T-Engine Forum Specification

TEF040-S212-01.00.00/en
January 14, 2004

Standard MIDI Device Driver Specification

Number: TEF040-S212-01.00.00/en
Title: Standard MIDI Device Driver Specification
Status: [ ] Working Draft, [ ] Final Draft for Voting, [*] Standard
Date: September 29, 2003 First Edited (Katsumi Ishikawa, Yamaha Corporation)
December 25, 2003 v0.92 Added setting and getting timeout in sending and receivingattribute data.
January 14, 2004 Voted.

Contents

1 Introduction

This document prescribes a device driver interface enabling the T-Engine platform to handle MIDI streams.

2 Driver Overview

The main T-Engine unit lacks a MIDI connector (DIN 5-pin), so some kind of protocol converter is required to connect MIDI devices. The MIDI device driver is for controlling these converters and sending or receiving logical MIDI streams. Individual MIDI devices are controlled by the MIDI messages sent and received by the application, and the MIDI device driver is not involved in these operations.

As for the MIDI interface device, the physical layer typically comprises devices using USB or serial connections (such as RS-232C or RS-422). In this document, there are no particular limitations for the physical transmission line. In addition, there are some composite devices with combined functions such as sound source modules with internal functions for MIDI interface, but this document focuses only on input and output of the MIDI stream and does not cover the internal module configuration inside devices.

The MIDI signal has a bit rate of 31.25 kbps and is sent or received via a serial transmission line. To maintain compatibility and ensure the number of channels*, expanding the bandwidth of the MIDI transmission capacity is done by multiplexing multiple streams instead of directly raising the bit rate of each stream. In multiplexing, the transmission line (or connection point) corresponding to one MIDI cable is called a "port" and the MIDI interface that supports multiplexing has been referred to as a multiport (MIDI) interface. When there is a need to distinguish them explicitly, the former interface is called a single port (MIDI) interface.

> * The MIDI protocol enables 16 logical channels to be used for each stream. Here, "channel" is the information included in the message packet to designate a part of the performance, different from "channel" as used in general multiplexing protocol.

As defined for the sub-class of the USB audio class in the USB-MIDI standard specification, a MIDI stream can be handled corresponding to up to 16 input and output ports. For serial MIDI as well, there is a de-facto industry standard multiplexing protocol.

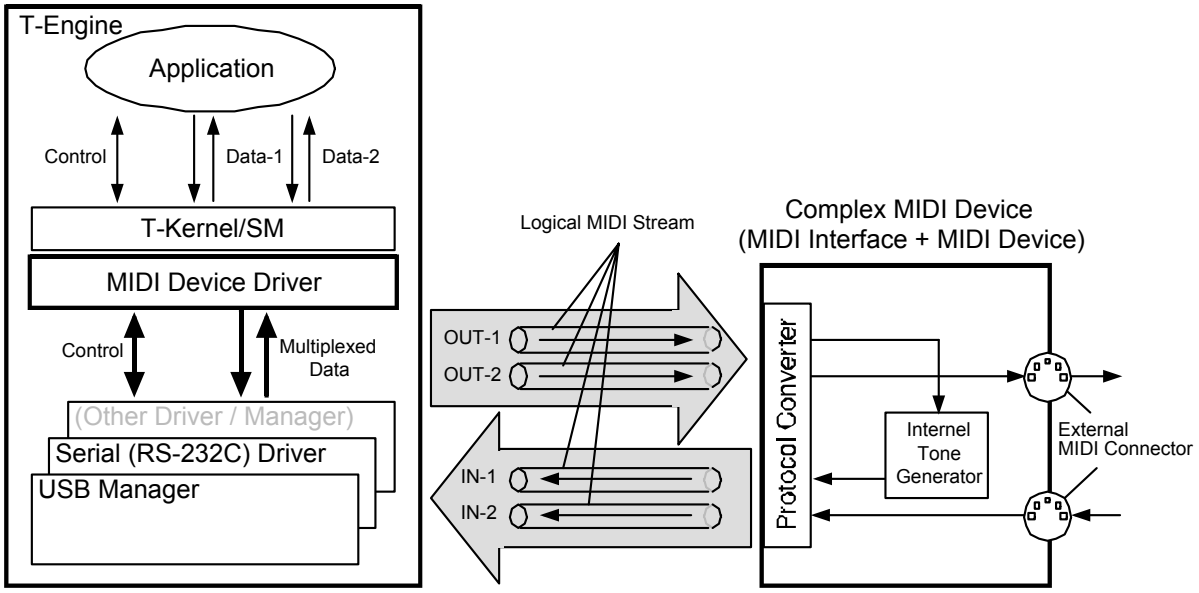

Figure 1. Functional block diagram of a sample configuration (a two-port composite device)

Figure 1 shows a functional block diagram of a sample configuration for handling MIDI streams on the T-Engine platform. In this example, the device connected to T-Engine is assumed to function as a multiport interface with two ports each for MIDI IN/OUT. Port 1 is the internal audio source, port 2 is connected to the external MIDI connector. (The MIDI is referred to as 1-Origin, excluding some exceptions.) This device can be regarded as a composite device combining MIDI interface functions and sound source functions. The application uses T-Kernel/SM device management functions to send and receive MIDI streams and control MIDI devices.

If the connected MIDI interface has multiport functions, the application must be aware of the presence of ports and specify the ports for MIDI device driver IO. Because each connector of the MIDI IO is physically independent, it is critical that they must both be able to handle the IO streams independently. For example, the performance information of a MIDI keyboard device connected to the MIDI IN connector can be sent from the MIDI OUT connector to produce a sound from a separate device with sound source functions. In principle, there is no association between the IN and OUT ports, and communication is performed via open control protocol. Although there are cases of handshake requests via bi-directional IN/OUT communication for a single device, these are not prescribed by the MIDI specification and they are established independently as high-level protocol.

The MIDI device driver performs protocol conversion with the MIDI interface device subject to control and provides MIDI stream IO functions and utility functions to the application. For multiport devices, it establishes logical ports corresponding to the number of ports that can be controlled and performs multiplexing. It must have device management functions consistent with the characteristics of the physical layer to be used.

3 Implementation

This section describes a sample module configuration of the MIDI device driver and precautions during implementation.

3.1 Sample module configuration

Figure 2 shows a sample module configuration of the device driver. Here it is assumed that the MIDI interface device is a multiport type with output n port and input m port. When driving a single-port interface, n=1 and m=1, and the multiplexing module (Multiplexer) depicted performs one-to-one protocol conversion.

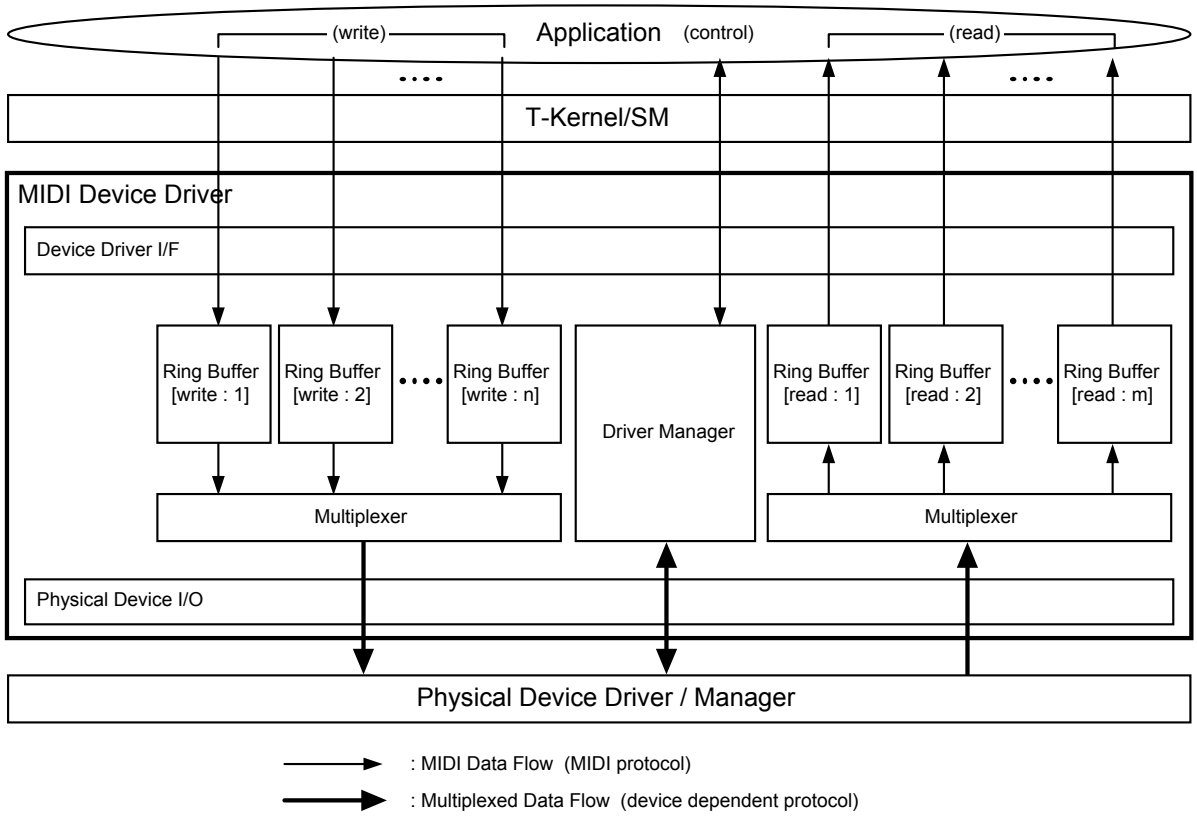

Figure 2. Device driver module configuration

Data stream between the application and device driver conforms to the MIDI protocol. (For easier implementation, some portion of specification are limited.) Data stream between the device driver and physical driver or manager uses multiplexing protocol in consistent with the device. For the multiplexing protocol, besides the USB-MIDI standard class specification, there are vendor- and device-specific methods, and this specification does not limit their implementation methods.

The device driver mainly consists of the following modules: (1) driver interface, (2) driver management module, (3) memory management module and data buffer, (4) protocol conversion module that also handles multiplexing, and (5) physical driver interface. To make it easier to isolate the IO timing adjustment and processing context, the typical method of implementation

provides a ring buffer (on the level of 256-512 bytes) for each port. The driver management module, besides performing handling related to the utility API established in this specification, provides control corresponding to the characteristics of the physical layer the driver will use. In addition, must be supported for the device management functions required by T-Kernel/SM.

The MIDI device driver provides the application with IO functions on the basis of each MIDI stream. The MIDI protocol, from the conception of channels, supports transmission and reception of 16 parts of musical performance information per stream, but the application is responsible for handling channel management and merging of information between performance parts.

3.2 Implementation precautions

3.2.1 Assignment of device name and subunit

The device name comprises "midi" to distinguish the type, followed by the unit, represented by an alphabet, and the subunit, represented by a number (from 0). If multiple drivers are loaded simultaneously, they are distinguished by the unit name. The MIDI device driver requires a subunit, and when device registration, 1 or more is designated for the attribute of the number of subunits. When the driver is opened, the physical device name ("midi" + "a"-) cannot be designated.

The MIDI device driver associates the MIDI interface device port numbers with the subunits. Caution is advised because although the subunit is 0-Origin, the port number is 1-Origin. The MIDI device driver distinguishes logical devices that are read-only from those that are write-only. Subunit 0-15 is for a read-only device, and 16-31 for a write-only device. Multiple opening of a subunit is not allowed.

Following these rules, an example of a MIDI interface device (1 IN, 4-OUT) assigned to the subunit is shown in Figure 3.

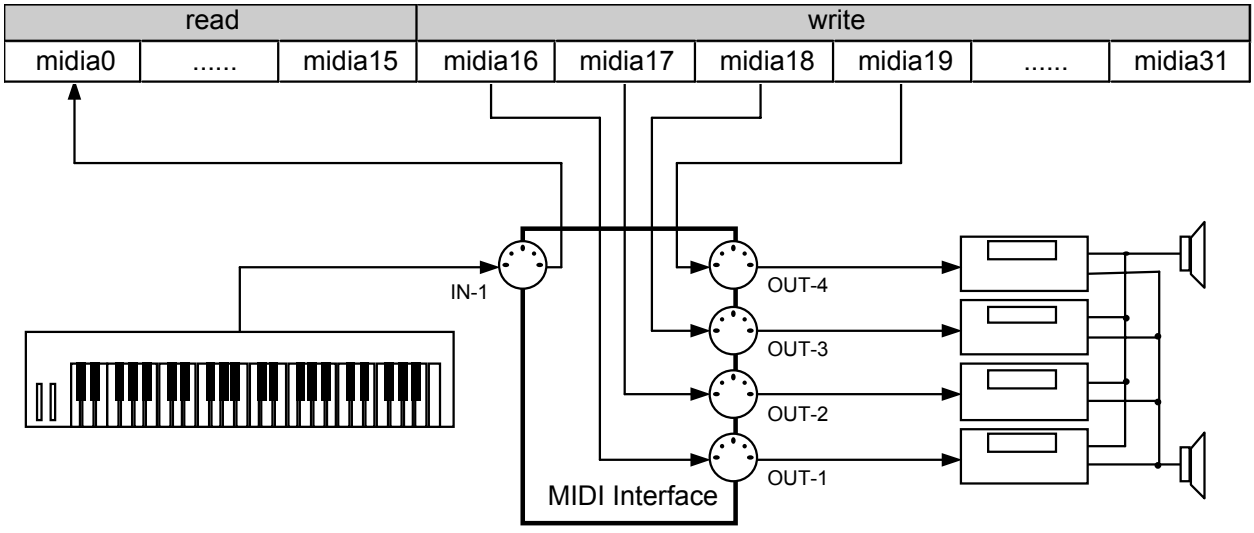

Figure 3. Relationship of subunit and MIDI ports (for a device with 1 IN and 4 OUT)

A device driver manages and controls only one device. To support connections of multiple devices, the unit name must be updated and as many device drivers as required devices must be registered. If each device uses the same control protocol, the code for driver processing can be shared.

The number of logical devices (MIDI port) that can be used with the respective physical devices can be referred with a read operation of attribute data. Referring specific information such as the device name and port name is also supported. For a transmission line supporting hot-plug, representative of USB technology, there may be a dynamic relationship between devices and ports, so the application must use port-specific information and specify a logical device.

3.2.2 Real-time of MIDI transmission and asynchronous transmission

The MIDI protocol works on a principle of instantaneous issuing and processing. Messages are not time-stamped, and the timing of their issuance itself is the expected trigger time for the event. With main messages expressed in three bytes or less, this is a protocol designed with real-time applications in mind. MIDI transmission handling is typically provided by a completion-type API assuming synchronous transmission. The API user continues the handling, viewing transmission handling as complete the moment of return from the function. In reception handling as well, synchronous transmission - that is, waiting in the driver until data of the requested size is received - is often the form of implementation. In addition, the MIDI protocol includes many messages that change behavior from past records, so in one stream handling system, the event handling sequence must be guaranteed by both the sender and receiver.

T-Kernel/SM device management functions provide extended service calls for synchronous and asynchronous transmission. For the MIDI device driver, however, these must be implemented in a way that takes into account MIDI protocol characteristics. Thus, handling requests, whether synchronous and asynchronous, must be received as promptly as possible. To avoid overhead from small-size, high-frequency access requests, if the driver's internal ring buffer is ready, that is, capable of reading or writing a portion equal to the requested size, it is advisable that handling be completed just with the call context. If a wait occurs, the procedure for asynchronous access is followed in compliance with the requested T-Kernel/SM specification.

For write requests, after data is written to the ring buffer, transfer is conducted via multiplexing and protocol conversion. As long as handling is normal for the latter stage, the transmission can be viewed as complete after writing to the buffer. Because the device driver does not append a timestamp during MIDI reception, the application must take care of timing for data retrieved from read requests.

3.2.3 Timeout of handling requests

The MIDI device driver deals with handling request timeout by dividing it into two stages.

One of these is prescribed by T-Kernel/SM device management functions: timeout occurring from delay in reception of the request itself. For details, see the T-Kernel specification.
The other is performed by MIDI device driver autonomously: canceling a request if the IO

handling for a received request is not complete by the designated time. The occurrence of timeout can be detected in the details of ioer in tk_wai_dev and in the return code for tk_srea_dev and tk_swri_dev. Even if handling timeouts occur, the data sent and received up to that moment is valid. The timeout interval can be set in the attribute data DN_MIDI_SETTMO. By default, it is 0 (no timeout).

The MIDI protocol is based on the principle of open-loop control, so timeout is not prescribed. If high-level applications require handshake with specific MIDI devices, however, timeout of the devices can be used.

3.2.4 Independence of processing among ports

With MIDI, transmission and reception must be treated independently for each port. Implementation must support independent read and write operations among different ports of a multiport device, and must also support read and write operations separately for also a single-port device.

To make a MIDI interface port support a T-Kernel/SM subunit, the device driver autonomously accepts access requests to a different subunit. Even if handling goes into a wait state for a particular subunit, this does not block requests to other subunits. Queuing of asynchronous access requests must be counted by each subunit autonomously.

3.2.5 Handling of attribute data

Requests for access to attribute data must be executed regardless of queuing of access requests for device-specific data. If asynchronous access requests to specific data of a particular port are queued, read or write requests for attribute data must be instantaneously accepted.

3.2.6 Other considerations

The MIDI device driver shall conform to the T-Kernel standard specification for matters not specifically prescribed in this specification.

4 API specification

4.1 ID tk_opn_dev(UB *devnm, UINT omode)

Reserves the MIDI port and establishes standby. Multiple opening of the same subunit is not allowed.

Argument

devnm

Pointer to the string of the logical device name. The physical device name cannot be designated.

Ex.) MIDI IN: "midi" + "a"-"z" + subunit (port number -1)
MIDI OUT: "midi" + "a"-"z" + subunit (port number +15)

omode

Designates options.

TD_NOLOCK	No locking of the device-specific data send and receive buffer by the device driver

4.2 ER tk_cls_dev(ID dd, UINT option)

Stops transmission and reception processing and releases the MIDI port. Queued asynchronous access requests are canceled.

4.3 ID tk_wai_dev(ID dd, ID reqid, INT *asize, ER *ioer, TMO tmout)

Waits completion of asynchronous access requests designated by reqid.

Argument

ioer

Returns an IO error.

(E_IO | E_MIDI_TMO)	Transmission and reception timed out from the settings in DN_MIDI_SETTMO

4.4 ID tk_wri_dev(ID dd, INT start, VP buf, INT size, TMO tmout) ER tk_swri_dev(ID dd, INT start, VP buf, INT size, INT *asize)

4.4.1 MIDI data transmission

Sends the designated size of MIDI data stored in the buffer. The buffer must not be released until the request is complete or canceled.

MIDI running status must not be used in the data to be sent. The status byte must be completed in the upper layer software. When requests are received, the device driver does not check the appropriateness of the MIDI message of the data in the buffer. (No error is returned.) At the actual time of sending (during protocol conversion), if an invalid message is detected, the buffer is read and discarded until a valid message appears. Multiple MIDI messages can be recorded in succession in a single buffer for sending together. MIDI messages are not required to be complete at the end of the buffer.

Argument

`start`

`DN_MIDI_SNDDATA(= 0)`

`buf`

Start address of the buffer where send data is stored

`size`

buffer size (in bytes)

No write when 0 is designated. The size that can be written at that moment is returned (T-Kernel/SM specification).

Return Code (`tk_swri_dev` only)

`(E_IO | E_MIDI_TMO)` Transmission handling timed out from the settings in `DN_MIDI_SETTMO`.

`For other errors, see the T-Kernel specification.`

4.4.2 Setting the transmission and reception timeout

Sets the transmission and reception handling timeout in milli-second for the relevant MIDI port. Regardless of the transmission or reception size of the read or write request, if the single request is not completed within the interval, timeout occurs. Data processed up to that point is valid. To disable timeout and enable a continued wait for transmission and reception, this value is set to 0 (the default).

Argument

`start`

`DN_MIDI_SETTMO`

`buf`

Pointer to the `UW`-type variable. Stores the timeout interval.

`size`

Must be `sizeof(UW)`.

4.4.3 Clearing the internal buffer

Clears the driver's ring buffer for transmission and reception. Executed on the basis of each MIDI port.

Argument

`start`

`DN_MIDI_CLRBUF`

`buf`

(no reference)

`size`

(no reference)

4.5 ID tk_rea_dev(ID dd, INT start, VP buf, INT size, TMO tmout) ER tk_srea_dev(ID dd, INT start, VP buf, INT size, INT *asize)

4.5.1 MIDI data reception

Provides a buffer and receives MIDI data of the designated size. The buffer must not be released until the request is complete or canceled.

When data is received from the device, the device driver completes the running status. If the received data is invalid as a MIDI message, writing is not executed to the ring buffer for reception in the driver. Reading and discarding occurs until a valid message appears. After processing is complete, depending on the designated size at the time of request, there are cases when the MIDI message is not complete at the end of the received data, but data reading can be continued with the next reception request. Even if for one reception request the MIDI message is read only partway, the remaining data bytes are not deleted by the driver.

In case this function is not executed sufficiently often with regard to the volume of messages sent from the MIDI device, the buffer is cleared by the driver if the driver's internal ring buffer for reception is flooded. Under these conditions, be aware that no notification is issued.

Argument

`start`

`DN_MIDI_RCVDATA(= 0)`

buf

Start address of the buffer where receive data is stored

size

Size of data to be received (in bytes)

No read when 0 is designated. The size that can be read at that moment is returned (T-Kernel/SM specification).

Return Code (tk_srea_dev only)

(E_IO | E_MIDI_TMO) Reception handling timed out from the settings in DN_MIDI_SETTMO.

For other errors, see the T-Kernel specification.

4.5.2 Getting the timeout for sending or receiving

Gets the current timeout (in ms) for transmission or reception with the relevant MIDI port. When no timeout has been set (and there is a sustained wait until the completion of processing), this is 0.

Argument

start

DN_MIDI_GETTMO

buf

Pointer to the UW-type variable. The timeout interval is stored.

size

Must be sizeof(UW).

4.5.3 Getting device information

Gets device information, including what ports are used for sending and receiving by this device descriptor. The same value would be got from any device descriptor of the subunit in one device.

Argument

start

DN_MIDI_GETDEVINFO

buf

Pointer to the start of the structure for storing the port mapping information

```
struct {
   W    nOutPortNum;
   W    nInPortNum;
} MidiDriverDevInfo;
```

`nOutPortNum`

Number of output ports for this device

`nInPortNum`

Number of input ports for this device

`size`

`sizeof(MidiDriverDevInfo)` must be given

4.5.4 Getting the port name (optional)

Gets the name of ports used for sending and receiving by this device descriptor. The port name can be retrieved from the USB String Descriptor or other sources as well, and it can be created from the fixed value in the driver.

The sting is terminated in "¥0" by the driver. If the port name is longer than the designated number of bytes, a portion of bytes equal to (size − 1) is written for the termination.

Argument

`start`

`DN_MIDI_GETPORTNAME`

`buf`

Start address of the buffer where the port name is stored

`size`

Size to be retrieved (in bytes)

If 0 is designated, no writing occurs and the port name size is returned.

4.5.5 Getting the device name (optional)

Gets the device name, including what ports are used for sending and receiving by this device descriptor. This is the same value when browsed from any device descriptor of the subunit in one device.

The device name can be retrieved from the USB String Descriptor or other sources as well, and it can be created from the fixed value in the driver. Because with USB devices, for example, there may be cases when the device structure dynamically changes or multiple devices of the same type are connected, it is advisable that the device name include the serial number to identify specific devices.

The sting is terminated in "¥0" by the driver. If the port name is longer than the designated number of bytes, a portion of bytes equal to (size − 1) is written for the termination.

Argument

`start`

`DN_MIDI_GETDEVNAME`

`buf`

Start address of the buffer where the device name is stored

`size`

Size to be retrieved (in bytes)

If 0 is designated, no writing occurs and the device name size is returned.

5 References

• MIDI 1.0 Standard / MIDI Standard & Recommended Practices (Japanese Edition)
Written and published by the Association of Musical Electronics Industry (AMEI)
Sold by Rittor Music, Inc.
ISBN4-8456-0348-9

• MIDI Bible 1 Basic Edition, MIDI Bible 2 Applied Edition (Japanese)
Edited by Rittor Music Publishing, Editorial Department
ISBN4-8456-0267-9 / ISBN4-8456-0303-9

• Universal Serial Bus Device Class Definition for MIDI Devices
http://www.usb.org/developers/devclass_docs#approved