel rotation amount    */
        H               rsv;       /* reserved (0)             */
} PdInput2;
```

This command is sent when the mouse wheel is rotated.
After the KBPD driver is loaded, the initial setting is read = 1.

```
Wheel > 0       wheel rotation toward user
Wheel < 0       wheel rotation away from user
```

INP_KEY: Send key input

```
typedef struct {
        UW              read: 1;        /* read complete flag */
        InputCmd        cmd: 7;         /* =INP_KEY         */
        UW              rsv1: 4;        /* reserved (0)     */
        DevError        err: 4;         /* device error     */
        UW              rsv2: 7;        /* reserved (0)     */
        UW              tenkey: 1;      /* 1 if numeric keypad */
        UW              kbid: 7;        /* keyboard ID      */
        UW              press: 1;       /* on: 1, off: 0    */
} KeyInStat;
typedef struct {
        T_MSG           head;
        KeyInStat       stat;
```

```
        W           keytop;         /* keytop code      */
} KeyInput;
```

Sent when a key is pressed or released. After the KBPD driver is loaded, the initial setting is read = 1.

INP_FLG: Register/unregister command event flag

```
typedef struct {
        UW          read: 1;        /* read complete flag       */
        InputCmd    cmd: 7;         /* =INP_FLG                 */
        UW          rsv1: 4;        /* reserved (0)             */
        DevError    err: 4;         /* always DEV_OK            */
        UW          rsv2: 7;        /* reserved (0)             */
        UW          kb: 1;          /* 1 if kbid is valid       */
        UW          kbid: 7;        /* keyboard ID              */
        UW          reg: 1;         /* register: 1, unregister: 0 */
} FlgInStat;

typedef struct {
        T_MSG       head;
        FlgInStat   stat;
        ID          flgid;          /* event flag ID            */
} FlgInput;
```

When the real IO driver is initialized, an event flag is registered if command receipt is required.
No event flag is registered if command receipt is not required.
Up to four event flags can be registered, and any more than that will be ignored.
When the real IO driver is terminated, the event flags are unregistered.

The KBPD driver sends the required commands to all registered event flags.

After the KBPD driver is loaded, the initial setting is read = 1.

## 9.10 Real IO Driver Commands

The following procedures are used for command exchange.

At the KBPD driver:

```
/* Wait for command setting READY */
tk_wai_flg(flg_id, 0x80000000, TWF_ORW | TWF_CLR, &dmy, tmo);
/* Set command : cmd < 0x80000000 */
tk_set_flg(flg_id, cmd);
```

At the real IO driver:

```
for (;;) {
```

```
            /* Command entry READY */
            tk_set_flg(flg_id, 0x80000000);
            /* Wait for command */
            tk_wai_flg(flg_id, 0x7fffffff, TWF_ORW | TWF_CLR, &cmd, tmo);
            <command processing>
    }
```

The KBPD driver sends the following commands to all the event flags registered by the real IO driver.

PD scan rate command:

This command is sent when the PD scan rate attribute changes or when an event flag is first registered. The actual rate is dependent on the real IO driver.

```
#define ScanRateCmd(rate)           (0x01000000 | (rate))
```

rate = PD scan rate, from 0 (minimum) to 15

PD sensitivity command:

This command is sent when the PD sensitivity attribute changes or when an event flag is first registered. The actual sensitivity is dependent on the real IO driver.

```
#define SenseCmd(sense)             (0x02000000 | (sense))
    sense =     PD sensitivity, from 0 (minimum) to 15
                | PD_ABS            absolute mode
                | PD_ACMSK          acceleration mask

#define PD_ABS              0x0100
```

When absolute coordinates mode is designated, if the PD supports absolute mode, absolute coordinates data must be sent with INP_PD.
When absolute coordinates mode is not designated, if the PD supports relative coordinates mode, relative coordinates data must be sent with INP_PD.

```
#define PD_ACMSK            0x0e00
```

Setting for pointer motion acceleration (valid only in relative coordinates mode)

| | |
|---|---|
| 0 | No acceleration |
| 1 to 7 | Acceleration (small to large) |

Input mode command:

This command is sent when the input mode changes among alphanumeric upper case, alphanumeric lower case, hiragana, and katakana, or when an event flag is initially registered.
The real IO driver activates LEDs based on the input mode.

```
#define InputModeCmd(mode)          (0x03000000 | (mode))
```

mode: InputMode value
(HiraMode, AlphaMode, KataMode, CapsMode)

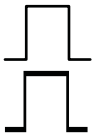
Suspend / resume:

This command causes a transition to SUSPEND state (SuspendKBPD) or return from SUSPEND state (ResumeKBPD).
When in SUSPEND state, the real IO driver need not accept any commands other than ResumeKBPD. (They may be ignored). No data is sent from the KB or PD.

```
#define SuspendKBPD     (0x10000000)
#define ResumeKBPD      (0x10000001)
```

When entering SUSPEND state, the KBPD driver must ensure all key and button states are up, not allowing them to remain pressed.

## 9.11 Valid Time,Invalid Time,and Other Detailed Specifications

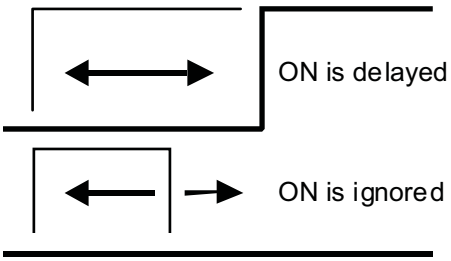


Actual key/button state

Processed key/button state

ontime:

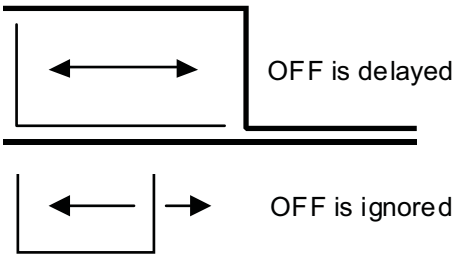
The valid time when a key or button is considered to be on.
When ON state continues for an interval of ontime or longer, the key or button is considered to be on.




offtime:

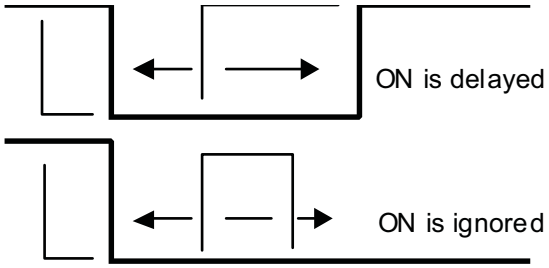
The valid time when a key or button is considered to be off.
When OFF state continues for an interval of offtime or longer, the key or button is considered to be off.

This function was not part of the conventional Enableware specification, but is useful for ignoring a momentary pen release, for example.



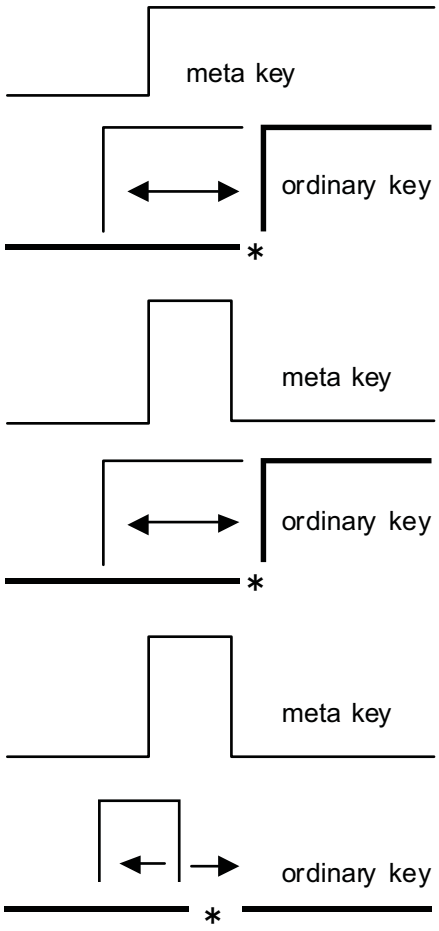

invtime:

The invalid time after going to OFF state.
During the interval invtime after going to OFF state, ON is ignored.



contime:

Permitted time for pressing concurrently with meta key. (Applies only to keys).
If a meta key and ordinary key are pressed within contime of each other, meta key latching is valid.
In each of the following cases, a meta key valid event is generated at point*.



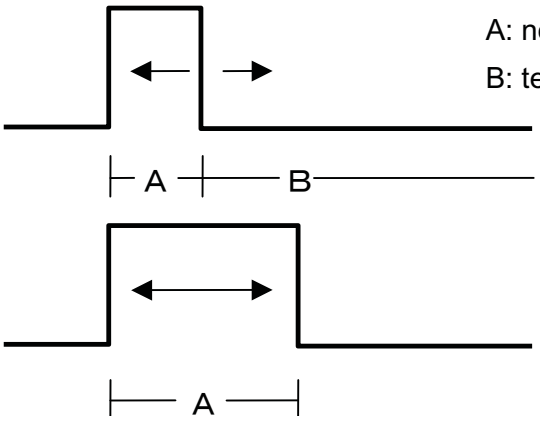

*ON/OFF events occur in succession.

timeout:

The button timeout interval. (Applies only to buttons.)
If there is no PD input during timeout after going to ON state, this is regarded as timeout, the state goes to OFF, and an OFF event is automatically generated.
Whether this function is valid or not is device dependent. It is a necessary function for a touch panel, for example, because otherwise the OFF state would not be notified.

sclktime:

The temporary shift valid interval. (Applies only to meta keys).
When a meta key is clicked for sclktime or less, a temporary shift state results.

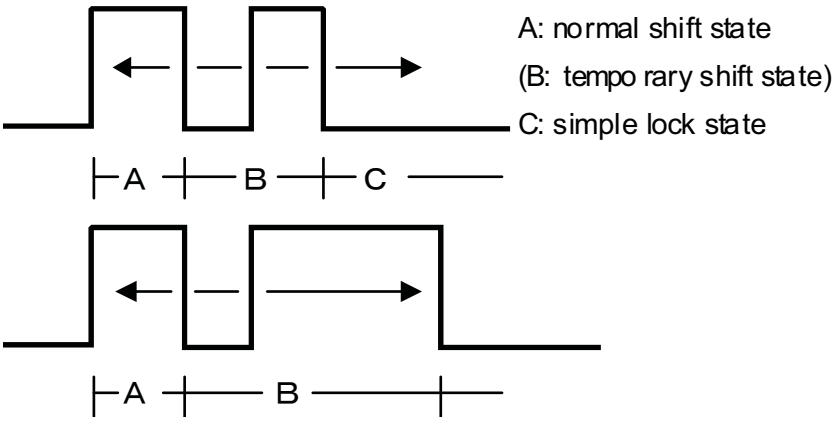



Temporary shift is released in the following cases.

When an ordinary key goes to OFF state
When the same meta key goes to OFF state
When a PD button is clicked

dclktime:

The double-click interval when simple lock is valid.
If a meta key is double-clicked within the dclktime interval, simple lock state occurs.

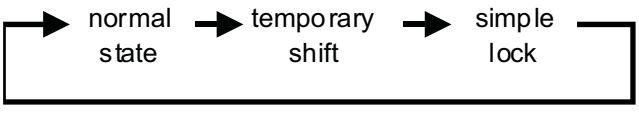



Simple lock is released in the following case.

When the same meta key goes to OFF state

tslock:

The temporary shift specification.
This is a state in which state transitions occur in the following way each time the meta key is pressed.
Ordinarily, in this state an ample value is used for sclktime and dclktime is set to 0.



## 9.12 PD Simulation

### 9.12.1 Standard PD simulation

Pressing "left shift" + "right shift" + "CC key (any of leftarrow, rightarrow, uparrow, or downarrow) toggles between standard PD simulation mode and regular mode (PD simulation off).

In standard PD simulation mode, uparrow is displayed in the pointer.

PD actions can be performed by the following key operations in standard PD simulation mode.

PgDn, sub leftarrow
: Button on
The button goes on when a key is pressed and stays on after the key is released.

PgUp, sub rightarrow
: Button off
The button goes off when a key is pressed and stays off after the key is released.

End, sub downarrow
: Same as button
The button goes on when a key is pressed and off when the key is released.

Home, sub uparrow
: Button click
Pressing a key simulates button ON - OFF - ON (click press) states. The button goes off when the key is released.

rightarrow, leftarrow, uparrow, downarrow
: PD motion

### 9.12.2 Main button PD simulation

Pressing "left shift" + "right shift" + "any of HOME, End, PgUp, PgDn/sub CC key (any of leftarrow, rightarrow, uparrow, or downarrow)" toggles between main button PD simulation mode and regular mode.

In main PD simulation mode, TRIANGLE is displayed in the pointer.

PD actions can be performed by the following key operations in main PD simulation mode.

PgDn, sub leftarrow
: Button on
The button goes on when a key is pressed and stays on after the key is released.

PgUp, sub rightarrow
: Button off
The button goes off when a key is pressed and stays off after the key is released.

End, sub downarrow
: Same as button
The button goes on when a key is pressed and off when the key is released.

Home, sub uparrow
: Button click
Pressing a key simulates button ON - OFF - ON (click press) states. The button goes off when the key is released.

The only difference from standard PD simulation is that there is no PD motion with leftarrow, rightarrow, uparrow, or downarrow.

### 9.12.3 Numeric keypad PD simulation

Pressing "left shift" + "right shift" + "any of leftarrow, rightarrow, uparrow, or downarrow in the numeric keypad" toggles between numeric keypad PD simulation mode and regular mode.

- "NumLock" can be either on or off.

In numeric keypad PD simulation mode, uparrow is displayed in the pointer.

In numeric keypad PD simulation mode:

- When "NumLock" is off, or
- When "NumLock" is on and either "left shift" or "right shift" is pressed,

PD actions can be performed by the following key operations.

PgDn in numeric keypad, sub leftarrow
: Button on
The button goes on when a key is pressed and stays on after the key is released.

PgUp in numeric keypad, sub rightarrow
: Button off
The button goes off when a key is pressed and stays off after the key is released.

End in numeric keypad, sub downarrow
: Same as button
The button goes on when a key is pressed and off when the key is released.

Home in numeric keypad, sub uparrow
: Button click
Pressing a key simulates button ON - OFF - ON (click press) state. The button goes off when the key is released.

rightarrow, leftarrow, uparrow, or downarrow in numeric keypad
: PD motion

### 9.12.4 Additional note

Switching from any of the three PD simulation modes to regular mode can be done by any of the following key operations.

- Pressing "left shift" + "right shift" + "any of rightarrow, leftarrow, uparrow, or downarrow in the numeric keypad)"

- Pressing "left shift" + "right shift" + "any of HOME, End, PgUp, PgDn/sub CC key (any of LEFT, rightarrow, uparrow, or downarrow)"
- Pressing "left shift" + "right shift" + CC key (any of LEFT, rightarrow, uparrow, or downarrow)"

In other words, switching to regular mode is possible with the key operations used to get to the current PD simulation mode. It is also possible by using either of the key operations for the other two PD simulation modes.

## 9.13 **Special** Key Codes

The following are special key codes used by the KBPD driver. These are the codes that were converted using the keycode table.

Meta keys:

```
#define KC_EIJI     0x1000    /* alphanumeric <--> Japanese                */
#define KC_CAPN     0x1001    /* hiragana <--> katakana                    */
#define KC_SHT_R    0x1002    /* right shift                               */
#define KC_SHT_L    0x1003    /* left shift                                */
#define KC_EXP      0x1004    /* extension                                 */
#define KC_CMD      0x1005    /* command                                   */
#define KC_JPN0     0x1006    /* hiragana                                  */
#define KC_JPN1     0x1007    /* katakana                                  */
#define KC_ENG0     0x1008    /* alphanumeric                              */
#define KC_ENG1     0x1009    /* alphanumeric CAPS                         */
#define KC_KBSEL    0x100a    /* kana <--> alphabet                        */
#define KC_ENGALT   0x100b    /* -> alphanumeric <--> alphanumeric CAPS    */
#define KC_JPNALT   0x100c    /* -> hiragana <--> katakana                 */

#define KC_HAN      0x1150    /* full-width <--> half-width                */
#define KC_JPN0_Z   0x1016    /* full-width & hiragana                     */
#define KC_JPN1_Z   0x1017    /* full-width & katakana                     */
#define KC_ENG0_H   0x1018    /* half-width & alphanumeric                 */
#define KC_ENG1_H   0x1019    /* half-width & alphanumeric CAPS            */
```

Keys used for PD simulation:

```
#define KC_HOME     0x1245    /* Home                                      */
#define KC_PGUP     0x1246    /* Page Up                                   */
#define KC_PGDN     0x1247    /* Page Down                                 */
#define KC_END      0x125e    /* End                                       */

#define KC_CC_U     0x0100    /* main CC key uparrow                       */
#define KC_CC_D     0x0101    /* main CC key downarrow                     */
#define KC_CC_R     0x0102    /* main CC key rightarrow                    */
#define KC_CC_L     0x0103    /* main CC key leftarrow                     */

#define KC_SC_U     0x0104    /* sub CC key uparrow                        */
#define KC_SC_D     0x0105    /* sub CC key downarrow                      */
#define KC_SC_R     0x0106    /* sub CC key rightarrow                     */
#define KC_SC_L     0x0107    /* sub CC key leftarrow                      */

#define KC_SS_U     0x0108    /* scroll key uparrow                        */
```

```
#define KC_SS_D     0x0109      /* scroll key downarrow      */
#define KC_SS_R     0x010a      /* scroll key rightarrow     */
#define KC_SS_L     0x010b      /* scroll key leftarrow      */

#define KC_PG_U     0x010c      /* page key uparrow          */
#define KC_PG_D     0x010d      /* page key downarrow        */
#define KC_PG_R     0x010e      /* page key rightarrow       */
#define KC_PG_L     0x010f      /* page key leftarrow        */
```

## 9.14 Error Codes

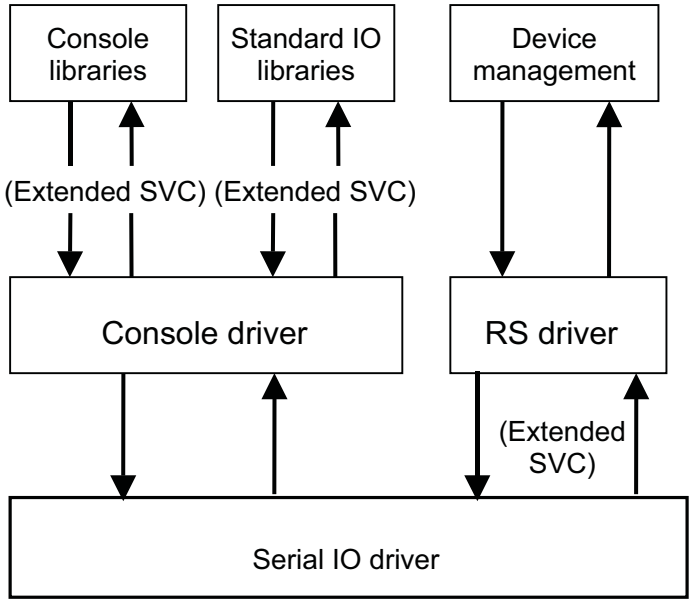
See the section on device management functions in the T-Kernel specification.
There are no special error codes specific to the KBPD driver.

# 10. Console

TEF040-S211-01.00.00/en

## 10.1 Console Overview

A console is a facility for standard character IO through serial ports and a virtual console. The overall system configuration is depicted below.




The console driver offers console functions while the serial IO driver handles the actual serial port input and output.

Applications use the console driver through standard IO libraries and console libraries. If a console is connected to a serial port, applications also use the serial IO driver.

When an application uses a serial port directly as an ordinary device, it uses the serial IO driver via device management and the RS-232C driver.

A console has a different structure than ordinary device drivers, and it employs dedicated system calls (extended SVC) for console functions.

## 10.2 Console

A system may have more than one console, creating them dynamically and using console port numbers to identify each console.

A console has the following attributes.

- Type (CONF)

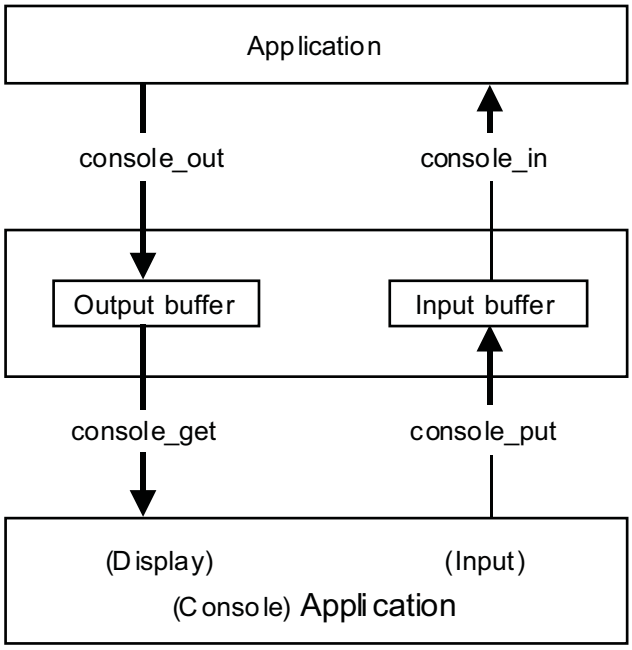
  When a console is created, one of the following attributes is designated to indicate its type.

  CONF_SERIAL: Serial port type

  The console is connected to a serial port (0 to N) through which IO is performed.

  CONF_BUFIO: Buffer IO type

  The console is not connected to any particular device. It consists only of an IO buffer. Input and output is performed when an application corresponding to a device uses this IO buffer for operations.




- Send timeout (SNDTMO)

  Indicates the timeout in milliseconds for sending (output) to the console. The default is -1 (no timeout).

- Receive timeout (RCVTMO)

  Indicates the timeout in milliseconds for receipt (input) from the console. The default is -1 (no timeout).

- Receive buffer size (RCVBUFSZ)
  Indicates the size in bytes of the console receive (input) buffer, designated when the console is created. If the console is a serial port type, the receive buffer of the serial port is used, so there is no console receive buffer.

- Send buffer size (SNDBUFSZ)
  Indicates the size in bytes of the console send (output) buffer, designated when the console is created.

- Echo (ECHO)
  Indicates whether characters received from the console (input) are automatically echoed back. The default is no echoback.

- Input mode (MINPUT)
  Indicates one of the following modes for receipt (input) from the console. The default is CANONICAL mode.

  RAW mode: Raw input 1 character at a time
  Console input is made 1 character (byte) at a time. Each input character is returned as is, without any conversion at all.

  CANONICAL mode: Line input

  Console input is made one line at a time. The end of the line is either an LF code or CR code, with CR codes being converted to LF codes when they are returned.

  EDIT mode: Editable line input

  Console input is made one line at a time with editing enabled. Input is always echoed back in this mode. The following control codes are valid.

  ESC A, ESC [ A, ^P (cursor uparrow)
  Call up previously input line (history).
  ESC B, ESC [ B, ^N (cursor downarrow)
  Call up next input line (history).
  ESC C, ESC [ C, ^F (cursor FWD)
  Move cursor right.
  ESC D, ESC [ D, ^B (cursor BWD)
  Move cursor left.
  ^H (BS), 0x7F (DEL)
  Move cursor left, deleting 1 character.
  ^X, ^U (CAN)
  Delete an entire line.
  ^K (ERASE)
  Delete to the right of the cursor position.
  ^M (CR)
  End a line. This is converted to LF code.
  ^J (LF)
  End a line.
  ^C (Ctrl-C)

Cancel input.

Other control codes

Those other than ^I (TAB) are ignored.

- Conversion of output line feed code (NEWLINE)

  Indicates whether an LF code sent to the console (output) is converted to CR code and LF code. The default is no conversion.

- Flow control (FLOWC)

  Indicates the flow control applied to console input and output. The following combinations are available. (The default is no flow control.)

| | |
|---|---|
| IXON | XON/XOFF output flow control |
| IXANY | When IXON is used, output resumes with receipt of any character. |
| IXOFF | XON/XOFF input flow control |

## 10.3 Console Port Numbers

Console ports are numbered sequentially from 1. Normally the following consoles are created when the system boots.

- Debug console (port number = 1)

```
CONF        = CONF_SERIAL (serial port #0)
SNDTMO      = -1
RCVTMO      = -1
RCVBUFSZ    = default
SNDBUFSZ    = default
ECHO        = 1
INPUT       = EDIT
NEWLINE     = 1
FLOWC  = IXON | IXOFF
```

- Standard RS port (port number = 2)

```
CONF        = CONF_SERIAL (serial port #0)
SNDTMO      = -1
RCVTMO      = -1
RCVBUFSZ    = default
SNDBUFSZ    = default
ECHO        = 0
INPUT       = CANONICAL
NEWLINE     = 0
FLOWC       = 0
```

One console is allocated to a process and is inherited by its child processes. The debug console (port number = 1) is allocated by default, but a different console can be allocated by changing the port number.

A standard IO library applies to the console allocated to the invoking process. In the case of a non-process task, it applies to port number = 1 (debug console).
Output by syslog() is also made to port number = 1 (debug console).

## 10.4 Data Definitions

```
// Port number

#define CONSOLE_PORT    1       /* debug console            */
#define RS_PORT         2       /* standard RS port         */

// cons_ioctl() command

#define GETCTL          0x100   /* get setting      */
#define ECHO            1       /* echo (0: off, 1: on)     */
#define INPUT           2       /* input mode (RAW, etc)    */
#define NEWLINE         3       /* output LF conversion (0: don't convert, 1: convert) */
#define FLOWC           4       /* flow control (0: none, IXON, etc)    */
#define SNDTMO          0x81    /* send timeout (ms), -1: none          */
#define RCVTMO          0x82    /* receive timeout (ms), -1: none       */

#define RCVBUFSZ        0x83    /* input buffer size: GET only          */
#define SNDBUFSZ        0x84    /* output buffer size: GET only         */

// Input mode

#define RAW             1       /* 1 character raw input    */
#define CANONICAL       3       /* 1 line input (CR converted to LF)    */
#define EDIT            5       /* 1 line editable input    */

// Flow control

#define IXON            0x01    /* XON/XOFF output flow control     */
#define IXANY           0x02    /* IXON output resumed on receipt of any character */
#define IXOFF           0x04    /* XON/XOFF input flow control      */

// cons_conf() commands

#define CS_CREATE       0x11    /* create console   */
#define CS_DELETE       0x12    /* delete console   */
#define CS_SETCONF      0x13    /* set console configuration            */
#define CS_GETCONF      0x14    /* get console configuration            */
#define CS_GETPORT      0x21    /* get standard console port            */
#define CS_SETPORT      0x22    /* set standard console port            */
#define CS_SRCHPORT     0x23    /* search console port      */

// Types (configuration)

#define CONF_SERIAL_0   (0)     /* serial port # 0    */
#define CONF_SERIAL(n)  (n)     /* serial port # N    */
#define CONF_BUFIO      (-2)    /* buffer IO          */
```

## 10.5 Console System Calls

The following services for working with consoles are provided as extended system calls.

### 10.5.1 console_in - Console input

[Format]

W console_in(W port, B *buf, UW len)

[Parameters]

| | |
|---|---|
| port | Console port number |
| buf | Input data buffer |
| len | Maximum input data size in bytes |

[Return Code]

| | |
|---|---|
| > 0 | Actual number of input bytes |
| = 0 | No bytes were input |
| = -1 | Input interrupted (only when input mode is EDIT mode) |

[Description]

Inputs up to len bytes of data from the console designated by port and stores the data in buf. The actual number of input bytes is indicated in the return code.

Depending on the designated console input mode, the operation differs as follows.

In RAW mode:

Returns after len bytes of data input.
Returns if no data is received within the receive timeout interval.
When echo is on, the input data is echoed back.

In CANONICAL mode, or in EDIT mode with len == 1:

Returns after len bytes of data input.
Returns when CR or LF is input. Converts CR to LF and stores in buf.
Returns when ^C is input. ^C is stored in buf.
Returns if no data is received within the receive timeout interval.
When echo is on, the input data is echoed back. LF is echoed back as CR + LF.

In EDIT mode with len > 1:

Inputs data a line at a time with editing enabled. len must be large enough to allow one-line editable input.
Returns when CR or LF is input. Converts CR to LF and stores in buf.
Returns with a return code of -1 when ^C is input. ^C is not stored in buf.
Returns if no data is received within the receive timeout interval.
Input data is echoed back. LF is echoed back as CR + LF.

[Error Code]

None

### 10.5.2 console_out - Console output

[Format]

| | |
|---|---|
| ERR | console_out(W port, B *buf, UW len) |

[Parameters]

| | |
|---|---|
| port | Console port number |
| buf | Output data buffer |
| len | Size in bytes of data to be output |

[Return Code]

| | |
|---|---|
| > 0 | Actual number of output bytes |
| = 0 | No bytes were output |

[Description]

Outputs len bytes of data from buf to the console designated by port and indicates the actual number of output bytes in the return code.

If data could not be output to the designated console within the designated output timeout limit, the system call returns at that point.

If output line feed conversion is on, LF is converted to CR + LF for output.

[Error Code]

None

### 10.5.3 console_ctl - Console control

[Format]

| | |
|---|---|
| W | console_ctl(W port, W req, W arg) |

[Parameters]

| | |
|---|---|
| port | Console port number |
| req | Command |
| arg | Command parameter |

[Return Code]

| | |
|---|---|
| any | The acquired current setting |
| = 0 | Setting successful |
| = -1 | Error |

[Description]

Performs the operation designated by req on the console designated by port as follows.

| | |
|---|---|
| ECHO \| GETCTL | Gets the current ECHO mode. (arg is not used) |
| ECHO | The ECHO mode is set in arg. |
| INPUT \| GETCTL | Gets the current INPUT mode. (arg is not used) |
| INPUT | The INPUT mode is set in arg. |
| NEWLINE \| GETCTL | Gets the current NEWLINE mode. (arg is not used) |

| | |
|---|---|
| NEWLINE | The NEWLINE mode is set in arg. |
| FLOWC \| GETCTL | Gets the current FLOWC mode. (arg is not used) |
| FLOWC | The FLOWC mode is set in arg. |
| SNDTMO \| GETCTL | Gets the current SNDTMO value. (arg is not used) |
| SNDTMO | The SNDTMO value is set in arg. (arg < 0 is regarded as -1) |
| RCVTMO \| GETCTL | Gets the current RCVTMO value. (arg is not used) |
| RCVTMO | The RCVTMO value is set in arg. (arg < 0 is regarded as -1) |
| RCVBUFSZ \| GETCTL | Gets the current RCVBUFSZ value. (arg is not used) |
| SNDBUFSZ \| GETCTL | Gets the current SNDBUFSZ value. (arg is not used) |

[Error Code]
None

### 10.5.4 console_get - Read console output data

[Format]

W console_get(W port, B *buf, UW len, W tmout)

[Parameters]

| | |
|---|---|
| port | Console port number |
| buf | Read data buffer |
| len | Maximum read data size in bytes |
| tmout | Timeout (ms) |

[Return Code]

| | |
|---|---|
| > 0 | Actual number of read bytes |
| = 0 | No bytes were read |

[Description]

Reads up to len bytes of data from the buffer IO console designated by port and stores the data in buf.
The actual number of read bytes is indicated in the return code.

The data read by this system call is data output by console_out().

If the console output buffer is empty, the behavior is as follows.

| | |
|---|---|
| tmout = 0: | Returns without waiting. |
| tmout = -1: | Waits indefinitely until data arrives in the output buffer. |
| tmout > 0: | Waits up to tmout ms for data to arrive in the output buffer. |

If the console type is not buffer IO, no operation occurs and 0 is returned.

[Error Code]
None

### 10.5.5 console_put - Write console input data

[Format]

ERR console_put(W port, B *buf, UW len, W tmout)

[Parameters]

| | |
|---|---|
| port | Console port number |
| buf | Write data buffer |
| len | Size in bytes of data to be written |
| tmout | Timeout (ms) |

[Return Code]

| | |
|---|---|
| > 0 | Actual number of bytes written |
| = 0 | No bytes were written |

[Description]

Writes len bytes of data from buf to the buffer IO console designated by port and indicates the actual number of written bytes in the return code.

The data written is data input by console_in().

If the console input buffer is full, the behavior is as follows.

| | |
|---|---|
| tmout = 0: | Returns without waiting. |
| tmout = -1: | Waits indefinitely until the input buffer has free space. |
| tmout > 0: | Waits up to tmout ms for free input buffer space. |

If the console type is not buffer IO, no operation occurs and 0 is returned.

[Error Code]

None

### 10.5.6 console_conf - Console configuration

[Format]

ERR console_conf(W req, UW *arg)

[Parameters]

| | |
|---|---|
| req | Command |
| arg | Command parameter |

[Return Code]

| | |
|---|---|
| = 0 | Normal completion |
| = -1 | Error |

[Description]

Console operations such as creation and configuration change are designated in req as follows.

CS_CREATE Create a console
arg[0] = port number OUT
arg[1] = console type IN
arg[2] = input buffer size IN
arg[3] = output buffer size IN
Creates a new console as designated in arg[1 to 3]. The created port number is returned in arg[0]. The other console attributes are the defaults.

CS_DELETE Delete a console
arg[0] = port number IN
Deletes the console designated in arg[0].

CS_SETCONF Set console configuration (recreate)
arg[0] = port number IN
arg[1] = console type IN
arg[2] = input buffer size IN
arg[3] = output buffer size IN
Changes the configuration of the console designated by arg[0] to the attribute settings made in arg[1 to 3]. The other console attributes are the defaults.

CS_GETCONF Get console configuration
arg[0] = port number IN
arg[1] = console type OUT
arg[2] = input buffer size OUT
arg[3] = output buffer size OUT
Returns in arg[1 to 3] the current configuration of the console designated by arg[0].

CS_GETPORT Get standard console
arg[0] = port number OUT
Returns in arg[0] the port number of the console currently set for the invoking process.

CS_SETPORT Set standard console
arg[0] = port number IN
Changes the console of the invoking process to the console having the port number designated in arg[0]. The changed console is inherited by the child processes.

CS_SRCHPORT Search console port
arg[0] = port number IN/OUT
arg[1] = configuration IN
Searches for a console port having a larger port number than that designated in arg[0] and matching the configuration designated in arg[1]. If one is found, the port number (> 0) is returned in the function value and arg[0]. If none is found, 0 is returned in the function value.

[Error Code]
None

## 10.6 Console Library

Console operations and low-level serial IO normally take place through the following library functions instead of direct use of system calls.

### 10.6.1 _PutString - Output string to console

[Format]

int _PutString(char *buf)

[Parameters]

buf Character (byte) string for output

[Return Code]

> 0 Number of characters (bytes) actually output

= -1 No characters (bytes) were output

[Description]

Outputs the character string in buf to the console currently allocated to the invoking process.

The character string must be null (0) terminated.

This library function uses the system call console_out().

Output by standard IO library functions such as printf() is performed using this library function.

### 10.6.2 _PutChar - Output 1 character to console

[Format]

int _PutChar(int c)

[Parameters]

c Character (byte) for output

[Return Code]

= 1 Character output succeeded

= -1 Output failed

[Description]

Outputs the character designated by c to the console currently allocated to the invoking process.

Only the low byte of c is valid.

This library function uses the system call console_out().

Output by standard IO library functions such as putchar() is performed using this library function.

### 10.6.3 _GetString - Input 1 line from the console

[Format]

int _GetString(char *buf)

[Parameters]

buf Memory area for storing the input character (byte) string

[Return Code]

> 0 Number of bytes actually input
= 0 No bytes were input
= -1 Input interrupted (only when input mode is EDIT mode)

[Description]

Inputs 1 line from the console currently allocated to the invoking process and stores it in buf.
The memory area designated by buf must be sufficiently large to accept the input.

The line stored in buf is terminated by 0, but a final LF code is not stored in buf. Input conforms to the input mode set for the console so if the mode is RAW, input is made 1 character at a time instead of as a line.

This library function uses the system call console_in().

Input by standard IO library functions such as gets() is performed using this library function.

### 10.6.4 _GetChar - Input 1 character from the console

[Format]

int _GetChar()

[Parameters]

None

[Return Code]

> 0 The input character (byte)
= -1 Input failed

[Description]

Inputs 1 character from the console currently allocated to the invoking process and indicates it in the return code.

Actual input conforms to the input mode set for the console.

This library function uses the system call console_in().

Input by standard IO library functions such as getchar() is performed using this library function.

### 10.6.5 cons_ioctl - Console control

[Format]

int cons_ioctl(int req, int arg)

[Parameters]

req Command
arg Command parameter

[Return Code]

any The acquired current setting
= 0 Setting succeeded

[Description]

Performs the control operation designed by req and arg for the console currently allocated to the invoking process.

This library function uses the system call console_ctl().

### 10.6.6 RS_putchar - Output 1 character to standard RS port

[Format]

int RS_putchar(int c)

[Parameters]

int Character (byte) for output

[Return Code]

= 1 Output succeeded
= -1 Output failed

[Description]

Outputs the character designated by c to the standard RS port. Only the low byte of c is valid.

This library function uses the system call console_out().

### 10.6.7 RS_getchar - Input 1 character from standard RS port

[Format]

int RS_getchar()

[Parameters]

None

[Return Code]

> 0 The input character (byte)
= -1 Input failed

[Description]

Inputs 1 character from the standard RS port and indicates it in the return code.
This library function uses the system call console_in().

### 10.6.8 RS_ioctl - Standard RS port control

[Format]

```
int    RS_ioctl(int req, int arg)
```

[Parameters]

req  Command
arg  Command parameter

[Return Code]

any  The acquired current setting retrieved
= 0  Setting succeeded

[Description]

Performs the control operation designated by req and arg for the standard RS port.

This library function uses the system call console_ctl().

### 10.6.9 cons_put - Write to console input buffer

[Format]

```
W    cons_put(W port, B *buf, UW len, W tmout)
```

[Description]

Executes console_put(port, buf, len, tmout).

### 10.6.10 cons_get - Read from console output buffer

[Format]

```
W    cons_get(W port, B *buf, UW len, W tmout)
```

[Description]

Executes console_get(port, buf, len, tmout).

### 10.6.11 cons_conf - Configure console

[Format]

```
W    cons_conf(W req, UW *arg)
```

[Description]

Executes console_conf(req, arg).

## 10.7 Console Application Processing

An application offering a virtual console on the screen generally performs the processing

outlined below.

1.Creates a buffer IO console.
arg[1] = CONF_BUFIO
cons_conf(CS_CREATE, arg)

2. Switches the invoking process console to the created buffer IO console.
cons_conf(CS_SETPORT, arg)
Thereafter this buffer IO console is allocated to created child processes.

3. Periodically performs buffer IO console data processing.
Output processing from a child process:
Displays the data acquired by cons_get(arg[0],..) on the screen.
Input processing to a child process:
Uses cons_put(arg[0],..) to set key input or other input data and inputs it to the child process.

4. On termination, deletes the created buffer IO console.
cons_conf(CS_DELETE, arg)
Because the console of a created child process remains, he child processes must usually also be terminated at this time.

# 11. Screen (display) Driver

TEF040-S214-01.00.00/en

## 11.1 Applicable Devices

- System display device

## 11.2 Device Name

- The device name is "SCREEN".

## 11.3 Device-specific Functions

- Display format information acquisition

  Getting device specifications, color map, bitmap position, and so on

- Display control

  Controller initialization, changing the color map, and so on

- Timing control

  Setting monitor frequency and timing and related tasks

- Display information acquisition

  Getting hardware-related information

## 11.4 Attribute Data

The following attribute data is supported.

R Read-only
W Write-only
RW Read/write enabled

```
/* SCREEN data numbers */
typedef enum {
    /* Common attributes */
    DN_SCRSPEC = TDN_DISPSPEC, /* DEV_SPEC        (R) */

    /* Device-specific attributes: -100 to -199 are generic */
        DN_SCRLIST          = -100,    /* TC[]          (R)   */
        DN_SCRNO            = -101,    /* W             (RW) */
        DN_SCRCOLOR         = -102,    /* COLOR[]       (RW) */
        DN_SCRBMP           = -103,    /* BMP           (R)   */

        DN_SCRBRIGHT        = -200,    /* W             (RW) */
```

```
        DN_SCRUPDFN     = -300,   /* FP          (R)  */
        DN_SCRVFREQ     = -301,   /* W           (RW) */
        DN_SCRADJUST    = -302,   /* ScrAdjust   (RW) */
        DN_SCRDEVINFO   = -303,   /* ScrDevInfo  (R)*/
        DN_SCRMEMCLK    = -304,   /* W           (RW) */
} ScrDataNo;
```

DN_SCRSPEC: Get device specifications (R)

```
data:   DEV_SPEC    devspec;

typedef struct {
        H   attr;       /* device attributes                    */
        H   planes;     /* number of planes                     */
        H   pixbits;    /* number of pixel bits (boundary/valid)        */
        H   hpixels;    /* horizontal pixels                    */
        H   vpixels;    /* vertical pixels      */
        H   hres;       /* horizontal resolution                */
        H   vres;       /* vertical resolution                  */
        H   color[4];   /* color information                    */
        H   resv[6];
}   DEV_SPEC;
```

Gets the device specifications for the current display mode. (See the Device Primitive specifications for details of DEV_SPEC.)

DN_SCRLIST: Get supported display modes (R)

```
data: TC    list[];
```

Gets a list of supported display modes in the following format.

<demarcator><display mode><demarcator><display mode>.....<0>

<demarcator> separates the display modes by putting all 0s in the high byte of the display numbers (1 to N < 256).

<display mode> is a character string indicating the resolution, color depth, and so on in a simple statement such as "1024*768 256C".

The display mode groups information such as resolution and color depth in an orderly sequence. In general, it is displayed as is.
As supported display modes are added, the order may change, but the display mode numbers stay the same.

DN_SCRNO: Set/get current display mode (RW)

```
data:       W       scrno;
```

Sets or gets the current display mode number.
The display mode number is the number assigned to display modes obtained by

DN_SCRLIST.
* On some hardware, the display mode can be obtained but not set by this function.

DN_SCRCOLOR: Set/get color map (RW)

data: COLOR map[*]

Sets or gets the color map used in the current display mode.
When DEV_SPEC.attr.P = 0, no color map is applied.
The color map is an array of absolute RGB color values indexed by pixel values.(For details on the COLOR specification, refer to the BTRON3 Specification Part 2, OS Specifications 2.2.3: "Color Representation.")
The maximum number of entries is determined by the number of planes * pixel bits in DEV_SPEC, but the actual number may be less.

DN_SCRBMP: Get device-specific image area (R)

data: BMP devbmp;

Gets information about the device-specific image area (bitmap) in the current display mode.
devbmp.baseaddr[*] indicates the memory location of the image area, which can be accessed directly by device primitives. (It must not be accessed directly by general applications, however.)
A device-specific image area exists only when DEV_SPEC.attr.M = 1.

DN_SCRBRIGHT: Set/get screen brightness (RW)

data: W brightness;

Sets or gets the screen brightness in the current display mode.
Screen brightness values are set in the range from 0 (dark) to 31 (bright).
* Some hardware may not support this attribute data.

DN_SCRUPDFN:Get screen update function (R)

data FP updfn(W x, W y, W dx, W dy)

x: X coordinate, y: Y coordinate, dx: X width, dy: Y width

Gets a function pointer for notification of which area was updated when the device-specific image area content is updated.
Device primitives get this function pointer, and if it is not NULL, they call this function directly when the device-specific image area is updated. It is therefore necessary to enable direct calling of this function by device primitives.
If the set area exceeds devbmp.bounds, the excess portion is ignored.
This function is used when special processing dependent on the display hardware and display mode is required for screen updates.

DN_SCRVFREQ:Set/get monitor vertical frequency (RW)

data: W vfreq;

Sets or gets the monitor vertical frequency (refresh rate) in the current display mode.

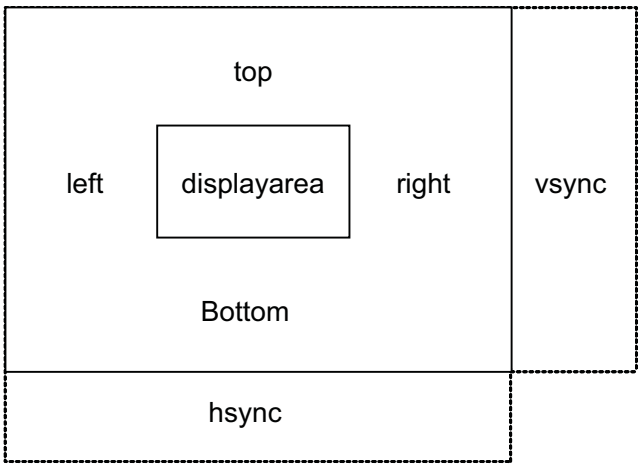
Get: vfreq < = 0 means unknown.
This function does not guarantee that the acquired value is currently applied accurately.
Set: vfreq < = 0 is ignored.
Settings must be made carefully because proper screen display is not guaranteed for all settings. There is also no guarantee that the setting will be applied accurately.
Values are normally set in the range from around 60 (Hz) to 90 (Hz).

* This function may not be supported by all display hardware or in all display modes.

DN_SCRADJUST: Set/get monitor timing adjust parameters (RW)

```
data:   ScrAdjust       adj;
typedef struct {
        UH      left;   /* left blank pixels (multiple of 8)          */
        UH      hsync;  /* horizontal blank pixels (multiple of 8)    */
        UH      right;  /* right blank pixels (multiple of 8)         */
        UH      top;    /* top blank pixels                           */
        UH      vsync;  /* vertical blank pixels                      */
        UH      bottom; /* bottom blank pixels                        */
} ScrAdjust;
```

Sets or gets the monitor timing adjustment parameters in the current display mode.




Increasing left + right + hsync narrows the display area width.
Decreasing left + right + hsync expands the display area width.
Increasing top + bottom + vsync narrows the display area height.
Decreasing top + bottom + vsync expands the display area height.
Adjusting left and right values moves the display area right and left.

Adjusting top and bottom values moves the display area up and down.

* left, hsync, and right are expressed in 8-dot units.

Settings must be made carefully because proper screen display is not guaranteed for all settings.
* This function may not be supported by all display hardware or in all display modes.

DN_SCRDEVINFO: Get device information (R)

```
data:   ScrDevInfo      info;

typedef struct {
        UB      name1[32];      /* name-1 (ASCII)                   */
        UB      name2[32];      /* name-2 (ASCII)                   */
        UB      name3[32];      /* name-3 (ASCII)                   */
        VP      framebuf_addr;  /* frame buffer physical address    */
        W       framebuf_size;  /* frame buffer size                */
        W       mainmem_size;   /* main memory size                 */
        UB      reserved[24];   /* reserved                         */
} ScrDevInfo;
```

Gets information about the display hardware.

name1, name2, and name3 indicate hardware-related information in ASCII code. To make the value a full 32 characters, the value is 0-padded.

framebuf_addr indicates the physical address of the (linear) frame buffer; it is set to NULL if a (linear) frame buffer is not used. The address may differ from one display mode to another.

framebuf_size indicates the size in bytes of the hardware frame buffer. This is not the size of the frame buffer actually used, but the total size available.
mainmem_size indicates the size of main memory used as a frame buffer.

DN_SCRMEMCLK: Set/get Video-RAM clock (RW)

```
data:   W       mclk;
```

Sets or gets the Video-RAM clock used by the graphics accelerator.

Get: mclk = 0 means unknown.
Otherwise the current Video-RAM clock (kHz) is stored in mclk.

Set: Sets the Video-RAM clock (kHz) in mclk.
For example, the Video-RAM clock is set to 133 MHz by designating mclk = 133000.
When mclk < = 0, the default for the screen driver (graphics accelerator) is used.
When mclk > 0, the largest valid setting no larger than the designated mclk

value is used. If the designated mclk value is less than the minimum valid setting, the minimum valid setting is used.
There is no guarantee that the exact value set in mclk will be used. The mclk value may also prevent correct screen display or may even cause the graphics accelerator to crash or become damaged from overheating.
The value and range that should apply to the mclk setting are dependent on the screen driver (graphics accelerator).

* This function may not be supported by all display hardware or in all display modes.

## 11.5 Device-specific Data

None

## 11.6 Basic Operations

| | |
|---|---|
| OPEN | No operation. |
| | Hardware initialization takes place when the driver starts up. |
| CLOSE, CLOSEALL | No operation. |
| ABORT | No operation (because there is no WAIT state). |
| READ, WRITE | (See above.) |
| SUSPEND, RESUME | No operation. |
| Alternatively, hardware- | dependent processing. |

## 11.7 Event Notification

None

## 11.8 Error Codes

See the section on device management functions in the T-Kernel specification.

The error code E_NOSPT is returned when setting or requesting attribute data are not supported by the hardware or in the current display mode.

## 11.9 T-Engine/SH7727 Related Information (Reference)

### 11.9.1 Unsupported functions

The following functions are not supported.

DN _SCRNO (display mode) setting
DN_SCRBRIGHT (screen brightness)
DN_SCRADJUST (monitor timing adjustment)
DN_SCRVFREQ (monitor vertical synchronization frequency)
DN_SCEMEMCLK (Video-RAM clock setting)

### 11.9.2 Supported display modes

Hsize 240
Vsize 320

| | |
|---|---|
| 256 | - |
| 65536[5-6-5] | 2 |
| 1677k[8-8-8] | - |

### 11.9.3 Display mode setting

Setting of display mode takes place only at system startup, thus no function is supported for setting this dynamically as attribute data.

The display mode is set in the DEVCONF file. When the screen driver starts up, it sets the display mode based on the settings in this file.

### 11.9.4 DEVCONF file

The following settings are made in the DEVCONF file. They take effect when the system is booted.

Display mode

VIDEOMODE mode [pmode] [w] [h] [pw] [ph]

The mode parameter sets the display mode number to be used.
The screen size is set in w (effective width) and h (effective height).
pmode, pw, and ph indicate the previous settings before the last change; these are not used by the screen driver.

CRT monitor vertical synchronization frequency (refresh rate)

VIDEOVFREQ vfreq [p_vfreq]

The vfreq parameter sets the monitor vertical synchronization frequency (refresh rate) to be used.
p_vfreq indicates the previous setting before the last change; it is not used by the screen driver.

This setting is valid only if DN_SCRVFREQ (monitor vertical synchronization frequency) is supported.

Video attributes

VIDEOATTR attr
Not used with T-Engine/SH7727. The behavior if this is set is not guaranteed.